Inteligencia artificial avanzada

PID_00250574

Raúl Benítez
Andrés Cencerrado Barraqué
Gerard Escudero
Samir Kanaan

Tiempo mínimo de dedicación recomendado: 19 horas

Universitat Oberta de Catalunya

Índice

Introducción

Este módulo está organizado de la forma siguiente: los métodos de búsqueda y optimización se describen en el apartado 2, donde se detallarán las técnicas de extracción de información de bases de datos que contengan información semántica, como por ejemplo web de noticias o las conversaciones entre diversos miembros de una red social.

Las técnicas de caracterización de datos se estudiarán en el apartado 3, describiendo las técnicas principales basadas en descomposición de los datos en modos principales. En el apartado 3 también se estudiarán las técnicas de extracción de características y un método de visualización de datos multidimensionales.

Los algoritmos de clasificación de datos se presentan en el apartado 4, en el que se estudiarán los principales métodos de clasificación y reconocimiento de patrones.

En el apartado 5 se explican algunas técnicas avanzadas de inteligencia evolutiva, algoritmos que utilizan reglas heurísticas inspiradas en el funcionamiento evolutivo de los sistemas biológicos.

Finalmente, en el apartado 6 se introducen las técnicas y arquitecturas más importantes de las redes neuronales y el aprendizaje profundo, que es el campo de la IA que mayor auge tiene actualmente.

1. Introducción a la inteligencia artificial (IA)

1.1. Neuronas y transistores

Empezaremos planteando la pregunta filosófica fundamental, y así podremos dedicar nuestros esfuerzos a aspectos de carácter cientifico-técnico.

¿Es físicamente posible que una máquina presente capacidad de abstracción similar a la inteligencia humana?

Para responder esta pregunta, hay que tener en cuenta que el cerebro humano es el sistema de reconocimiento de patrones más complejo y eficiente que conocemos. Los humanos realizamos acciones tan sorprendentes como identificar a un conocido entre la multitud o reconocer de oído el solista de un concierto para violín. En el cerebro humano, las funciones cognitivas se realizan mediante la activación coordinada de unas 90.000.000.000 células nerviosas interconectadas mediante enlaces sinápticos. La activación neuronal sigue complejos procesos biofísicos que garantizan un funcionamiento robusto y adaptativo, y nos permite realizar funciones como el procesado de información sensorial, la regulación fisiológica de los órganos, el lenguaje o la abstracción matemática.

La neurociencia actual todavía no aporta una descripción detallada sobre cómo la activación individual de las neuronas da lugar a la formación de representaciones simbólicas abstractas. Lo que sí parece claro es que en la mayoría de procesos cognitivos existe una separación de escalas entre la dinámica a nivel neuronal y la aparición de actividad mental abstracta. Esta separación de escalas supone la ruptura del vínculo existente entre el hardware (neuronas) y el software de nuestro cerebro (operaciones abstractas, estados mentales), y constituye la hipótesis de partida para que los símbolos abstractos puedan ser manipulados por sistemas artificiales que no requieran un substrato fisiológico natural. La posibilidad de manipular expresiones lógicas y esquemas abstractos mediante sistemas artificiales es la que permite la existencia de lo que conocemos como *inteligencia artificial*.

Lecturas complementarias

C. Koch (1999). *Biophysics of Computation: Information Processing in Single Neurons*. USA: Oxford University Press.

Por supuesto, el cerebro no es el único sistema físico en el que se produce una separación de la dinámica a diferentes escalas. Esta característica también se observa en otros muchos sistemas complejos que presentan fenómenos de autoorganización no lineal. De la misma forma que es posible describir las co-

rrientes oceánicas sin necesidad de referirse al movimiento microscópico de las moléculas de agua, el pensamiento abstracto puede analizarse sin necesidad de referirse la activación eléctrica cerebral a nivel neuronal.

En cualquier caso, una de las cuestiones de mayor relevancia y aún no resueltas de la neurociencia actual es saber si existen procesos mentales -como la conciencia, la empatía o la creatividad-, que estén intrínsecamente ligados a la realidad biofísica del sistema nervioso humano y sean por tanto inaccesibles a un sistema artificial.

Otro aspecto importante en el funcionamiento del cerebro humano es el papel que tiene la experiencia y el aprendizaje. El cerebro humano actual no es solo resultado de una evolución biológica basada en alteraciones genéticas, sino también del conjunto de técnicas y conocimientos que la humanidad ha ido acumulando con el tiempo. Aspectos como la cultura o el lenguaje, transmitidas de generación en generación, también determinan la forma en la que se establecen patrones de activación neuronal en nuestro cerebro, y por lo tanto, contribuyen a la emergencia de procesos de abstracción en los que se basan áreas como las matemáticas o la literatura. A nivel biológico, existen diversos mecanismos que permiten la existencia de procesos de activación neuronal dependientes de la experiencia previa y del entrenamiento. El mecanismo principal es conocido como *plasticidad sináptica*, un fenómeno por el que las conexiones sinápticas entre neuronas modulan su intensidad en función de la actividad que hayan experimentado previamente. De esta forma, cuanto más veces se active un cierto canal de activación neuronal, más fácil resultará activarlo en el futuro e integrarlo a nuevos procesos cognitivos de mayor complejidad.

La plasticidad neuronal es la base de la mayoría de procesos de aprendizaje y memoria. A este paradigma se le conoce como *aprendizaje reforzado*, ya que la actividad sináptica se refuerza en función del número de veces que se establece una conexión entre neuronas. Esta regla –que relaciona la actividad neuronal con la función cognitiva–, se le conoce como *regla de Hebb* por los trabajos del neuropsicólogo canadiense Donald O. Hebb publicados en su libro de 1949 *The organization of behavior*. Algunas técnicas de inteligencia artificial como los métodos de *aprendizaje supervisado* también se basan en reglas similares que permiten modificar de forma adaptativa la forma en que el sistema artificial procesa la información.

La inteligencia artificial (IA) es una disciplina académica relacionada con la teoría de la computación cuyo objetivo es emular algunas de las facultades intelectuales humanas en sistemas artificiales. Con inteligencia humana nos referimos típicamente a procesos de percepción sensorial (visión, audición, etc.) y a sus consiguientes procesos de reconocimiento de patrones, por lo que las

aplicaciones más habituales de la IA son el tratamiento de datos y la identificación de sistemas. Eso no excluye que la IA, desde sus inicios en la década del 1960, haya resuelto problemas de carácter más abstracto como la demostración de teoremas matemáticos, la adquisición del lenguaje, el jugar a ajedrez o la traducción automática de textos. El diseño de un sistema de inteligencia artificial normalmente requiere la utilización de herramientas de disciplinas muy diferentes como el cálculo numérico, la estadística, la informática, el procesado de señales, el control automático, la robótica o la neurociencia. Por este motivo, pese a que la inteligencia artificial se considera una rama de la informática teórica, es una disciplina en la que contribuyen de forma activa numerosos científicos, técnicos y matemáticos. En algunos aspectos, además, se beneficia de investigaciones en áreas tan diversas como la psicología, la sociología o la filosofía.

Pese a que se han producido numerosos avances en el campo de la neurociencia desde el descubrimiento de la neurona por Santiago Ramón y Cajal a finales del siglo XIX, las tecnologías actuales están muy lejos de poder diseñar y fabricar sistemas artificiales de la complejidad del cerebro humano. De hecho, a día de hoy estamos lejos de reproducir de forma sintética las propiedades electroquímicas de la membrana celular de una sola neurona. Pero como hemos comentado anteriormente, la manipulación de conceptos y expresiones abstractas no está supeditada a la existencia de un sistema biológico de computación. En definitiva, un ordenador no es más que una máquina que procesa representaciones abstractas siguiendo unas reglas predefinidas.

Avances de la microelectrónica

En la actualidad, la única tecnología que permite implementar sistemas de inteligencia artificial son los sistemas electrónicos basados en dispositivos de estado sólido como el transistor. En efecto, gracias a los grandes avances de la microelectrónica desde los años setenta, los ordenadores actuales disponen de una gran capacidad de cálculo que permite la implementación de sistemas avanzados de tratamiento de datos y reconocimiento de patrones. Los sistemas digitales basados en transistor constituyen una tecnología rápida, robusta y de tamaño reducido que permite la ejecución secuencial de operaciones aritmético-lógicas. En muchas aplicaciones concretas, los sistemas artificiales pueden llegar a presentar un rendimiento notablemente mejor que el del propio cerebro humano. Este es el caso, por ejemplo, en aquellas situaciones que requieran gestionar grandes cantidades de datos o que exigan una rápida ejecución de cálculos matemáticos.

Un sistema de inteligencia artificial requiere de una secuencia finita de instrucciones que especifique las diferentes acciones que ejecuta la computadora para resolver un determinado problema. Esta secuencia de instrucciones constituye la *estructura algorítmica* del sistema de inteligencia artificial.

Se conoce como *método efectivo o algoritmo* al procedimiento para encontrar la solución a un problema mediante la reducción del mismo a un conjunto de reglas.

En ocasiones, los sistemas de IA resuelven problemas de forma *heurística* mediante un procedimiento de ensayo y error que incorpora información relevante basada en conocimientos previos. Cuando un mismo problema puede resolverse mediante sistemas naturales (cerebro) o artificiales (computadora), los algoritmos que sigue cada implementación suelen ser completamente diferentes puesto que el conjunto de instrucciones elementales de cada sistema son también diferentes. El cerebro procesa la información mediante la activación coordinada de redes de neuronas en áreas especializadas (cortex visual, cortex motor, etc.). En el sistema nervioso, los datos se transmiten y reciben codificados en variables como la frecuencia de activación de las neuronas o los intervalos en los que se generan los potenciales de acción neuronales. El elevado número de neuronas que intervienen en un proceso de computación natural hace que las fluctuaciones fisiológicas tengan un papel relevante y que los procesos computacionales se realicen de forma estadística mediante la actividad promediada en subconjuntos de neuronas.

En un sistema IA, en cambio, las instrucciones básicas son las propias de una computadora, es decir operaciones aritmetico-lógicas, de lectura/escritura de registros y de control de flujo secuencial. La tabla 1 describe las diferencias fundamentales entre sistemas de inteligencia artificial y natural en las escalas más relevantes.

Tabla 1. Comparación entre inteligencia natural y artificial a diferentes niveles

Nivel	Natural	Artificial
Abstracción	Representación y manipulación de objetos abstractos	Representación y manipulación de objetos abstractos
Computacional	Activación coordinada de áreas cerebrales	Algoritmo / procedimiento efectivo
Programación	Conexiones sinápticas plasticidad	Secuencia de operaciones artimético-lógicas
Arquitectura	Redes excitatorias e inhibitorias	CPU + memoria
Hardware	Neurona	Transistor

La idea principal es que, a pesar de las enormes diferencias entre sistemas naturales y artificiales, a un cierto nivel de abstracción ambos pueden describirse como sistemas de procesado de objetos abstractos mediante un conjunto de reglas.

1.2. Breve historia de la IA

El nacimiento de la IA como disciplina de investigación se remonta a 1956, durante una conferencia sobre informática teórica que tuvo lugar en el Dartmouth College (Estados Unidos). A esa conferencia asistieron algunos de los científicos que posteriormente se encargaron de desarrollar la disciplina en

diferentes ámbitos y de dotarla de una estructura teórica y computacional apropiada. Entre los asistentes estaban John McCarthy, Marvin Minsky, Allen Newell y Herbert Simon. En la conferencia, A. Newell y H. Simon presentaron un trabajo sobre demostración automática de teoremas al que denominaron *Logic Theorist*. El Logic Theorist fue el primer programa de ordenador que emulaba características propias del cerebro humano, por lo que es considerado el primer sistema de ingeligencia artificial de la historia. El sistema era capaz de demostrar gran parte de los teoremas sobre lógica matemática que se presentaban en los tres volúmenes de los *Principia Mathematica* de Alfred N. Whitehead y Bertrand Russell (1910-1913).

Minsky y McCarthy fundaron más tarde el laboratorio de inteligencia artificial del Massachusets Institute of Technology (MIT), uno de los grupos pioneros en el ámbito. La actividad de los años cincuenta es consecuencia de trabajos teóricos de investigadores anteriores como Charles Babbage (autor de la Máquina analítica, 1842), Kurt Gödel (teorema de incompletitud, 1930), Alan Turing (máquina universal, 1936), Norbert Wiener (cibernética, 1943) y John von Newmann (arquitectura del computador, 1950). La arquitectura de von Newmann consta de una unidad central de proceso (CPU) y de un sistema de almacenamiento de datos (memoria), y fue utilizada en 1954 por RAND Corporation para construir JOHNIAC (John v. Neumann Numerical Integrator and Automatic Computer), una de las primeras computadoras en las que más tarde se implementaron sistemas de inteligencia artificial como el Logic Theorist de Newell y Simon.

En 1954 también apareció el IBM 704, la primera computadora de producción en cadena, y con ella se desarrollaron numerosos lenguajes de programación específicamente diseñados para implementar sistemas de inteligencia artificial como el LISP. Junto con estos avances, se produjeron los primeros intentos para determinar la presencia de comportamiento inteligente en una máquina. El más relevante desde el punto de vista histórico fue propuesto por Alan Turing en un artículo de 1950 publicado en la revista *Mind* y titulado *Computing Machinery and Intelligence*.

> En este trabajo se propone un test de inteligencia para máquinas según el cual una máquina presentaría un comportamiento inteligente en la medida en que fuese capaz de mantener una conversación con un humano sin que otra persona pueda distinguir quién es el humano y quién el ordenador. Aunque el *test de Turing* ha sufrido innumerables adaptaciones, correcciones y controversias, pone de manifiesto los primeros intentos de alcanzar una definición objetiva de la inteligencia.

En este contexto, es de especial relevancia el *Teorema de incompletitud de Gödel* de 1931, un conjunto de teoremas de lógica matemática que establecen las li-

mitaciones inherentes a un sistema basado en reglas y procedimientos lógicos (como lo son todos los sistemas de IA).

Tras los primeros trabajos en IA de los años cincuenta, en la década de los sesenta se produjo un gran esfuerzo de formalización matemática de los métodos utilizados por los sistemas de IA.

HAL 9000

El impacto social de la IA durante la década de los sesenta se pone de manifiesto en la película "2001: A Space Odyssey", dirigida en 1968 por Stanley Kubrick y basada en una novela homónima de ciencia ficción de Arthur C. Clarke. El protagonista principal de la película es HAL 9000 (Heuristically programmed ALgorismic computer), un ordenador de dotado de un avanzado sistema de inteligencia artificial que es capaz de realizar tareas como mantener una conversación, reconocimiento de voz, lectura de labios, jugar a ajedrez e incluso manifestar un cierto grado de sensibilidad artística.

Los años setenta, en parte como respuesta al test de Turing, se produjo el nacimiento de un área conocida como *procesado del lenguaje natural* (NLP, Natural Language Processing), una disciplina dedicada a sistemas artificiales capaces de generar frases inteligentes y de mantener conversaciones con humanos. El NLP ha dado lugar a diversas áreas de invetigación en el campo de la lingüística computacional, incluyendo aspectos como la desambigüación semántica o la comunicación con datos incompletos o erróneos. A pesar de los grandes avances en este ámbito, sigue sin existir una máquina que pueda pasar el test de Turing tal y como se planteó en el artículo original. Esto no es tanto debido a un fracaso de la IA como a que los intereses del área se han ido redefiniendo a lo largo de la historia. En 1990, el controvertido empresario Hugh Loebner y el Cambridge Center for Behavioral Studies instauraron el *premio Loebner*, un concurso anual ciertamente heterodoxo en el que se premia al sistema artificial que mantenga una conversación más indistinguible de la de un humano. Hoy en dia, la comunidad científica considera que la inteligencia artificial debe enfocarse desde una perspectiva diferente a la que se tenía en los años cincuenta, pero iniciativas como la de Loebner expresan el impacto sociológico que sigue teniendo la IA en la sociedad actual.

Aplicaciones de la IA

En la actualidad, la IA se ha consolidado como una disciplina que permite diseñar aplicaciones de gran utilidad práctica en numerosos campos. Actualmente, existe una enorme lista de ámbitos de conocimiento en los que se utilizan sistemas de IA, entre los que son de especial relevancia la minería de datos, el diagnóstico médico, la robótica, la visión artificial, el análisis de datos bursátiles o la planificación y logística.

En los años ochenta empezaron a desarrollarse las primeras aplicaciones comerciales de la IA, fundamentalmente dirigidas a problemas de producción, control de procesos o contabilidad. Con estas aplicaciones aparecieron los primeros sistemas expertos, que permitían realizar tareas de diagnóstico y toma de decisiones a partir de información aportada por profesionales expertos. En torno a 1990, IBM construyó el ordenador ajedrecista Deep Blue, capaz de plantarle cara a un gran maestro de ajedrez utilizando algoritmos de búsqueda y análisis que le permitían valorar cientos de miles de posiciones por segundo.

Más allá del intento de diseñar robots humanoides y sistemas que rivalicen con el cerebro humano en funcionalidad y rendimiento, el interés hoy en día es diseñar e implementar sistemas que permitan analizar grandes cantidades de datos de forma rápida y eficiente. En la actualidad, cada persona genera y recibe a diario una gran cantidad de información no sólo a través de los canales clásicos (conversación, carta, televisión) sino mediante nuevos medios que nos permiten contactar con más personas y transmitir un mayor número de datos en las comunicaciones (Internet, fotografía digital, telefonía móvil). Aunque el cerebro humano es capaz de reconocer patrones y establecer relaciones útiles entre ellos de forma excepcionalmente eficaz, es ciertamente limitado cuando la cantidad de datos resulta excesiva. Un fenómeno similar

ocurre en el ámbito empresarial, donde cada día es más necesario barajar cantidades ingentes de información para poder tomar decisiones. La aplicación de técnicas de IA a los negocios ha dado lugar a ámbitos de reciente implantación como la inteligencia empresarial, *business intelligence* o a la minería de datos, *data mining*. En efecto, hoy más que nunca la información está codificada en masas ingentes de datos, de forma que en muchos ámbitos se hace necesario extraer la información relevante de grandes conjuntos de datos antes de proceder a un análisis detallado.

En resumen, hoy en día el objetivo principal de la inteligencia actual es el tratamiento y análisis de datos.

Algunas veces nos interesará caracterizar los datos de forma simplificada para poder realizar un análisis en un espacio de dimensión reducida o para visualizar los datos de forma mas eficiente. Por ejemplo, puede ser interesante saber qué subconjunto de índices bursátiles internacionales son los más relevantes para seguir la dinámica de un cierto producto o mercado emergente. En otras ocasiones, el objetivo será identificar patrones en los datos para poder clasificar las observaciones en diferentes clases que resulten útiles para tomar decisiones respecto un determinado problema. Un ejemplo de este segundo tipo de aplicación sería el análisis de imágenes médicas para clasificar a pacientes según diferentes patologías y así ayudar al médico en su diagnóstico. Por último, en muchos casos se hace necesario realizar búsquedas entre una gran cantidad de datos u optimizar una determinada función de coste, por lo que también será necesario conocer métodos de búsqueda y optimización. En esta clase de problemas encontramos, por ejemplo, el diseño de los horarios de una estación de trenes de forma que se minimice el tiempo de espera y el número de andenes utilizados.

1.3. Ámbitos de aplicación de la inteligencia artificial

Las aplicaciones más frecuentes de la inteligencia artificial incluyen campos como la robótica, el análisis de imágenes o el tratamiento automático de textos. En *robótica*, uno de los campos de investigación actual con mayor proyección es el del *aprendizaje adaptativo*, en el que un sistema robotizado explora diferentes configuraciones con el objetivo de realizar un movimiento complejo (caminar, agarrar un objeto, realizar una trayectoria, jugar a golf, etc.). El objetivo podría ser, por ejemplo, que un robot cuadrúpedo se levante y ande de forma autónoma, de forma que siga un proceso de exploración y aprendizaje similar al que realiza un recién nacido durante los primeros meses de vida. Un sistema de IA en este caso se encargaría de explorar diferentes movimientos de forma aleatoria, midiendo las variables de cada articulación y comprobando en cada momento el grado de éxito alcanzado por cada secuencia de

acciones (altura del centro de masas, desplazamiento horizontal, fluctuaciones en la posición vertical, velocidad de desplazamiento, etc.). El sistema de IA modularía la ejecución de las diferentes acciones, incrementando la probabilidad de aquellas que presenten un mejor rendimiento y restringiendo las que no comporten una mejora de la función objetivo. Un área afín a la robótica es la de las *interfaces cerebro-computadora* (BCI, Brain-computer Interfaces), sistemas artificiales que interactúan con el sistema nervioso mediante señales neurofisiológicas con el objetivo asistir a personas discapacitadas durante la ejecución de determinadas tareas motoras.

Dentro del campo de la IA, una de las ramas con mayor proyección son los denominados *sistemas expertos*, en los que el objetivo es diseñar un sistema que permita analizar un conjunto de datos y realizar tareas típicamente asociadas a la figura de un profesional experto como el diagnóstico, la detección de fallos, la planificación o la toma de decisiones. Los datos con los que trabaja el sistema experto pueden ser de naturaleza muy diversa.

En un sistema de diagnóstico clínico, por ejemplo, se puede partir de imágenes radiológicas, de una serie temporal de la frecuencia del ritmo cardíaco de un paciente o de un conjunto de datos con valores extraídos de análisis de sangre o de orina. La utilización de todas las señales anteriores a la vez constituye un sistema de *fusión multimodal*, en el que el estado clínico del paciente se describe desde una perspectiva multiorgánica más completa. En los sistemas expertos se combina información extraída de datos con el conocimiento del sistema que aporta un experto especializado. A estos sistemas se les conoce como *sistemas basados en conocimiento* (KBS, *knowledge-based systems*), y permiten integrar reglas heurísticas y árboles de decisiones elaborados por una comunidad de expertos durante años de trabajo y experimentación. Estas reglas no pueden ser inferidas directamente de los datos observados, y son de gran utilidad en aplicaciones sobre diagnóstico y toma de decisiones.

Ejemplo

Un ejemplo de este tipo de técnicas serían el protocolo de diagnóstico que realiza un médico especialista a partir de un conjunto de pruebas, o los criterios que aplica un ingeniero de caminos para validar la resistencia mecánica de un puente.

La información aportada por humanos expertos es también necesaria para diseñar sistemas artificiales que jueguen a juegos como el ajedrez o el go.

Go

Go es un tradicional juego de mesa chino que se practica en un tablero reticular y tiene por objetivo ocupar con las fichas una región mayor que el adversario. El desarrollo de máquinas que jueguen a Go con humanos a un alto nivel es un ámbito en el que actualmente se dedican grandes esfuerzos desde el campo de la IA. En este juego, el gran número de posibles movimientos en cada jugada impide aplicar técnicas de búsqueda global, por lo que los sistemas inteligentes deben incorporar información sobre estrategias y tácticas aportada por jugadores humanos expertos.

El *análisis de textos* es otro ejemplo interesante en el que se desarrollan numerosos sistemas de IA. Aspectos como la traducción automática de textos han evolucionado de forma sorprendente durante los últimos años gracias a técnicas de IA. Hoy en día es fácil encontrar plataformas web gratuitas que permitan traducir textos entre más de 50 lenguas diferentes. Estos sistemas de traducción automática presentan resultados de notable calidad, y tienen en cuenta aspectos semánticos y contextuales antes de proponer una traducción del texto. El éxito de estas plataformas se debe en gran parte a sistemas de IA que utilizan enormes cantidades de información textual aportada por los usuarios del servicio. Sistemas similares permiten realizar búsquedas inteligentes en la web, de forma que los vínculos que se ofrecen tengan en cuenta las preferencias estadísticas propias y del resto de usuarios, o enviar publicidad de forma selectiva a partir de la información que aparece en el campo asunto de nuestro buzón de correo electrónico.

Otro ejemplo de sistema en el ámbito de la *ingeniería de procesos* es un sistema de detección de fallos en una planta compleja. Una planta industrial dispone de múltiples sensores distribuidos que permiten monitorizar el proceso de forma continua. En la fase de entrenamiento, el sistema aprende un conjunto de patrones dinámicos de los sensores que corresponden a situaciones en los que se produce un error en la planta. En el futuro, cuando se produce un error, el sistema de IA detecta su existencia de forma automática y es capaz de *diagnosticar* el origen del fallo comparando la respuesta de los sensores con las respuestas características de cada error. En una tercera fase, el sistema puede incluso tomar decisiones sobre qué acciones hay que realizar para resolver el problema de la forma más rápida y eficiente. Las aplicaciones industriales de la IA es un campo con una gran proyección en el que los sistemas van dirigidos a mejorar procesos de fabricación, control de calidad, logística o planificación de recursos.

En muchas ocasiones, los sistemas de IA constan de dos fases, una primera fase de *aprendizaje* y una segunda de *predicción*. En la fase de aprendizaje se aporta un conjunto de datos representativo de aquellas situaciones que se desea analizar, de forma que el sistema IA aprende las características fundamentales de los datos y es capaz de *generalizar* su estructura. Dicha generalización no es más que la construcción de un modelo de los datos que permita realizar una predicción acertada a partir de nuevas observaciones.

Reconocimiento de caras

Consideremos el ejemplo de un sistema automático para el reconocimiento de caras. Al sistema se le proporcionan 100 imágenes faciales de 10 personas diferentes, 10 caras por persona. Las imágenes son tomadas en diferentes condiciones (diferente expresión facial, ropa, iluminación, exposición, fondo, etc.), de forma que sean representativas de las características faciales de cada persona. El sistema identifica las características principales de cada una de las fotos y es capaz de agruparlas en un determinado número de grupos (presumiblemente 10, excepto si se han incluido gemelos monocigóticos). Entre las características utilizadas pueden aparecer, entre otras, el color de los ojos, el grosor de los labios o el perímetro de la cabeza. En la fase de predicción, se parte de una imagen de uno de los 10 individuos que no haya sido incluida en el conjunto de datos de entrenamiento. El sistema calcula las características faciales del nuevo individuo, y debe

ser capaz de identificar esa imagen como perteneciente a uno de los grupos definidos durante el proceso de aprendizaje (presumiblemente al grupo del individuo correcto, a no ser que la nueva imagen no tenga apenas rasgos comunes con las 10 imágenes de entrenamiento). Si la nueva imagen es identificada de forma correcta, diremos que el sistema de IA acierta en su predicción, mientras que cuando el sistema asigna la imagen a otro grupo de forma errónea, se produce un error de clasificación.

Tabla 2. Principales ámbitos de aplicación de los sistemas de inteligencia artificial

Área	Aplicaciones
Medicina	Ayuda al diagnóstico
	Análisis de imágenes biomédicas
	Procesado de señales fisiológicas
Ingeniería	Organización de la producción
	Optimización de procesos
	Cálculo de estructuras
	Planificación y logística
	Diagnóstico de fallos
	Toma de decisiones
Economía	Análisis financiero y bursátil
	Análisis de riesgos
	Estimación de precios en productos derivados
	Minería de datos
	Marketing y fidelización de clientes
Biología	Análisis de estructuras biológicas
	Genética médica y molecular
Informática	Procesado de lenguaje natural
	Criptografía
	Teoría de juegos
	Lingüística computacional
Robótica y automática	Sistemas adaptativos de rehabilitación
	Interfaces cerebro-computadora
	Sistemas de visión artificial
	Sistemas de navegación automática
Física y matemáticas	Demostración automática de teoremas
	Análisis cualitativo sistemas no-lineales
	Caracterización de sistemas complejos

2. Recomendadores y agrupamientos

Actualmente la gran mayoría de instrumentos de medida de cualquier tipo de magnitud (desde magnitudes físicas como longitud, temperatura, tiempo, hasta magnitudes de comportamiento como patrones de búsqueda y navegación en la web y preferencias de compra *online*, pasando por herramientas comunes como las cámaras digitales) son capaces de volcar sus mediciones a algún formato digital; de esa forma todos esos datos pueden estar fácilmente disponibles para su tratamiento.

En este momento el problema no es disponer de datos, pues se dipone en gran abundancia de ellos; el reto es conseguir extraer información a partir de los datos, o sea, darles un sentido y extraer conclusiones útiles de ellos. Esta tarea se conoce por el nombre de **minería de datos** (*data mining*).

Uno de los principales retos en el procesamiento de datos es el de integrar los datos procedentes de múltiples fuentes, para así dotar de diferentes perspectivas al conjunto de estos, lo que permite extraer información más rica. Esta tendencia se da en casi todas las áreas: recopilación de la actividad de los usuarios en un sitio web; integración de sensores de temperatura, presión atmosférica y viento en el análisis meteorológico; uso de diferentes datos financieros y bursátiles en la previsión de inversiones; entre otros ejemplos.

La tarea de integración de múltiples fuentes de datos recibe el nombre de **filtrado colaborativo** (*collaborative filtering*).

De la misma forma que en las ciencias experimentales es fundamental utilizar un instrumental adecuado y unas unidades consistentes, un aspecto clave en cualquier tarea de procesamiento de datos es el uso de **métricas**, o sea medidas de distancia, adecuadas para el tipo de datos que se está tratando.

Ved también

En el subapartado 2.1 de este módulo se estudian algunas de las métricas más habituales.

Una de las aplicaciones en las que se centra este módulo es la de los **recomendadores**. Un recomendador es un sistema que recoge y analiza las preferencias de los usuarios, generalmente en algún sitio web (comercios, redes sociales, sitios de emisión o selección de música o películas, etc.). La premisa básica de

Ved también

En el subapartado 2.2 se describen algunas estrategias sencillas para construir recomendadores.

los recomendadores es que usuarios con actividad o gustos similares continuarán compartiendo preferencias en el futuro. Al recomendar a un usuario productos o actividades que otros usuarios con gustos similares han elegido previamente el grado de acierto acostumbra a ser más elevado que si las recomendaciones se basan en tendencias generales, sin personalizar.

La tarea de encontrar a los usuarios más afines y utilizar esta información para predecir sus preferencias puede inscribirse en una tarea más general que recibe el nombre de **agrupamiento** (*clustering*), y que consiste en encontrar la subdivisión óptima de un conjunto de datos, de forma que los datos similares pertenezcan al mismo grupo.

Ved también

En el subapartado 2.3 se presentan los métodos de agrupamiento más importantes.

2.1. Métricas y medidas de similitud

Una **métrica** es una función que calcula la distancia entre dos elementos y que por tanto se utiliza para medir cuán diferentes son. Existen varias formas de medir la distancia entre dos elementos, y elegir la métrica adecuada para cada problema es un paso crucial para obtener buenos resultados en cualquier aplicación de minería de datos.

A menudo se utilizan funciones que miden la similitud entre dos elementos en lugar de su distancia.

En este subapartado utilizaremos la **distancia euclídea** y estudiaremos una medida de similitud habitual, la **correlación de Pearson**.

Por último, una forma habitual y segura* de convertir una función de distancia d en una función de similitud s es la siguiente:

$$s(P,Q) = \frac{1}{1 + d(P,Q)} \quad (1)$$

Ved también

Las distancias y similitudes se utilizan en gran cantidad de métodos, por lo que en este módulo se presentan otras métricas como la distancia de Hamming en el subapartado 4.3.1 y la información mutua en el subapartado 3.1.3.

*** Segura porque no incurre en divisiones por cero si la distancia es cero.**

2.1.1. Ejemplo de aplicación

En un sitio web de visualización de películas a la carta se recoge la valoración de cada usuario sobre las películas que va viendo, con el objetivo de poder proponer a los usuarios las películas que más se adapten a sus gustos. Tras ver una película, un usuario ha de dar una valoración entre 1 y 5, donde las valoraciones más bajas corresponden a películas que han disgustado al usuario, y las más alta a las que le han gustado más. Se desea descubrir similitudes entre usuarios de manera que a cada usuario se le propongan las películas que más han gustado a los usuarios con gustos más parecidos al suyo.

Para poner en práctica estas pruebas se utilizarán los conjuntos de datos disponibles en *https://grouplens.org/datasets/movielens/*, concretamente el conjun-

to de 100 k valoraciones (fichero *ml-100k.zip*), que contiene cien mil valoraciones (del 1 al 5) de 1.700 películas realizadas por mil usuarios. Si bien el conjunto de datos contiene 23 ficheros, en este módulo solo se utilizará el fichero *u.data*, que es el que contiene las valoraciones de los usuarios. En la tabla 3 se muestra un ejemplo meramente ilustrativo de valoraciones. Como se puede observar, los usuarios no tienen por qué valorar todas las películas, sino solo las que han visto.

En el fichero *u.data* cada valoración aparece en una fila, y las columnas son «IdUsuario», «IdPelícula», «valoración» y «fecha».

Tabla 3. Valoraciones de ocho películas (ejemplo)

IdUsuario/IdPelícula	1	2	3	4	5	6	7	8
1		3	1		4	3	5	
2	4	1	3		5			2
3	2	1		5				1
4	3		2			5		4

Los datos se presentan en forma matricial por legibilidad y concisión.

2.1.2. Distancia euclídea

La distancia euclídea de dos puntos $P = (p_1,p_2,...,p_n)$ y $Q = (q_1,q_2,...,q_n)$ en el espacio n-dimensional $\mathbb{R}^n$ viene dada por la fórmula:

$$d(P,Q) = \sqrt{\sum_{i=1}^{n}(p_i - q_i)^2} \qquad (2)$$

La distancia euclídea no es más que la generalización a n dimensiones del teorema de Pitágoras. Si las distancias en sí no son importantes, sino sólo su comparación, a menudo se utiliza la **distancia euclídea cuadrada**, es decir, sin la raíz cuadrada, pues la comparación entre distancias euclídeas cuadradas da los mismos resultados que entre las distancias euclídeas y puede resultar mucho más rápida de calcular, ya que la raíz cuadrada es una operación computacionalmente costosa.

Dados dos vectores numéricos, es muy sencillo calcular su distancia euclídea utilizando la función *euclidean* de la biblioteca SciPy, tal y como se muestra en el código 2.1. De la misma forma, a partir de la distancia euclídea se puede derivar fácilmente una similitud de acuerdo con la expresión 1.

SciPy

SciPy es un conjunto de bibliotecas científicas para Python que incluye, entre otras, NumPy (manejo de matrices), Matplotlib (generación de gráficas), Pandas (estructuras de datos), SymPy (procesamiento simbólico) y muchas funciones científicas. Podéis descargarlas desde su web: https://www.scipy.org/.

Código 2.1: distancia y similitud euclídea entre dos vectores

```
1 import numpy as np
2
3 from scipy.spatial.distance import euclidean
4
5 a = np.array([5, 3, 4, -1, 0])
6 b = np.array([2, 4, 0, -5, -2])
```

```
7
8  dist = euclidean(a, b)
9
10 print('Distancia =', dist)
11
12 simil = 1 / (1 + dist)
13
14 print('Similitud =', simil)
```

En la tabla 4 se muestran las similitudes entre las valoraciones de algunos usuarios utilizando la distancia euclídea. Como se puede comprobar, $s(P,P) = 1$, ya que $d(P,P) = 0$) y $s(P,Q) = s(Q,P)$, o sea, la matriz de similitudes es simétrica.

Tabla 4. Similitud euclídea de las valoraciones

Id usuario	1	2	3	4
1	1,0	0,25	0,33	0,31
2	0,25	1,0	0,31	0,29
3	0,33	0,31	1,0	0,24
4	0,31	0,29	0,24	1,0

Supongamos que el usuario 4 desea que se le recomiende alguna película; el sistema, de momento, puede sugerirle que vea las películas que han gustado al usuario 1, ya que es el usuario más similar al 4; en el ejemplo, el sistema recomendaría las películas 5 y 7 al usuario 4, que son las que más le han gustado al usuario 1 y que aún no ha visto el 4.

Una limitación de la similitud (y la distancia) euclídea es que es muy sensible a la escala: si un usuario tiende a puntuar las películas que le gustan con un 4 y otro con un 5, habrá una cierta distancia entre ellos, en especial si el número de valoraciones es alto, aunque en el fondo ambos usuarios comparten gustos a pesar de utilizar las puntuaciones de formas diferentes.

Además, a mayor número de dimensiones (valoraciones, en este caso), la distancia euclídea tiende a ser mayor, lo que puede distorsionar los resultados. Por ejemplo, si dos usuarios comparten 2 valoraciones y hay una diferencia de 1 entre cada una de ellas, su distancia será $d = \sqrt{1^2 + 1^2} = 1{,}41$; sin embargo, si comparten 5 valoraciones con una diferencia de 1 entre ellas, su distancia será $d = \sqrt{1^2 + 1^2 + 1^2 + 1^2 + 1^2} = 2{,}24$; una distancia bastante mayor para unas valoraciones en apariencia similares. Cuantas más valoraciones compartan dos usuarios (aunque no sean muy diferentes), más alejados estarán, lo que parece contrario a la lógica.

Utilidad de la distancia euclídea

La distancia euclídea es una métrica útil en numerosas aplicaciones, en especial si las magnitudes son lineales y su escala es uniforme; además, es sencilla y rápida de calcular.

2.1.3. Correlación de Pearson

El coeficiente de correlación de Pearson es una medida de similitud entre dos variables que resuelve los problemas de la similitud euclídea. Se trata de una

medida de cómo las dos variables, una frente a otra, se organizan en torno a una línea recta (línea de mejor ajuste), tal y como se puede ver en la figura 1. Cuanto más similares son las valoraciones de dos usuarios, más se parecerá su recta a la recta $y = x$, ya que las valoraciones serán de la forma (1,1), (3,3), (4,4), etc.

Figura 1. Correlación entre las valoraciones los usuarios 1 y 2 de la tabla 3

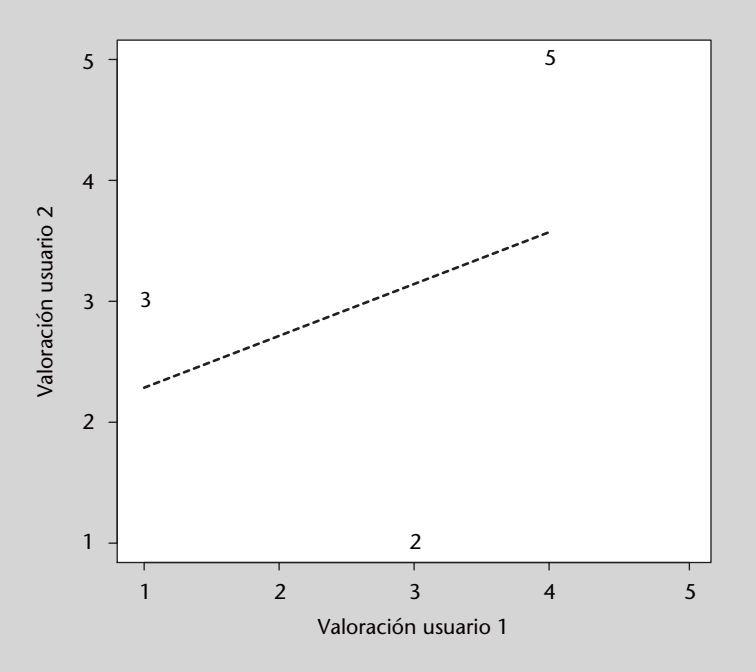


Figura 1

Diagrama de dispersión (*scatter plot*) de las valoraciones de dos usuarios: se representa la valoración de cada película en común tomando el eje x como la valoración de un usuario, y el eje y como la valoración del otro usuario. Los números en el diagrama corresponden a los identificadores de las películas comunes.

El coeficiente de correlación de Pearson (en este subapartado, simplemente *correlación*) está relacionado con la pendiente de la recta representada en la figura 1, y puede tomar un valor en el rango [–1,1]. Si su valor es 1 indica que las dos variables están perfectamente relacionadas; si es 0, no hay relación lineal entre ellas*; si es negativo es que existe una correlación negativa, en este caso que las valoraciones de un usuario son opuestas a las del otro.

*** El coeficiente de correlación de Pearson sólo mide relaciones lineales; aunque valga 0, puede haber relaciones no lineales entre las dos variables.**

El cálculo del coeficiente de correlación de Pearson sobre dos muestras de datos alineados (valoraciones de usuarios, en nuestro caso) x_i e y_i viene dado por la fórmula:

$$r_{xy} = \frac{\sum_{i=1}^{n}(x_i - \overline{x})(y_i - \overline{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \overline{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \overline{y})^2}} \qquad (3)$$

donde $\overline{x}$ es la media de los valores de x y $\overline{y}$ la media de los valores de y.

Nótese que para efectuar el cálculo los datos deben estar alineados: en nuestro caso, solo se deben tomar las valoraciones comunes a los dos usuarios. También hay que prever que el denominador pueda valer cero. Con todas esas

consideraciones, en el código 2.2 se muestra el código en Python que calcula el coeficiente de correlación de Pearson entre dos vectores.

Código 2.2: coeficiente de Pearson entre dos vectores

```
1  import numpy as np
2
3  from scipy.spatial.distance import correlation
4
5  a = np.array([5, 3, 4, -1, 0])
6  b = np.array([2, 4, 0, -5, -2])
7
8  correl = correlation(a, b)
9
10 print('Correlacion a y b =', correl)
```

Como se deduce de su definición, el coeficiente de correlación de Pearson es simétrico y vale 1 al calcularlo respecto a la misma variable. En la tabla 5 puede verse los valores correspondientes a las valoraciones de los usuarios de la tabla 3. En este ejemplo la correlación entre los usuarios 1 y 3 es 0 porque sólo tienen una película en común, con lo que la recta de ajuste no se puede definir.

Tabla 5. Coeficiente de Pearson de las valoraciones

Id usuario	1	2	3	4
1	1,0	0,33	0,0	1,0
2	0,33	1,0	0,95	–0,5
3	0,0	0,95	1,0	–1,0
4	1,0	–0,5	–1,0	1,0

También se observa que los usuarios 1 y 4 tienen una correlación de 1,0; eso es debido a que, independientemente de factores de escala, su tendencia a valorar las películas es igual. Nótese la diferencia con la similitud euclídea mostrada en la tabla 4, si bien el usuario más similar al 4 sigue siendo el 1 y por lo tanto las películas que podría elegir serían las mismas. Por otra parte, los usuarios 3 y 4 hacen valoraciones contrarias, por lo que su coeficiente es –1,0. No obstante, sería conveniente disponer de más datos para obtener medidas más realistas.

El coeficiente de correlación de Pearson resulta útil como medida de similitud porque es independiente de los desplazamientos y escalas de los valores estudiados; para conseguir estas propiedades utilizando la similitud euclídea sería necesario normalizar los datos previamenente a su análisis. Uno de sus principales inconvenientes en su uso en filtrado colaborativo es que requiere que haya al menos dos valores comunes para poder dar un resultado significativo; eso limita su aplicación en aquellos casos en que se desea sugerir productos similares a un cliente que ha hecho una única elección.

2.2. Sistemas recomendadores

Los **sistemas recomendadores** (*recommender systems*) tienen como objetivo sugerir a los usuarios productos, contenidos, grupos, amistades, etc., que se adecúen a sus gustos. El funcionamiento básico de estos sistemas es el siguiente: en primer lugar, se almacenan las elecciones y valoraciones de los usuarios; a continuación, se comparan las preferencias de los diferentes usuarios para determinar cuáles son aquellos con gustos afines; finalmente, se calculan nuevas sugerencias para un usuario a partir de las preferencias de los usuarios cuyos gustos son más parecidos.

2.2.1. Conceptos generales

Como los sistemas de recomendación son sistemas complejos que integran varias etapas de procesamiento de datos, existen diferentes criterios a la hora de diseñar un sistema de este tipo. A continuación veremos la tipología más habitual en las que se suelen clasificar los recomendadores:

1) Tareas: en primer lugar hay que hacer una distinción entre varias tareas relacionadas. La primera es la **valoración** de elementos (*item rating*), que consiste en asignar un valor de calidad a un elemento (un producto, una canción, etc.). La segunda tarea es la de **clasificación**, en el sentido de clasificación en un evento deportivo (*item ranking*), en la que los elementos previamente valorados se ordenan para obtener los más relevantes. Por último, se puede hablar de sistemas recomendadores, una tarea en la que se proponen a un usuario sus elementos más relevantes para que los compre o los utilice de la forma adecuada. Así pues, generalmente los sistemas recomendadores contienen sistemas de valoración y de clasificación de elementos.

2) Estrategia: otro criterio importante al diseñar un sistema recomendador es elegir la estrategia general que se seguirá, definida en gran medida por la información utilizada para generar las recomendaciones. Se distinguen tres estrategias:

- Basada en conocimiento: utiliza conocimiento específico del área de aplicación para diseñar un recomendador personalizado. En definitiva, las reglas de valoración y de clasificación se diseñan a mano teniendo en cuenta la experiencia en el negocio. La solución solo sirve para una tarea o un negocio muy específicos y requieren un buen conocimiento de las preferencias de los clientes.

- Basada en contenidos: utiliza las características de los elementos que se recomiendan para sugerir elementos similares. Un sistema basado en contenidos usaría, por ejemplo, información como el género musical (*rock*, pop, *jazz*, etc.), el tipo de película (acción, comedia, drama, etc.), las características de un coche (tipo, rango de precio, potencia, etc.). Generalmente, el

usuario especifica uno o más elementos «semilla» y el sistema busca elementos parecidos de acuerdo con las características que contempla. Así funcionan sitios como Internet Movie Database (IMDB), Pandora Radio, etc. El problema es que el sistema suele producir recomendaciones dentro del mismo estilo, no sorprende al usuario con elementos nuevos pero que le pueden gustar.

- Filtrado colaborativo (*collaborative filtering*): en lugar de usar las características de los productos, lo que hace un sistema de filtrado colaborativo es identificar usuarios con preferencias parecidas y recomendar los productos comprados o consultados por los usuarios más similares. Así funcionan sitios como Amazon. La ventaja es que permiten descubrir productos nuevos pero que sin embargo pueden interesar al usuario. La desventaja es la complejidad de los métodos necesarios, en especial por el hecho de que los usuarios solo valoran una parte mínima de todos los elementos disponibles, y eso requiere usar métodos especiales para tratar con datos tan dispersos. Esta será la estrategia que seguiremos en estos materiales.

3) Información almacenada: otro criterio importante al diseñar un sistema recomendador es qué información almacenar. Idealmente se puede trabajar con toda la información disponible (todas las compras hechas por todos los usuarios), pero este planteamiento puede tener requisitos computacionales muy altos. Es lo que se conoce como **recomendadores basados en memoria**.

La alternativa son los **recomendadores basados en modelos**, en los que los datos se «resumen» para quedarse con la información más relevante y, por ejemplo, para formar grupos de usuarios en lugar de almacenar todos y cada uno de los usuarios individualmente. Así, un nuevo usuario se asigna al grupo más parecido y a partir de ahí se generan las recomendaciones.

En primer lugar veremos métodos basados en memoria y en el subapartado 2.3 estudiaremos métodos de agrupamiento que permiten generar recomendadores basados en modelos.

2.2.2. La biblioteca Surprise

La biblioteca Surprise* ofrece gran número de funciones y estructuras de datos adecuadas para la generación de sistemas recomendadores en Python. Además, permite cargar de forma muy sencilla algunos conjuntos de datos estándar y también ofrece funciones de evaluación de los resultados. Por esas razones la utilizaremos en los ejemplos siguientes.

* http://surpriselib.com/

En primer lugar, veremos cómo cargar el conjunto de datos habitual en ejemplos de recomendadores, MovieLens en su versión 100 k. Si bien Surprise permite cargar automáticamente esos datos, en este ejemplo descargaremos los ficheros y veremos cómo cargarlos con Surprise. Así veremos el proceso genérico de carga de datos, lo que os capacitará para trabajar con otros conjuntos de datos en el futuro.

La lectura de datos en Surprise se hace generando un objeto Reader, en el que se especifica el significado de las columnas del fichero de datos. Al utilizarlo para cargar el fichero, se genera un objeto Python al que se pueden aplicar directamente los diferentes métodos de Surprise. El siguiente programa muestra cómo cargar los datos de Movielens-100k, suponiendo que el fichero *u.data* se encuentra en la carpeta indicada.

Código 2.3: carga de datos con Surprise

```
1  import surprise.dataset
2
3  fich  = 'ml-100k/u.data'
4
5  # Los nombres de las columnas son predefinidos y tienen significado
6  # para Surprise, asi que no se pueden cambiar
7  columnas = 'user item rating timestamp'
8
9  lector=surprise.dataset.Reader(line_format = columnas, sep='\t')
10 datos=surprise.dataset.Dataset.load_from_file(fich,reader=lector)
11
12 datos.split(n_folds = 5)
```

Además, la última línea del código anterior divide los datos en cinco bloques (*folds*), lo que permite realizar experimentos con validación: se toman cuatro bloques como datos de entrada para el método elegido, y se comprueban los resultados aplicados al bloque restante. Esto se repite para cada uno de los cinco bloques. Este procedimiento estándar de desarrollo y ajuste de métodos de IA se denomina validación cruzada (*cross validation*). Al dividir los datos en bloques hay que ser cuidadoso para que la distribución de las características en cada bloque sea lo más parecida posible a la del conjunto de datos; por ejemplo, que no queden todos los aficionados a la ciencia ficción en uno de los bloques.

A partir de aquí, se puede utilizar el objeto Datos para aplicar los métodos que veremos a continuación.

2.2.3. Vecinos más cercanos

Uno de los métodos de clasificación más sencillos es el llamado *k* vecinos más cercanos (*k-nearest neighbours* o simplemente kNN). La idea es bastante intuitiva: dado un usuario para el que queremos generar recomendaciones, busca los *k* usuarios más semejantes en la base de datos y utiliza sus valoraciones para generar una serie de recomendaciones personalizadas.

Dentro de kNN hay muchas variantes posibles: la función para calcular la similitud entre usuarios (distancia euclídea, correlación, etc.); cuántos vecinos se seleccionan, es decir el valor de *k*; si se normalizan las valoraciones de los usuarios; se puede dar más peso a las valoraciones de los usuarios más similares, etc.

Surprise permite controlar algunas de estas variantes de forma sencilla. Concretamente ofrece funciones con diferentes variantes de kNN, así como algunos parámetros para regular su funcionamiento. Las variantes y los parámetros más relevantes son los siguientes:

- Parámetro **k**: número de vecinos que el algoritmo tiene en cuenta al generar la recomendación.

- Valoración media: la versión básica de kNN es la función **KNNBasic**, que no tiene en cuenta los valores medios de valoración, pero la función **KNNWithMeans** sí que calcula las valoraciones medias de cada elemento antes de generar las recomendaciones.

- La función **KNNBaseline**, por otra parte, tiene en cuenta los niveles de valoración de cada usuario para generar recomendaciones más precisas, calculando el nivel basal (*baseline*) de las valoraciones y por tanto calculando cuáles son las que realmente destacan respecto a ese nivel base.

- Por último, el parámetro **sim_options**, común a las funciones mencionadas antes, permite controlar qué función de similitud se utiliza y cuál es su configuración.

El fragmento de código siguiente utiliza el objeto Datos generado anteriormente para entrenar un recomendador y calcular un par de valoraciones para pares de usuario y producto utilizando KNNBasic; simplemente cambiando la función se pueden utilizar las otras variantes de kNN disponibles.

Código 2.4: generación de valoraciones individuales

```
# Extraer el subconjunto de entrenamiento y entrenar kNN con el
entrenamiento = datos.build_full_trainset()

metodo    = surprise.KNNBasic()
metodo.train(entrenamiento)

# Estimar una valoracion que ya existe
usuario  = str(196)
elemento = str(242)
prediccion = metodo.predict(usuario, elemento, r_ui=3)
print('Estimada =', prediccion.est, ' real=', prediccion.r_ui)

# Estimar una valoracion que no existe en los datos (la pareja
# usuario/elemento no se encuentra)
usuario  = str(196)
elemento = str(302)
prediccion = metodo.predict(usuario, elemento)
print('Estimada =', prediccion.est)
```

Antes de utilizar un sistema recomendador en una aplicación, es necesario saber cuál es la calidad de sus resultados para estimar el grado de fiabilidad que tiene, así como para introducir unas mejoras en los métodos y parámetros que permitan incrementar esa fiabilidad.

Como se acaba de mencionar, el procedimiento estándar de evaluación de un sistema de aprendizaje automático es la **validación cruzada**, en la que los datos se han dividido en varios bloques (*folds*) y de forma iterativa se usan todos menos uno para entrenar el sistema y con el bloque restante se mide la calidad de los resultados. El objeto Datos contiene los datos de MovieLens divididos en cinco bloques, como se ha visto en el código 2.3; es decir, está preparado para utilizarse en la evaluación de métodos. Surprise incluye métodos de evaluación que utilizan automáticamente la validación cruzada para proporcionar resultados. Por ejemplo, si queremos probar otra variante de kNN con algunos parámetros y estudiar la calidad de los resultados, podemos hacer lo siguiente:

Código 2.5: validación cruzada

```
# kNN con diferente k y utilizando la correlacion de Pearson como
# medida de similitud
similOpts = {'name': 'pearson'}

metodo2 = surprise.KNNWithMeans(k = 20, sim_options = similOpts)
metodo2.train(entrenamiento)

# Se evalua el metodo2 usando RMSE y MAE como medidas de calidad
results=surprise.evaluate(metodo2,datos,measures=['RMSE','MAE'])

surprise.print_perf(results)
```

En este caso se utilizan dos medidas de calidad para evaluar los resultados. La raíz del error medio cuadrado (*root mean squared error*, RMSE) calcula la raíz cuadrada de la diferencia entre el valor esperado $\hat{y}_i$ y el calculado por el recomendador y_i elevada al cuadrado, dividido por el número de elementos n, es decir:

$$RMSE = \sqrt{\frac{\sum_{i=1}^{n}(\hat{y}_i - y_i)^2}{n}} \tag{4}$$

Se trata de una medida estándar para cuantificar la diferencia entre dos series de valores. El error medio absoluto (*mean absolute error*, MAE) es otra medida similar que calcula el valor absoluto entre el valor esperado y el calculado:

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|\hat{y}_i - y_i| \tag{5}$$

Lógicamente, en ambos casos se trata de medidas de error, lo que quiere decir que nuestro objetivo será reducirlas al mínimo; un recomendador con menor RMSE/MAE es mejor que otro con mayores valores de error.

El resultado de ejecutar el código anterior es una tabla con los valores de RMSE y MAE calculados sobre las valoraciones calculadas por el método respecto a los datos reales existentes en el conjunto de datos de validación, es decir, cada uno de los bloques que se descartan para realizar la evaluación. La última columna incluye el valor medio de RMSE y MAE sobre los cinco bloques. Esto

permite comparar diferentes métodos y configuraciones para encontrar el más adecuado.

Tabla 6. Resultados de la evaluación de un recomendador

Bloque	1	2	3	4	5	Media
RMSE	0.9528	0.9559	0.9623	0.9613	0.9586	0.9582
MAE	0.7471	0.7468	0.7556	0.7532	0.7512	0.7508

2.2.4. Descomposición en valores singulares

La técnica de descomposición en valores singulares (*singular value decomposition*, SVD) es una potente técnica matemática cuyos fundamentos se describen en el subapartado 3.1.1. En este apartado no entraremos en detalles, sino que simplemente veremos por qué es una técnica utilizada en recomendadores.

Sea M la matriz de valoraciones, en la que las filas corresponden a los usuarios y las columnas a cada elemento valorado (películas, libros, etc.). Así, el elemento M_{ij} será la valoración que el usuario i hace del elemento j. La descomposición SVD de M permite estimar las valoraciones de los usuarios aunque no estén definidas en el conjunto de datos. Es muy sencillo utilizar SVD con Surprise, como se ve en el ejemplo siguiente.

Código 2.6: recomendador con SVD

```
1 metodo3 = surprise.SVD()
2 metodo3.train(entrenamiento)
3
4 results=surprise.evaluate(metodo3,datos,measures=['RMSE','MAE'])
5 surprise.print_perf(results)
```

El método SVD de Surprise tiene quince parámetros que conviene ajustar para conseguir los mejores resultados posibles. Hacer ese ajuste manualmente resultaría extremadamente tedioso, y por ese motivo Surprise permite explorar diferentes combinaciones de parámetros de forma automática y comparar los resultados. Esta técnica se conoce como búsqueda en rejilla (*grid search*), ya que se prueban todas las combinaciones de los parámetros indicados. Veamos un ejemplo de cómo hacer esto con Surprise y SVD.

Código 2.7: ajuste de parámetros

```
1  paramRejilla = {'n_epochs':[5, 10], 'lr_all':[0.002, 0.005],
2                  'reg_all': [0.4, 0.6]}
3
4  busqueda=surprise.GridSearch(surprise.SVD, paramRejilla,
5                               measures=['RMSE'])
6
7  busqueda.evaluate(datos)
8
9  # Mostrar mejor resultado y parametros con los que se ha conseguido
10 print(busqueda.best_score['RMSE'])
11 # >>> 0.96067593271
12 print(busqueda.best_params['RMSE'])
13 # >>> {'reg_all': 0.4, 'n_epochs': 10, 'lr_all': 0.005}
```

En este ejemplo se han explorado diferentes valores para tres parámetros del proceso de optimización interna de SVD: el número de iteraciones (parámetro *n_epochs*), la tasa de aprendizaje de todos los parámetros (*lr_all*, algo así como la velocidad a la que aprende) y el factor de regularización, que controla si el aprendizaje se ajusta muy estrictamente a los datos disponibles o por el contrario si intenta encontrar soluciones más generales (*reg_all*).

Hay que tener en cuenta que este tipo de exploración prueba todas las combinaciones posibles de los valores dados, así que el tiempo de ejecución se puede disparar si se intentan combinar muchos parámetros. Al final se puede obtener cuál ha sido la mejor configuración, para así poderla utilizar en siguientes aplicaciones.

2.2.5. Conclusiones

Los recomendadores vistos en este subapartado permiten mejorar de una forma muy sencilla sitios web y otras aplicaciones similares para sintonizar mejor con los usuarios y sugerirles productos que les puedan interesar, mejorando así tanto su satisfacción con la aplicación como los éxitos potenciales de venta o acceso.

Las ventajas de estos métodos es que permiten generar valoraciones de productos nuevos que sugerir a los usuarios de una forma bastante acertada, lo que tiene muchas aplicaciones y es un área de negocio e investigación muy activa. Además, son métodos que tratan adecuadamente las características de los datos en este tipo de aplicaciones, principalmente las valoraciones que suelen ser muy dispersas, ya que cada usuario solo valora una fracción muy pequeña del total de elementos.

Las limitaciones de los métodos vistos hasta ahora son que, como se ha explicado, son métodos basados en memoria: requieren almacenar y procesar todos los datos cada vez que se realiza una consulta. De este modo, una operación sencilla no resulta excesivamente costosa, pero como se tiene que repetir completamente en cada consulta puede dar lugar a una carga computacional y de memoria grande en aplicaciones medianas y grandes.

Otra limitación fundamental es que los métodos de recomendación que se acaban de estudiar no abstraen los datos de ninguna forma, es decir, no proporcionan información global acerca de los datos de los que se dispone, con lo que su uso está limitado a producir recomendaciones de productos, pero no pueden utilizarse para abordar estudios más complejos sobre los tipos de usuarios o productos de que se dispone, estudios que son esenciales para analizar el funcionamiento de la aplicación y diseñar su futuro. A continuación se estudiarán métodos que sí abstraen y producen información de alto nivel a partir de los datos disponibles.

2.3. Algoritmos de agrupamiento (*clustering*)

2.3.1. Ejemplo de aplicación

Una empresa de juegos en línea está analizando el comportamiento de sus clientes con el fin de ofrecerles los productos y características más adecuados a sus intereses. En este estudio se miden dos características: número de horas diarias jugando y número de horas diarias charlando (chateando) con otros jugadores, ambas en valor promedio para cada jugador. Los objetivos son dos: el primero, determinar qué tipos o clases de jugadores existen, según su actividad; el segundo, clasificar a los nuevos jugadores en alguna de esas clases para poder ofrecerles las condiciones y productos más adecuados. El segundo objetivo se trata extensamente en el apartado 3 dedicado a la clasificación, mientras que en este subapartado nos centraremos en cómo organizar un conjunto de datos extenso en unas cuantas clases o grupos, desconocidos *a priori*.

En la figura 2 se muestran los datos recogidos de 2.084 jugadores. Se trata de datos sintéticos (generados artificialmente) con el propósito de que ilustren más claramente los métodos que se expondrán a continuación. Concretamente, se pueden observar cinco clases o grupos claramente diferenciados, cada uno marcado con un símbolo diferente. En los datos reales no suele observarse una separación tan clara, y por consiguiente la ejecución de los métodos de agrupamiento no produce resultados tan definidos.

Los datos de estos jugadores están disponibles en los materiales de la asignatura.

Figura 2. Actividad de los jugadores en línea

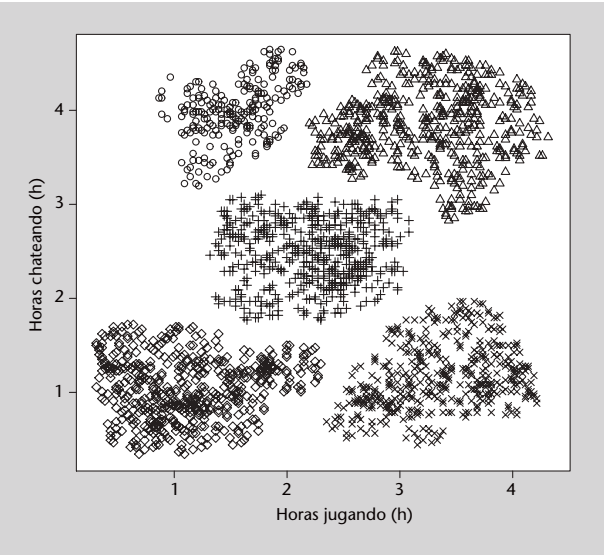


Figura 2

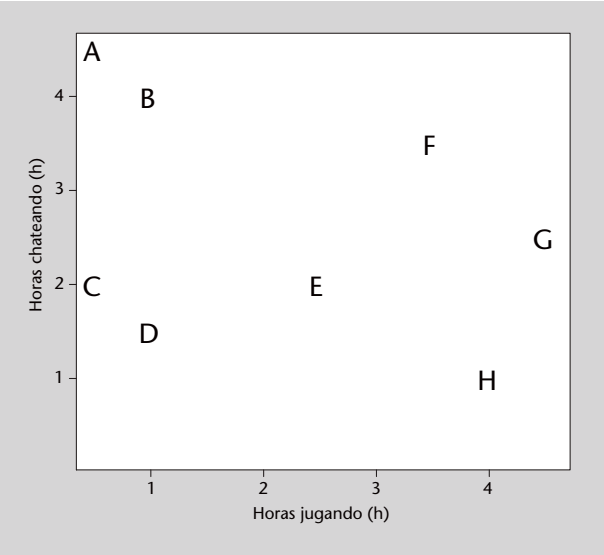
Las dos variables utilizan las mismas unidades (horas) y magnitudes similares y por tanto pueden utilizarse directamente para agrupar; en un caso más general, en el que las variables tengan magnitudes diferentes, suele ser necesario normalizarlas para que su magnitud se equipare y no tengan más influencia unas que otras (salvo que se pretenda justamente eso por la naturaleza del problema).

De hecho, para mostrar la ejecución paso a paso de algunos algoritmos de agrupamiento es necesario utilizar muchos menos datos, que en este caso son los que se muestran en la tabla 7 y en la figura 3.

Tabla 7. Jugadores de ejemplo

Jugador	Horas juego	Horas chat
A	0,5	4,5
B	1	4
C	0,5	2
D	1	1,5
E	2,5	2
F	3,5	3,5
G	4,5	2,5
H	4	1

Figura 3. Actividad de 8 jugadores de ejemplo



2.3.2. Conceptos generales

La tarea de agrupamiento de datos es una tarea **no supervisada**, ya que los datos que se le proporcionan al sistema no llevan asociada ninguna etiqueta o información añadida por un revisor humano; por el contrario, es el propio método de agrupamiento el que debe descubrir las nuevas clases o grupos a partir de los datos recibidos.

Si bien los algoritmos de agrupamiento se han introducido en el subapartado anterior como herramienta para crear recomendadores, sus aplicaciones van mucho más allá y se emplean en numerosos ámbitos, como por ejemplo:

- Procesamiento de imagen, para separar unas zonas de otras (típicamente con imágenes de satélite).

- Agrupamiento de genes relacionados con una determinada característica o patología.

- Agrupamiento automático de textos por temas.

- Definir tipos de clientes y orientar las estrategias comerciales en función de esos grupos.

- En general, definir grupos en conjuntos de datos de los que no se conoce una subdivisión previa: astronomía, física, química, biología, medicina, farmacología, economía, sociología, psicología, etc.

A menudo el agrupamiento de los datos precede a la clasificación de nuevos datos en alguno de los grupos obtenidos en el agrupamiento; por esa razón, el agrupamiento y la clasificación de datos están estrechamente relacionados, como se podrá observar en este subapartado, que comparte algunas técnicas con el subapartado 4.3.

Los algoritmos de agrupamiento se pueden clasificar en dos tipos: los **jerárquicos** son progresivos, es decir que van formando grupos progresivamente, y se estudiarán en primer lugar; los **particionales** sólo calculan una partición de los datos en k grupos: los restantes algoritmos estudiados en este subapartado pertenecen a esta categoría.

Utilizando algoritmos de agrupamiento es posible construir recomendadores **basados en modelos**, llamados así porque a diferencia de los recomendadores basados en datos no necesitan almacenar todos los datos de que se dispone, sino que producen una abstracción de los datos dividiéndolos en grupos; para producir una recomendación sólo es necesario asociar un usuario o producto a un grupo existente, por lo que la información a almacenar se reduce a la descripción de los grupos obtenidos previamente. De esta forma la generación de una recomendación se simplifica notablemente, y por tanto los recursos requeridos para generarla.

Más allá de los recomendadores, una de las características más importantes de los algoritmos de agrupamiento es que permiten organizar datos que en principio no se sabe o puede clasificar, evitando criterios subjetivos de clasificación, lo que produce una información muy valiosa a partir de datos desorganizados.

Una característica común a casi todos los algoritmos de agrupamiento es que no son capaces de determinar por sí mismos el número de grupos idóneo, sino que hay que fijarlo de antemano o bien utilizar algún criterio de cohesión para saber cuándo detenerse (en el caso de los jerárquicos). En general ello requiere probar con diferentes números de grupos hasta obtener unos resultados adecuados.

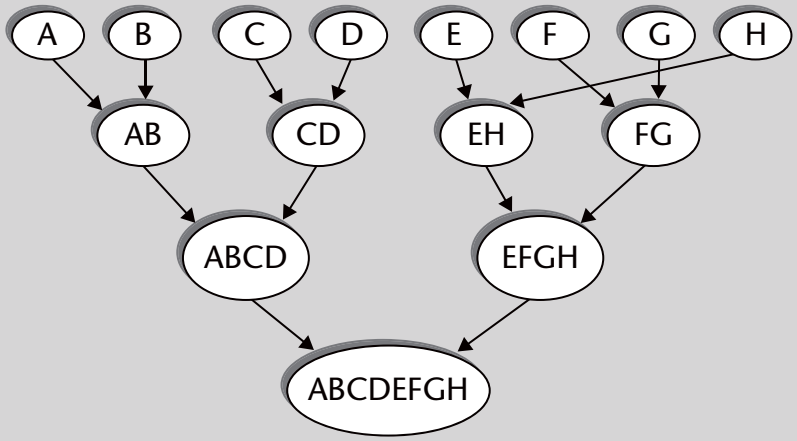
2.3.3. Agrupamiento jerárquico. Dendrogramas

Hay dos tipos de algoritmos de agrupamiento jerárquicos. El algoritmo **aglomerativo** parte de una fragmentación completa de los datos (cada dato tiene su propio grupo) y fusiona grupos progresivamente hasta alcanzar la situación contraria: todos los datos están reunidos en un único grupo. El algoritmo **divisivo**, por su parte, procede de la forma opuesta: parte de un único grupo que contiene todos los datos y lo va dividiendo progresivamente hasta tener un grupo para cada dato.

El **dendrograma*** es un diagrama que muestra las sucesivas agrupaciones que genera (o deshace) un algoritmo de agrupamiento jerárquico. En la figura 4 puede verse el dendrograma resultante de aplicar agrupamiento aglomerativo a los datos mostrados en la tabla 7.

***O diagrama de árbol, del griego *dendron*, árbol, y *gramma*, dibujo.**

Figura 4. Dendrograma de los 8 jugadores de ejemplo de la tabla 7



En los algoritmos jerárquicos de agrupamiento, se parte del estado inicial y se aplica un criterio para decidir qué grupos unir o separar en cada paso, hasta que se alcance el estado final. Si bien conceptualmente ambos tipos de algoritmos jerárquicos (aglomerativo y divisivo) son equivalentes, en la práctica el algoritmo aglomerativo es más sencillo de diseñar por la razón de que sólo hay una forma de unir dos conjuntos, pero hay muchas formas de dividir un conjunto de más de dos elementos.

Criterios de enlace

En cada paso de un algoritmo de agrupamiento jerárquico hay que decidir qué grupos unir (o dividir). Para ello hay que determinar qué grupos están más cercanos, o bien qué grupo está menos cohesionado. Supongamos que

estamos programando un algoritmo aglomerativo; la distancia entre dos grupos puede determinarse según diferentes expresiones, que reciben el nombre de **criterios de enlace**. Algunos de los criterios más utilizados para medir la distancia entre dos grupos A y B son los siguientes:

- Distancia máxima entre elementos de los grupos (enlace completo):

$$max\{d(x,y) : x \in A, y \in B\}$$

- Distancia mínima entre elementos de los grupos (enlace simple):

$$min\{d(x,y) : x \in A, y \in B\}$$

- Distancia media entre elementos de los grupos (enlace medio):

$$\frac{1}{|A||B|} \sum_{x \in A} \sum_{y \in B} d(x,y)$$

Una desventaja del criterio de enlace simple es que puede provocar que un único elemento fuerce la unión de todo su conjunto con otro conjunto que, por lo demás, no sea especialmente cercano. Por esa razón en general se preferirán otros criterios (completo, medio u otros).

Código del algoritmo aglomerativo

En el programa siguiente se muestra cómo utilizar las funciones de las librerías SciPy, NumPy y Matplotlib para generar un agrupamiento jerárquico y mostrar el dendrograma. En el ejemplo se utiliza un enlace completo y una distancia euclídea, pero en SciPy se ofrecen otras opciones con las que podéis experimentar.

En primer lugar, se muestra una función que carga los datos desde el fichero *players.data* que se han descrito anteriormente. El resultado se almacena en una matriz de NumPy para poderla usar a continuación en el agrupamiento.

Código 2.8: carga de datos de los jugadores

```
1 import numpy
2
3 # Lee un fichero con los datos de los jugadores, devuelve una
4 # matriz con dos columnas, horasJuego y horasChat
5 # (cada fila es un jugador)
6 def leeJugadores(nombreFichero):
7     with open(nombreFichero) as fichero:
8         lineas = [(l.strip()).split('\t') for l in fichero]
9
```

```
    horasJuego = [float(x[0]) for x in lineas]
    horasChat  = [float(x[1]) for x in lineas]

    matriz = numpy.column_stack((horasJuego, horasChat), )

    return matriz


# Leer los datos de los jugadores
datos = leeJugadores('data/players.data')
```

En las líneas siguientes se importan las funciones necesarias para crear un diagrama de árbol o **dendrograma** y para generar el agrupamiento. Concretamente, la función *linkage* genera una matriz de enlace en la que se recoge todo el proceso de agrupamiento jerárquico paso a paso, ya que cada fila de esa columna define un enlace entre dos elementos o grupos, primero los más cercanos atendiendo al criterio de enlace utilizado, hasta que finalmente se agrupan todos los elementos. En el código pueden verse algunas líneas de esta matriz y su formato. El hecho de disponer de una matriz de enlace nos permite seleccionar el grado de agrupamiento que deseemos, así como visualizar el agrupamiento.

Código 2.9: agrupamiento jerárquico

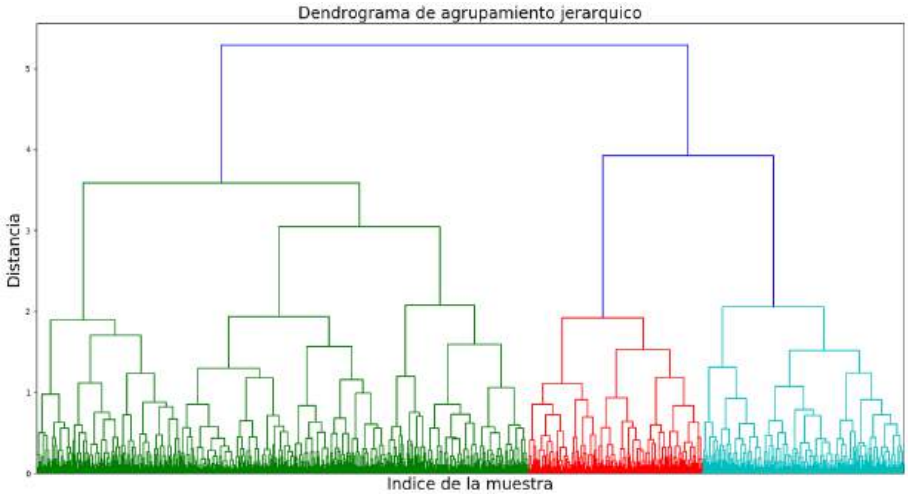
```
from scipy.cluster.hierarchy import dendrogram, linkage

# Generar la matriz de enlace (enlace completo)
enlace = linkage(datos, method = 'complete', metric = 'euclidean')

# El formato de la matriz de enlace es id1, id2, distancia, numero
# de puntos unidos, y representa que puntos se unen en cada paso
# del agrupamiento aglomerativo. Por ejemplo:
for i in [0, 1, 100, 1000, 2000, 2082]:
    print('id1=%d id2=%d dist=%.4f num=%d'%tuple(enlace[i,]))

# id1=14 id2=19 dist=0.0250 num=2
# id1=28 id2=33 dist=0.0250 num=2
# id1=769 id2=782 dist=0.0250 num=2
# id1=430 id2=2516 dist=0.0559 num=3
# id1=3787 id2=3993 dist=0.5154 num=22
# id1=4164 id2=4165 dist=5.2923 num=2084

# Los indices mayores al numero de elementos se refieren a los
# grupos formados por el algoritmo. Concretamente, es la linea
# en la que fueron creados (al numero hay que restarle el numero
# de elementos del conjunto, por ejemplo si el indice es 3000 y
# hay 2000 elementos, entonces se trata del grupo formado en la
# linea 1000).
```

Finalmente, se crea un dendrograma del agrupamiento obtenido mediante la función *dendrogram*, en la que se puede visualizar de forma clara de qué forma se van formando los grupos hasta obtener un único grupo al final del agrupamiento. En la figura 5 se puede ver el gráfico resultante.

Código 2.10: agrupamiento jerárquico

```
from matplotlib import pyplot as plt

# Mostrar el dendrograma o diagrama de arbol
plt.figure(figsize=(25, 10))
plt.title('Dendrograma de agrupamiento jerarquico', fontsize=20)
plt.xlabel('Indice de la muestra', fontsize=20)
plt.ylabel('Distancia', fontsize=20)

dendrogram(enlace, orientation = 'top', no_labels = True)

plt.show()
```

Figura 5



Conclusiones

El principal interés de los algoritmos jerárquicos radica en que generan diferentes grados de agrupamiento, y su evolución, lo que puede ser tan útil como la obtención de un agrupamiento definitivo. Además son sencillos conceptualmente y desde el punto de vista de la programación. Por otra parte, los algoritmos jerárquicos de agrupamiento actúan de forma **voraz** (*greedy*), ya que en cada paso toman la mejor decisión en ese momento, sin tener en cuenta la futura conveniencia de tal decisión. Por ese motivo, en general dan peor resultado que otros algoritmos de agrupamiento.

2.3.4. k-medios (*k-means*)

El algoritmo de agrupamiento k-medios busca una partición de los datos tal que cada punto esté asignado al grupo cuyo centro (llamado *centroide*, pues no tiene por qué ser un punto de los datos) sea más cercano. Se le debe indicar el número *k* de clusters deseado, pues por sí mismo no es capaz de determinarlo.

A grandes rasgos el algoritmo es el siguiente:

1) Elegir *k* puntos al azar como centroides iniciales. No tienen por qué pertenecer al conjunto de datos, aunque sus coordenadas deben estar en el mismo intervalo.

2) Asignar cada punto del conjunto de datos al centroide más cercano, formándose así *k* grupos.

3) Recalcular los nuevos centroides de los *k* grupos, que estarán en el centro geométrico del conjunto de puntos del grupo.

4) Volver al paso 2 hasta que las asignaciones a grupos no varíen o se hayan superado las iteraciones previstas.

El siguiente programa muestra cómo ejecutar k-medios sobre los datos de los jugadores. Si bien se trata de un algoritmo sencillo y rápido, al tener en cuenta sólo la distancia a los centroides puede fallar en algunos casos (nubes de puntos de diferentes formas o densidades), ya que por lo general tiende a crear esferas de tamaño similar que particionen el espacio. Así, para los datos mostrados en la figura 2, la partición en grupos aproximada es la mostrada en la figura 6, en la que se ve que, por ejemplo, algunos puntos del grupo superior derecho (triángulos) se han asignado al superior izquierdo (círculos) por estar más cercanos al centroide, sin tener en cuenta el espacio que los separa. El mismo problema se da en otros puntos. Además, su resultado puede variar de una ejecución a otra, pues depende de los centroides generados aleatoriamente en el primer paso del algoritmo.

Código 2.11: agrupamiento k-medios

```
1  from sklearn.cluster import KMeans
2
3  # Configurar el algoritmo para que calcule 4 grupos y aplicarlo
4  # a los datos
5  kmedios = KMeans(n_clusters = 4)
6  kmedios.fit(datos)
7
8  # Ver las etiquetas de grupo de los 100 primeros puntos
9  print(kmedios.labels_[:100])
10 # [0 0 0 ..., 0 0 2]
11
12 # Ver cuales son los centros de los 4 grupos
13 print(kmedios.cluster_centers_)
14 #[[ 3.22010753  3.92903226]
15 # [ 3.29138889  1.21      ]
16 # [ 2.02299107  2.9937872 ]
17 # [ 1.13938632  1.07887324]]
18
19
20 # Preguntar al agrupamiento a que grupo pertenecerian nuevos puntos
21 print(kmedios.predict([[2, 3], [4, 1]]))
22 # [2 1]
```

Figura 6. K-medios ejecutado sobre los datos de la figura 2

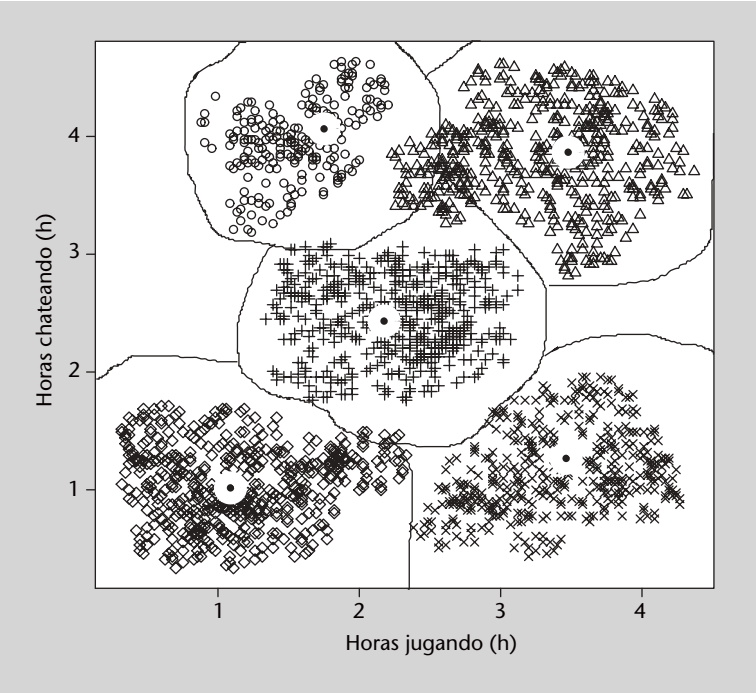


Figura 6

Los puntos negros rodeados de una zona blanca representan los centroides de los grupos obtenidos.

2.3.5. c-medios difuso (*Fuzzy c-means*)

Resulta lógico pensar que, en k-medios, un punto que esté junto al centroide estará más fuertemente asociado a su grupo que un punto que esté en el límite con el grupo vecino. El agrupamiento difuso (*fuzzy clustering*) representa ese diferente grado de vinculación haciendo que cada dato tenga un grado de pertenencia a cada grupo, de manera que un punto junto al centroide puede tener 0,99 de pertenencia a su grupo y 0,01 al vecino, mientras que un punto junto al límite puede tener 0,55 de pertenencia a su grupo y 0,45 al vecino. Esto es extensible a más de dos grupos, entre los que se repartirá la pertenencia de los datos. Se puede decir que el agrupamiento discreto (no difuso) es un caso particular del difuso en el que los grados de pertenencia son 1 para el grupo principal y 0 para los restantes.

El algoritmo de c-medios difuso es prácticamente idéntico al algoritmo k-medios visto anteriormente, las principales diferencias son:

- Cada dato x tiene asociado un vector de k valores reales que indican el grado de pertenencia $m_x(k)$ de ese dato a cada uno de los k grupos. El grado de pertenencia de un punto a un grupo depende de su distancia al centroide correspondiente. Habitualmente la suma de los grados de pertenencia de un dato es igual a 1.

- En lugar de crear k centroides aleatoriamente, se asigna el grado de pertenencia de cada punto a cada grupo aleatoriamente y después se calculan los centroides a partir de esa información.

- El cálculo de los centroides está ponderado por el grado de pertenencia de cada punto al grupo correspondiente $m_x(k)$.

A grandes rasgos las ventajas e inconvenientes de c-medios difuso son las mismas que las de k-medios: simplicidad, rapidez, pero no determinismo y excesiva dependencia de la distancia de los puntos a los centroides, sin tener en cuenta la densidad de puntos de cada zona (por ejemplo, espacios vacíos). Se trata de un algoritmo especialmente utilizado en procesamiento de imágenes.

2.3.6. Agrupamiento espectral (*spectral clustering*)

El **espectro** de una matriz es el conjunto de sus **valores propios**. La técnica de análisis de componentes principales descrita en el subapartado 3.1.2 está estrechamente relacionada con el agrupamiento espectral, pues determina los ejes en los que los datos ofrecen mayor variabilidad tomando los vectores asociados a valores propios mayores; en la técnica que aquí nos ocupa, por el contrario, se buscan los valores propios menores, pues indican escasa variabilidad y por tanto pertenencia a un mismo grupo.

Ved también

Los valores y vectores propios de una matriz y su obtención con Python están explicados en el subapartado 3.1.1.

Si bien el fundamento matemático detallado queda fuera del ámbito de estos materiales, el algoritmo de agrupamiento espectral consta de los siguientes pasos:

Lectura complementaria

U. von Luxburg (2007). *A Tutorial on Spectral Clustering*. Statistics and Computing (vol. 4, núm. 17).

1) Calcular la matriz de distancias W de los datos que se desea agrupar.

2) Calcular la **laplaciana** de la matriz de distancias mediante la fórmula $L_{rw} = D - W$, donde D es una matriz en la que los elementos de la diagonal son la suma de cada fila de W.

Laplaciana no normalizada

La laplaciana calculada en el algoritmo de agrupamiento espectral se trata de la laplaciana no normalizada. Existen otras dos laplacianas aplicables en agrupamiento espectral, la simétrica y la de recorrido aleatorio; en muchos casos su comportamiento es equivalente.

3) Calcular los valores y vectores propios de L_{rw}.

4) Ordenar los valores propios de menor a mayor; reordenar las columnas de la matriz de vectores propios según ese mismo orden.

5) El número de grupos k sugerido por el algoritmo es el número de valores propios muy pequeños (menores que 10^{-10} si los grupos están realmente separados).

6) Aplicar un algoritmo de *clustering* convencional (k-medios por ejemplo) a las k primeras columnas de la matriz de vectores propios, indicando que se desea obtener k grupos.

El código 2.12 muestra cómo ejecutar el agrupamiento espectral en la versión de sklearn.

Sklearn

Sklearn (scikit-learn, http://scikit-learn.org/) es un conjunto de herramientas de análisis de datos y aprendizaje automático para Python que reúne gran cantidad de métodos y está diseñado para ser compatible con SciPy y NumPy.

Código 2.12: agrupamiento espectral

```
1 from sklearn.cluster import SpectralClustering
2
3 # Configurar el algoritmo para que calcule 4 grupos y aplicarlo
4 sc = SpectralClustering(n_clusters = 4)
5 sc.fit(datos)
6
7 # Ver las etiquetas de grupo de los 100 primeros puntos
8 print(sc.labels_[:100])
9 # [0 0 0 ..., 1 1 0]
```

El agrupamiento espectral es un método de agrupamiento muy potente, pues es capaz de agrupar datos teniendo en cuenta las diferencias de densidad, datos convexos (por ejemplo un círculo rodeando a otro), etc. Además indica el número de grupos que deben formarse.

Habitualmente se aplica un núcleo gaussiano* al calcular la matriz de distancias para mejorar su rendimiento.

***Véase el subapartado 4.5.2 en adelante.**

2.3.7. Recomendadores basados en modelos

La aplicación de algoritmos de agrupamiento para la construcción de recomendadores se basa, por lo general, en utilizar los grupos obtenidos para decidir qué recomendar a un usuario, en lugar de utilizar todos los datos disponibles. Esto agiliza enormemente el proceso de decisión y permite disponer de varias "vistas" de los datos: grupos de usuarios, de productos, etc. De esta manera, en cada circunstancia se puede encontrar una recomendación adecuada de forma rápida.

3. Extracción y selección de atributos

Supongamos que deseamos analizar un conjunto de datos para resolver un problema determinado. Imaginemos por ejemplo una estación meteorológica que cada minuto realice una medida de la temperatura y de la humedad relativa. Al cabo de una hora habremos obtenido un conjunto de datos formado por $m = 60$ observaciones de $n = 2$ variables, que representaremos con la matriz de tamaño 60×2. Si representamos las medida *i*–ésima de la temperatura con la variable $a_{i,1}$ y la de humedad relativa con $a_{i,2}$, la matriz de datos vendrá dada por

$$\begin{pmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \\ \vdots & \vdots \\ a_{60,1} & a_{60,2} \end{pmatrix} \tag{6}$$

Antes de proceder al análisis de los datos, debemos encontrar una forma adecuada de representarlos. A lo largo del apartado, representaremos los conjuntos de datos mediante una matriz multidimensional A de tamaño $m \times n$, en la que las n columnas representan las variables observadas y las m filas corresponden a las observaciones realizadas:

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,n} \\ a_{2,1} & a_{2,2} & \dots & a_{2,n} \\ & \vdots & & \\ a_{m,1} & a_{m,2} & \dots & a_{m,n} \end{pmatrix} \tag{7}$$

Una buena representación de datos resulta esencial porque por lo general los datos son redundantes o contienen información espuria. Antes de proceder a su análisis, es recomendable obtener un conjunto reducido de variables que permita describir los datos de forma simplificada. En el ejemplo anterior, si la temperatura y la humedad mantienen una relación determinada, los datos podrían simplificarse mediante una única variable que sea función de la temperatura y la humedad.

La simplificación de datos también se conoce como *reducción de dimensionalidad*, y pretende eliminar aspectos supérfluos como correlaciones entre variables (redundancias) o fluctuaciones estocásticas (ruido). Estos aspectos son agrupados en lo que comúnmente llamamos *ruido*, es decir, todos aquellos factores que no representen información relevante del proceso que vayamos a estudiar.

En el caso anterior, el ruido provendrá de las fluctuaciones de temperatura y humedad debidas a los flujos micrometeorológicos, o del ruido electrónico introducido por los equipos electrónicos de medida.

Por contra, denominaremos *señal* a aquellos factores que contribuyen de forma relevante a los datos, de forma que en general se cumplirá la identidad siguiente:

$$\text{datos} = \text{señal} + \text{ruido} \tag{8}$$

Las fuentes más habituales de ruido se asocian a los procesos de obtención y transmisión de los datos. Cuando los datos son obtenidos por algún tipo de sistema instrumental, se introducirán ciertas fluctuaciones inherentes al sistema físico de medida o a las etapas de amplificación y acondicionamiento. Al ruido en los sistemas de medida debe añadírsele el que se produce durante la transmisión de los datos a través de líneas de comunicación, típicamente debidos a interferencias electromagnéticas o a la pérdida o corrupción de datos en canales analógicos o digitales. Finalmente, también habrá que considerar cualquier distorsión de los datos debida a otros factores como redundancias, errores o a la incorporación de información que no resulte relevante. Como resulta evidente, uno de los problemas fundamentales del tratamiento de datos es separar señal y ruido. Pero ese no es el único aspecto que consideraremos en este subapartado. En ocasiones, el interés no será la eliminación del ruido sino la identificación de los diferentes aspectos que contribuyen a los datos. En estos casos será necesario agrupar algunas de las características de los datos para disponer de una representación de los mismos en función de k componentes elementales más simples que nos facilitan su interpretación

$$\text{datos} = \text{componente}_1 + \text{componente}_2 + \cdots + \text{componente}_k. \tag{9}$$

Las componentes se suelen ordenar siguiendo algún criterio objetivo que determine su importancia, lo que nos permite una interpretación simplificada de los datos mediante reconstrucciones parciales obtenidas a partir de algunas de las componentes. En general, asumiremos que los datos se obtienen mediante

la realización de medidas experimentales de un sistema determinado, de forma que la estructura de los datos necesariamente refleja las características de dicho sistema. Por ejemplo, al analizar los datos de la estación meteorológica anterior, podremos separar los datos en dos factores, uno correspondiente a los efectos diurnos y otro a los nocturnos.

El objetivo entonces será separar los datos en componentes que den cuenta de alguna de las características principales del sistema subyacente. Consideremos, por ejemplo, un conjunto de datos que consista en una secuencia de imágenes por satélite de una determinada zona geográfica. Resulta lógico pensar que la presencia de diferentes características geográficas en la zona (regiones montañosas, campos de cultivo, ríos, etc.) se reflejarán de alguna forma en el conjunto de datos a analizar. Las técnicas que estudiaremos en este subapartado nos van a permitir identificar las diferentes subzonas geográficas a partir de un análisis de la estructura de los datos. El procedimiento consistirá en definir un subconjunto de características de los datos (por ejemplo, el tamaño en píxeles de zonas de color verde en las imágenes) para poder identificar las distintas subzonas que se reflejan en los datos, y por lo tanto para descomponer los mismos en componentes diferenciadas.

El nombre genérico de este conjunto de las técnicas descritas en el subapartado 3.1 es *métodos de factorización matricial* porque descomponen en factores una matriz de datos. Una de las formas más eficientes de separar los datos consiste en identificar las componentes que mejor describen su variabilidad. El objetivo será encontrar un sistema de variables que describan los ejes en los que los datos presentan una mayor variabilidad. Esta es la idea básica de la técnica de descomposición en componentes principales (PCA), que describiremos más adelante, en el subapartado 3.1.2. Otra de las formas consiste en descomponer los datos en sus fuentes independientes, es decir, en aquellos mecanismos de naturaleza independiente del sistema subyacente a los datos. Eso es lo que aprenderemos en el subapartado 3.1.3 cuando estudiemos la técnica de descomposición en componentes independientes (ICA). La tercera técnica que estudiaremos en el subapartado 3.1.4 es conocida como factorización de matrices no-negativas (NMF) y permite una descomposición de los datos de forma que cada componente contribuye de forma acumulativa a la matriz global.

En el subapartado 3.3.1 también aprenderemos cómo utilizar otras técnicas para la visualización de datos multidimensionales o para identificar qué variables permiten separar los datos en grupos diferentes en el subapartado 3.2.1.

3.1. Técnicas de factorización matricial

Las técnicas de extracción y selección de atributos nos permiten describir los datos en función de unas nuevas variables que pongan de manifiesto algu-

nos de los aspectos de los datos en los que estemos interesados, omitiendo o atenuando aquellos factores que no son relevantes.

Las técnicas que se describen en este apartado (SVD, PCA, ICA y NMF) son conocidas como *técnicas de factorización matricial*, puesto que todas ellas descomponen una matriz de datos como el producto de matrices más simples. En cualquier caso, la factorización de una matriz no es única, y cada técnica pone de manifiesto aspectos diferentes de la información contenida en los datos originales.

3.1.1. Descomposición en valores singulares (SVD)

La descomposición en valores singulares es una herramienta de álgebra lineal que tiene innumerables aplicaciones en campos como el cálculo numérico, el tratamiento de señales o la estadística. El motivo por el que resulta tan útil es que SVD permite descomponer una matriz en una forma en la que nos resultará muy sencillo calcular sus valores y vectores propios (diagonalización de matrices). En sistemas de inteligencia artificial, SVD permite descomponer una matriz de datos expresándola como la suma ponderada de sus vectores propios. Una de las formas más simples de simplificar un conjunto de datos es descomponerlos utilizando SVD y reconstruirlos a partir de los vectores propios con mayor valor propio. Los vectores propios con menor valor propio suelen corresponder a aspectos de detalle de los datos que pueden no ser relevantes en una primera exploración. Además, SVD también se utiliza en la descomposición en componentes principales que describiremos en el subapartado 3.1.2.

Diagonalizar una matriz

Diagonalizar una matriz consiste en obtener sus valores y vectores propios. Los vectores propios definen un nuevo sistema de coordenadas en el que la matriz de datos es diagonal.

En el caso particular de una matriz cuadrada de tamaño $n \times n$, la *diagonalización* consiste en encontrar el conjunto de valores y vectores propios $\{\lambda_i, v_i\}, i = 1 \dots n$ que cumplan la identidad

$$A \cdot v_i = \lambda_i v_i, \tag{10}$$

para luego expresar la matriz A en la forma

$$A = V^{-1} \cdot \Lambda \cdot V, \tag{11}$$

donde Λ es una matriz diagonal $n \times n$ cuyos componentes son los valores propios λ_i y V es una matriz $n \times n$ cuyas n columnas son los vectores propios v_i (cada uno de tamaño $1 \times n$).

El código 3.1 indica cómo obtener los valores y vectores propios de una matriz en Python. En la primera línea se importa la librería NumPy, que incluye gran parte de las funciones que necesitaremos en este apartado. La línea 3 introduce una matriz A de tamaño 3×3, y en la línea 10 se obtienen los valores y vectores propios de A utilizando la rutina *svd* del paquete de rutinas de álgebra lineal *linalg* incluido en la librería NumPy. En las instrucciones de las líneas 20 y 23 se comprueba que la ecuación 10 se cumple para el primero de los valores propios v_1 (las líneas 20 y 23 corresponden respectivamente a los términos izquierdo y derecho de la ecuación 10).

Código 3.1: ejemplo diagonalización en Python

```
1  >>> from numpy import *
2
3  >>> A = array([[1, 2, 3],[4, 5, 6],[7, 8, 9]])
4
5  >>> A
6  array([[1, 2, 3],
7         [4, 5, 6],
8         [7, 8, 9]])
9
10 >>> val,vec = linalg.eig(A)
11
12 >>> val
13 array([  1.61168440e+01,  -1.11684397e+00,  -1.30367773e-15])
14
15 >>> vec
16 array([[-0.23197069, -0.78583024,  0.40824829],
17        [-0.52532209, -0.08675134, -0.81649658],
18        [-0.8186735 ,  0.61232756,  0.40824829]])
19
20 >>> dot(A,vec[:,0])
21 array([ -3.73863537,  -8.46653421, -13.19443305])
22
23 >>> val[0]*vec[:,0]
24 array([ -3.73863537,  -8.46653421, -13.19443305])
```

Como las matrices de datos no suelen ser cuadradas, SVD generaliza la diagonalización en matrices con un tamaño arbitrario $m\times n$. El resultado consiste en una descomposición factorial de la matriz A de tamaño $m \times n$ como producto de tres matrices U, S y V

$$A = U \cdot S \cdot V^T, \tag{12}$$

donde S es una matriz diagonal de tamaño $m \times n$ con componentes no negativos y U,V son matrices unitarias de tamaño $m \times m$ y $n \times n$ respectivamente. A los componentes diagonales no nulos de la matriz S se les conoce como *valores singulares*, mientras que las m columnas de la matriz U y las n columnas de la matriz V se denominan *vectores singulares* por la izquierda y por la derecha, respectivamente. El número máximo de valores singulares diferentes de la matriz A está limitado por el rango máximo de la matriz A, $r = \min\{m,n\}$.

El código 3.2 describe cómo se realiza una descomposición SVD en Python utilizando las herramientas de la librería NumPy. El código empieza definien-

do una matriz A de tamaño 4×2, y en la línea 8 se obtiene su descomposición SVD. La última instrucción (línea 28) comprueba que los términos izquierdo y derecho de la ecuación 12 sean iguales utilizando la instrucción *allclose* de Python.

Código 3.2: ejemplo SVD en Python

```
1  >>> from numpy import *
2  >>> A = array([[1,3,5,7],[2,4,6,8]]).T
3  >>> A
4  array([[1, 2],
5         [3, 4],
6         [5, 6],
7         [7, 8]])
8  >>> U,S,Vt = linalg.svd(A,full_matrices=True)
9  >>> U
10 array([[-0.15248323, -0.82264747, -0.39450102, -0.37995913],
11        [-0.34991837, -0.42137529,  0.24279655,  0.80065588],
12        [-0.54735351, -0.0201031 ,  0.69790998, -0.46143436],
13        [-0.74478865,  0.38116908, -0.5462055 ,  0.04073761]])
14 >>> S
15 array([ 14.2690955 ,   0.62682823])
16 >>> Vt
17 array([[-0.64142303, -0.7671874 ],
18        [ 0.7671874 , -0.64142303]])
19 >>> S1 = zeros((4,2))
20 >>> S1[:2,:2] = diag(S)
21 >>> S1
22 array([[ 14.2690955 ,   0.        ],
23        [  0.        ,   0.62682823],
24        [  0.        ,   0.        ],
25        [  0.        ,   0.        ]])
26 >>> S1.shape
27 (4, 2)
28 >>> allclose(A, dot(U, dot(S1, Vt)))
29 True
```

SVD también puede interpretarse como una descomposición en la que la matriz A se expresa como la suma de $r = \min\{m,n\}$ matrices, conocidas como *componentes SVD*

$$A = U \cdot S \cdot V^T = S(1,1)\ u_1 \cdot v_1^T + S(2,2)\ u_2 \cdot v_2^T + \cdots + S(r,r)\ u_r \cdot v_r^T, \qquad (13)$$

donde $u_1, u_2, \ldots, u_m$ son los vectores columna de la matriz de vectores singulares por la izquierda $U = \{u_1, u_1, \ldots, u_m\}$, y $v_1, v_2, \ldots, v_n$ son los vectores columna de la matriz de vectores singulares por la derecha $V = \{v_1, v_1, \ldots, v_n\}$. En esta representación, cada una de las r matrices $u_j \cdot v_j^T$ tienen un peso que viene dado por el valor singular correspondiente $S(j,j)$. Así, los vectores singulares con mayor valor singular serán más representativos de la matriz A, mientras que los vectores singulares con un menor valor singular aportarán muy poco a la descomposición en componentes SVD de A. Este hecho es el que permite que SVD sea utilizada como técnica de representación de datos, pues resulta una herramienta útil para identificar qué vectores singulares son relevantes para describir la matriz A y cuáles pueden considerarse como residuales.

Aplicando este procedimiento a una matriz de datos obtendremos una representación de la información en función de unos pocos vectores singulares, y

por lo tanto dispondremos de una representación de los datos en un espacio de dimensionalidad reducida. Representar los datos de forma fiable en un espacio de dimensionalidad reducida tiene evidentes ventajas para realizar un análisis exploratorio de la información, y puede ser utilizado como técnica de reducción del ruido o de compresión de datos. El código 3.3 implementa un ejemplo de esta descomposición en componentes SVD, comprobando que la matriz A puede reconstruirse como la suma de sus componentes (la línea 7 corresponde a la parte derecha de la ecuación 13).

Observación

La instrucción z.reshape(-1,1) convierte el vector z en un vector columna. Para convertir un vector z en un vector fila utilizaríamos la instrucción z.reshape(1,-1).

Código 3.3: descomposición de una matriz en sus componentes SVD

```
1  >>> from numpy import *
2
3  >>> A = array([[1, 2],[3, 4]])
4
5  >>> U,S,Vh = linalg.svd(A)
6
7  >>> dot(U, dot(diag(S), Vh))
8
9  array([[ 1.,  2.],
10        [ 3.,  4.]])
```

Otra propiedad interesante de la descomposición SVD de la matriz A es que permite calcular directamente los vectores y valores propios de la matriz $A^T \cdot A$. En efecto, tenemos que

$$A = U \cdot S \cdot V^T, \tag{14}$$

$$A^T = V \cdot S \cdot U^T, \tag{15}$$

y aplicando la unicidad de U, $U^T \cdot U = 1$, tenemos que

$$A^T \cdot A = V \cdot S \cdot U^T \cdot U \cdot S \cdot V^T = V \cdot S^2 \cdot V^T, \tag{16}$$

lo que demuestra que los vectores propios de $A^T \cdot A$ vienen dados (con la posible excepción del signo) por el vector V^T obtenido en la descomposición SVD de A, y sus vectores propios por S^2. Por lo tanto, la descomposición SVD de una matriz de datos A permite conocer los valores y vectores propios de la matriz de correlación $A^T \cdot A$. Esta propiedad se describe con un ejemplo en el código 3.4, en el que se comparan los resultados de aplicar SVD a una matriz A con los obtenidos de diagonalizar la matriz $A^T \cdot A$.

Código 3.4: SVD permite calcular los valores y vectores propios de la matriz $A^T \cdot A$

```
1  >>> from numpy import *
2  >>> A = array([[1, 2, 3],[4, 5, 6],[7, 8, 9]])
3
4  >>> cov_A = dot(A.T,A)
5  >>> cov_A
```

```
array([[ 66,  78,  90],
       [ 78,  93, 108],
       [ 90, 108, 126]])

>>> val,vec = linalg.eig(cov_A)

>>> U,S,Vt = linalg.svd(A)

>>> val
array([  2.83858587e+02,   1.14141342e+00,   7.28303082e-15])

>>> S*S.T
array([  2.83858587e+02,   1.14141342e+00,   2.16916414e-32])

>>> vec
array([[-0.47967118, -0.77669099,  0.40824829],
       [-0.57236779, -0.07568647, -0.81649658],
       [-0.66506441,  0.62531805,  0.40824829]])

>>> Vt.T
array([[-0.47967118, -0.77669099,  0.40824829],
       [-0.57236779, -0.07568647, -0.81649658],
       [-0.66506441,  0.62531805,  0.40824829]])
```

3.1.2. Análisis de componentes principales (PCA)

Ejemplo de aplicación

A modo de ejemplo introductorio, supongamos que disponemos de un conjunto de datos con información sobre los jugadores de fútbol de primera división. Teniendo en cuenta que la liga española dispone de 20 equipos y que cada uno consta de 25 jugadores con licencia federativa, dispondremos de un total de $m = 500$ observaciones. Para cada jugador disponemos de tres variables con la media de pases, goles y balones robados por partido respectivamente ($n = 3$ variables, la matriz de datos tendrá un tamaño $m = 500 \times 3$). Los datos, representados en el sistema de ejes tridimensional de las variables, adoptan un aspecto similar al de una nube de puntos en el espacio.

PCA es una técnica que permite determinar los ejes principales de la distribución de la nube de puntos. Encontrar los ejes principales de la nube de puntos equivale a encontrar un nuevo conjunto de variables en los que los datos presentan una dispersión máxima, es decir, aquellos ejes en los que la variabilidad de los datos es mayor. En el ejemplo de los datos de futbolistas, estos ejes corresponden a combinaciones de las tres variables en las que los futbolistas tienen una mayor variabilidad. Podría pasar, por ejemplo, que la mayoría de futbolistas realicen un número promedio de pases por partido similar, y que las discrepancias entre jugadores se deban a una combinación entre el número promedio de balones que roban y de goles que marcan. Esta combinación de variables describirá un nuevo eje de coordenadas (el primer componente principal), en el que se producirá una mayor dispersión de los datos en el espacio original. Una proyección de los datos en un subconjunto de los ejes principales permite una simplificación de los datos a partir de una representación en un espacio de dimensionalidad reducida. Más adelan-

te veremos un ejemplo genérico en el que aplicaremos la técnica PCA a un conjunto de datos tridimensional. La técnica PCA se basa en la diagonalización de la matriz de covarianza de los datos, y sus características principales se describen a continuación.

Descripción de la técnica

Si A representa una matriz de datos de tamaño $m \times n$ con m observaciones de n variables, la matriz $A^T \cdot A$ es conocida como *matriz de autocorrelación* y expresa las relaciones estadísticas existentes entre las n variables. Básicamente, la técnica PCA consiste en diagonalizar la matriz de autocorrelación $A^T \cdot A$ o la matriz de covarianza $(A - \bar{A})^T \cdot (A - \bar{A})$ de los datos A.

Como las matrices de autocorrelación o autocovarianza indican correlación estadística entre variables, sus vectores y valores propios nos permitirán obtener unas nuevas variables entre las que la correlación estadística sea mínima. Los vectores propios darán lugar a las *componentes principales*, definiendo unos nuevos ejes de coordenadas orientados en las direcciones en las que los datos presenten una mayor variabilidad. Los vectores propios con mayor valor propio corresponderán a las direcciones en las que los datos presentan una mayor dispersión, y aquellos vectores propios con un autovalor pequeño indicarán una dirección en la que los datos apenas varían su valor.

Como hemos visto en el subapartado 3.1.1, la descomposición SVD de una matriz A permite calcular los vectores y valores propios de la matriz $A^T \cdot A$, por lo que SVD nos será de gran utilidad para descomponer A en sus componentes principales.

La relación entre SVD y PCA se pone de manifiesto al escribir la descomposición SVD de la matriz A en la forma

$$A = R \cdot V^T \tag{17}$$

donde $R = U \cdot S$ es conocida como *matriz de resultados* (en inglés, *scores matrix*), y a la matriz de vectores singulares por la derecha V^T se le conoce como *matriz de cargas* (en inglés, *loadings matrix*).

La ecuación 17 expresa la descomposición PCA de la matriz A como una factorización en las matrices R y V^T.

Los pasos a seguir para realizar una descomposición PCA de una matriz de datos en Python se indican en el ejemplo descrito en el código 3.5.

Código 3.5: ejemplo de PCA en Python

```
# -*- coding: utf-8 -*-
from numpy import *
import pylab
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plot
set_printoptions(precision = 3)

# Datos: distribucion normal multivariada en 3d
mean = [1,5,10]
cov = [[-1,1,2],[-2,3,1],[4,0,3]]
d = random.multivariate_normal(mean,cov,1000)

# representacion grafica de los datos:
fig1 = plot.figure()
sp = fig1.gca(projection = '3d')
sp.scatter(d[:,0],d[:,1],d[:,2])
plot.show()

# ANALISIS PCA:
# Paso 1: Calcular la matriz de covarianza de los datos (N x N):
d1 = d - d.mean(0)
matcov = dot(d1.transpose(),d1)

# Paso 2: Obtener los valores y vectores propios de la matrix de
    covarianza:
valp1,vecp1 = linalg.eig(matcov)

# Paso 3: Decidir que vectores son los relevantes representando
# los valores propios en orden decreciente (scree plot):
ind_creciente = argsort(valp1) # orden creciente
ind_decre = ind_creciente[::-1] # orden decreciente
val_decre = valp1[ind_decre] # valores propios en orden decreciente
vec_decre = vecp1[:,ind_decre] # ordenar tambien vectores propios
pylab.plot(range(0,len(val_decre)), val_decre,'o-')
pylab.show()


# Proyectar datos a la nueva base definida por los
# todos los vectores propios (espacio PCA 3D)
d_PCA = zeros((d.shape[0],d.shape[1]))
for i in range(d.shape[0]):
    for j in range(d.shape[1]):
        d_PCA[i,j] = dot(d1[i,:],vecp1[:,j])

# recuperar datos originales invirtiendo la proyeccion (reconstruccion)
    :
orig_means = d.mean(0)
d_recon = zeros((d.shape[0],d.shape[1]))
for i in range(d.shape[0]):
    d_recon[i] = orig_means
    for j in range(d.shape[1]):
         d_recon[i] += d_PCA[i,j]*vecp1[:,j]

# comprobar que se recuperan los datos originales:
allclose(d,d_recon)


# Proyectar datos a la nueva base definida por los dos
# vectores propios con mayor valor propio (espacio PCA 2D)
d_PCA2 = zeros((d.shape[0],2))
for i in range(d.shape[0]):
    for j in range(2):
        d_PCA2[i,j] = dot(d1[i,:],vec_decre[:,j])

# reconstruir datos invirtiendo la proyeccion PCA 2D:
```

Nuevo sistema de coordenadas

PCA permite caracterizar un conjunto de datos mediante un nuevo sistema de coordenadas en las direcciones en las que los datos presentan una mayor variabilidad.

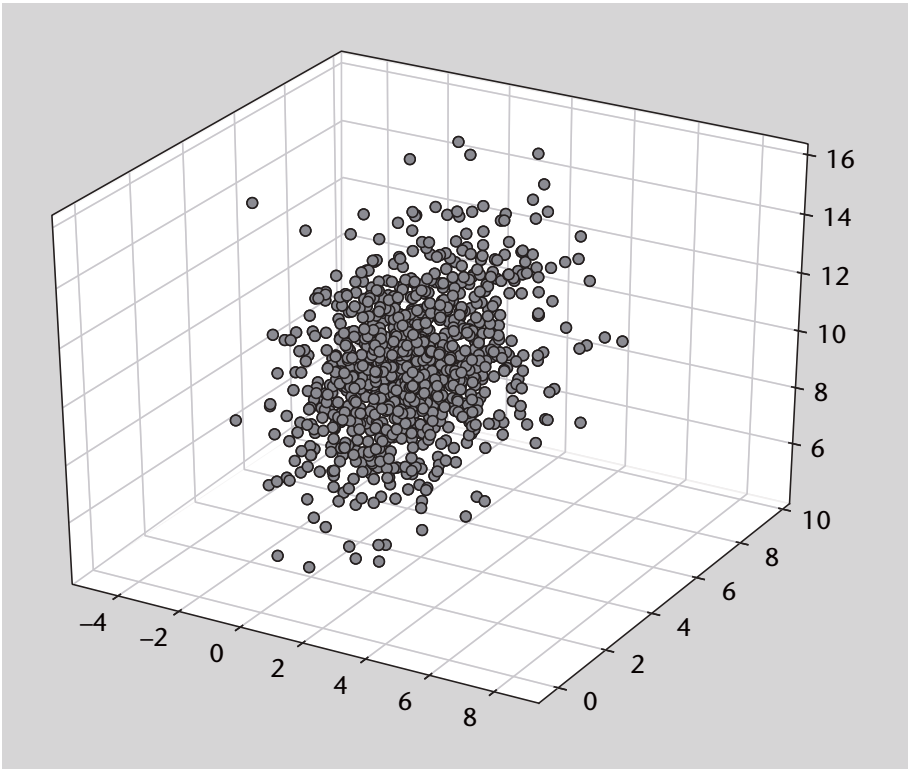
```
64 d_recon2 = zeros((d.shape[0],d.shape[1]))
65 for i in range(d.shape[0]):
66     d_recon2[i] = orig_means
67     for j in range(2):
68         d_recon2[i] += d_PCA2[i,j]*vec_decre[:,j]
69
70 # En general dara falso por el error introducido al descartar
71 # una componente
72 allclose(d,d_recon2)
73
74 # representacion grafica de los datos:
75 fig2 = plot.figure()
76 sp2 = fig2.gca(projection = '3d')
77 sp2.scatter(d_recon2[:,0],d_recon2[:,1],d_recon2[:,2],c='r',marker='x')
78 plot.show()
```

En la línea 10 se genera una matriz 1000×3 de datos distribuidos aleatoriamente según una densidad de probabilidad normal multivariada con media (1,5,10) y matriz de covarianza

$$\begin{pmatrix} -1 & 1 & 2 \\ -2 & 3 & 1 \\ 4 & 0 & 3 \end{pmatrix}. \tag{18}$$

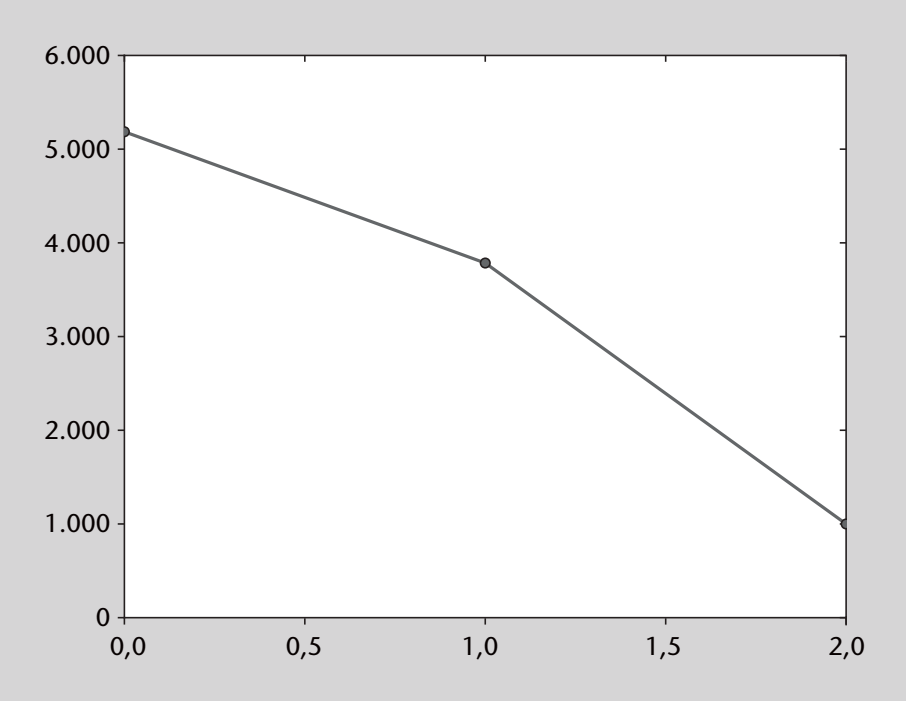
Esta matriz de datos es similar a la descrita en el ejemplo de los datos de futbolistas al que hacíamos referencia en la introducción de este subapartado. Los datos se representan en la figura 7 mediante una gráfica tridimensional, en la que se aprecia la distribución de datos en forma de nube de puntos.

Figura 7. Datos del ejemplo PCA descrito en el código 3.5



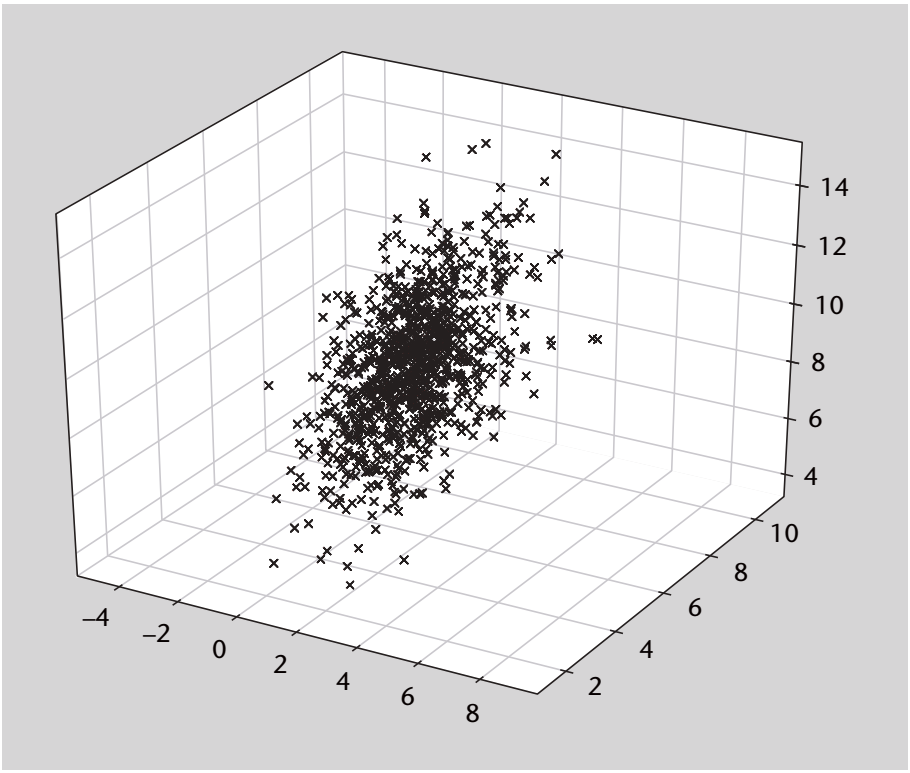
El primer paso para realizar PCA es obtener la matriz de autocovarianza de los datos (línea 21), que se obtiene tras substraer la media a cada columna de la matriz *d* (línea 20). El segundo paso es obtener los valores y vectores propios de la matriz de autocovarianza (línea 24). Para decidir qué vectores propios son más relevantes, lo habitual es representar gráficamente los valores propios en orden decreciente en lo que se conoce como un *scree plot* (diagrama de pedregal, por la forma abrupta en la que decrecen los autovalores que recuerda a la ladera de una montaña). Los valores y vectores propios son ordenados siguiendo un orden decreciente del valor propio (líneas 28-31). Este diagrama aparece representado en la figura 8, en la que se obseva que uno de los autovalores es significativamente menor que los otros dos. Cada uno de los vectores propios es un vector que apunta en tres direcciones ortogonales en las que los datos presentan una mayor variabilidad (espacio PCA).

Figura 8. Scree plot de los valores propios de la matriz de autocovarianza del ejemplo del código 3.5



En las líneas de la 37 a 40 se proyectan los datos originales al espacio PCA realizando el producto escalar entre los datos y los vectores propios (línea 40). Los datos originales se pueden recuperar invirtiendo la proyección (líneas 43), lo que se comprueba en la línea 49 del código. A continuación, se realiza una proyección de los datos a un espacio PCA que sólo tiene en cuenta los dos vectores propios con mayor valor propio asociado (líneas 53-56). Al considerar los dos componentes más relevantes que la reconstrucción de los datos a partir del espacio PCA de dimensionalidad reducida se asemeja a los datos originales (figura 9).

Figura 9. Reconstrucción de los datos del ejemplo 3.5 utilizando los dos vectores propios con mayor valor propio



Implementación de PCA con las librerías sklearn

Las librerías sklearn permiten realizar un análisis PCA de una forma más directa, tal y como se describe en el código 3.6. Es importante comentar que antes de realizar la descomposición PCA (línea 30) se procede a centrar y escalar los datos en lo que se conoce como una estandarización de los mismos (línea 19). La importancia de este escalado de los datos es evidente: como hemos comentado anteriormente, lo que se pretende con PCA es obtener una representación de los datos en un espacio de dimensionalidad reducida en el que las nuevas variables están lo más descorrelacionadas posible.

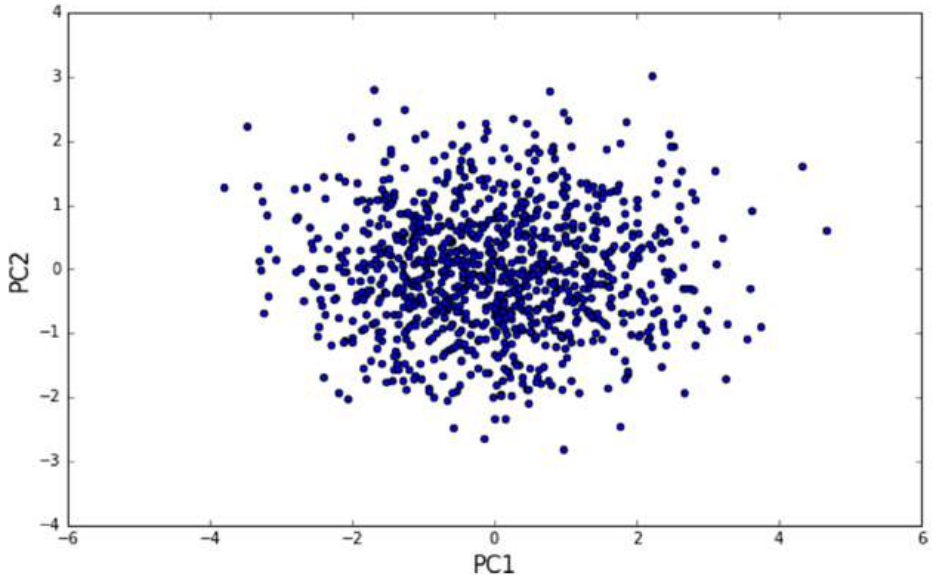
Teniendo en cuenta que las dos primeras componentes PCA explican más del 90 % de la varianza de los datos, es razonable considerar una reducción de dimensionalidad de los datos a un espacio PCA 2D. De las últimas líneas del código se obtiene la proyección de los datos originales en un espacio PCA bidimensional (figura 10). Los datos aparecen claramente descorrelacionados al ser representados en el nuevo espacio de dimensionalidad reducida.

Código 3.6: ejemplo de PCA en Python utilizando las librerías sklearn

```
1 # -*- coding: utf-8 -*-
2 from numpy import *
3 import matplotlib.pyplot as plot
4 set_printoptions(precision = 3)
5
6 # Datos: distribucion normal multivariada en 3d
```

```
mean = [1,5,10]
cov = [[2,-1,0],[-1,2,-1],[0,-1,2]]
d = random.multivariate_normal(mean,cov,1000)

# Representacion grafica de los datos:
fig1 = plot.figure(1,figsize=(10, 6))
sp = fig1.gca(projection = '3d')
sp.scatter(d[:,0],d[:,1],d[:,2])
plot.show()

# ANALISIS PCA:
# Estandarizacion de los datos ( d1 = (d - d.mean(0))/d.std(0) ):
from sklearn import preprocessing
d1 = preprocessing.scale(d)

# Representacion grafica de los datos estandarizados:
fig0 = plot.figure(1,figsize=(10, 6))
sp = fig0.gca(projection = '3d')
sp.scatter(d1[:,0],d1[:,1],d1[:,2])
plot.show()

import numpy as np
from sklearn.decomposition import PCA
pca1 = PCA(n_components=3)
pca1.fit(d1)

# Varianza explicada acumulada por cada componente
print(cumsum(pca.explained_variance_ratio_))

# Representar los datos en el espacio PCA:
X = pca.transform(d1)

fig2 = plot.figure(1, figsize=(10, 6))
ax = fig2.gca(projection = '3d')
ax.scatter(X[:, 0], X[:, 1], X[:, 2])
```

Figura 10. Proyección de los datos del ejemplo 3.6 en los dos primeros componentes principales PCA



La importancia de escalar las variables antes de realizar un análisis PCA

Como hemos dicho anteriormente, PCA es una técnica que se basa en la contribución relativa de cada variable a la variabilidad total de los datos. En resumen, PCA permite representar los datos en una nueva base ortogonal en la que las nuevos ejes son una combinación lineal de las variables originales. Dichos ejes son también conocidos como vectores propios o componentes principales. La utilidad de PCA proviene de que los ejes de la nueva base contribuyen a la variabilidad total de forma proporcional a su valor propio correspondiente. Eso nos permite prescindir de aquellos ejes o componentes principales con una contribución a la variabilidad relativamente baja. O lo que es lo mismo, representar los datos en un espacio de dimensionalidad reducida utilizando únicamente los ejes que explican un cierto porcentaje de la variabilidad de los datos. La nueva base puede ser considerada como un nuevo juego de atributos entre los que la correlación estadística es mínima. PCA es útil en situaciones en las que las variables originales del problema presenten correlaciones entre ellas, lo cual es un fenómeno que se observa de forma habitual en muchas bases de datos.

Teniendo en cuenta lo comentado anteriormente, antes de realizar un análisis de componentes principales es importante garantizar que las variables presenten un rango de valores que sea comparable. En efecto, si dos variables toman valores en rangos muy diferentes, la variable con un rango mayor contribuirá más a la variabilidad total de los datos independientemente de si existe o no correlación entre ellas. Para que los resultados no dependan de factores de escala en las variables originales (por ejemplo, las unidades en las que están escritas), es importante realizar un escalado de los datos antes de aplicar PCA.

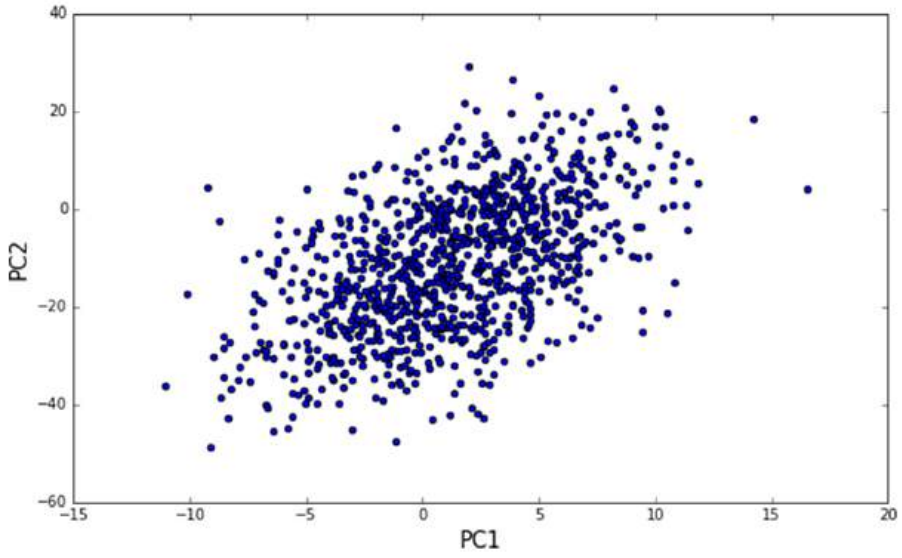
El ejemplo del código 3.7 muestra la disparidad en los resultados obtenidos al hacer un análisis PCA al realizar o no un escalado previo de los datos. La matriz de covarianza desde la que se generan las muestras gaussianas presenta ahora una desviación típica de las variables de 2, 20 y 200, con lo que conseguimos que la variabilidad en las variables originales sea claramente diferente. Como se puede observar, al aplicar PCA a los datos sin escalar, el primer componente principal explica más del 90 % de la variabilidad de los datos. En cambio, cuando escalamos los datos la variabilidad explicada por el primer componente se reduce a menos del 40 %, y los tres componentes contribuyen de forma equivalente. El resultado obtenido con los datos sin escalar debe ser considerado como un error de aplicación de la técnica y es debido a las grandes diferencias en variabilidad en las variables originales. El fenómeno se comprueba al proyectar los datos en los dos primeros componentes principales (figuras 11 y 12), donde podemos observar que cuando los datos no han

sido escalados los datos proyectados presentan claramente una mayor correlación que cuando los datos son escalados.

Código 3.7: importancia de escalar las variables antes de aplicar un análisis PCA

```
# -*- coding: utf-8 -*-
from numpy import *
import matplotlib.pyplot as plot
set_printoptions(precision = 3)

# Datos: distribucion normal multivariada en 3d
mean = [1,5,10]
cov = [[2,-1,0],[-1,20,-1],[0,-1,200]]
d = random.multivariate_normal(mean,cov,1000)

# ANALISIS PCA:
# Estandarizacion de los datos ( d1 = (d - d.mean(0))/d.std(0) ):
from sklearn import preprocessing
d1 = preprocessing.scale(d)

# representacion grafica de los datos:
fig1 = plot.figure(1,figsize=(10, 6))
sp = fig1.gca(projection = '3d')
sp.scatter(d[:,0],d[:,1],d[:,2])
plot.show()

import numpy as np
from sklearn.decomposition import PCA
pca1 = PCA(n_components=3)
pca1.fit(d)
X = pca1.transform(d)
print(pca1.explained_variance_ratio_)

pca2 = PCA(n_components=3)
pca2.fit(d1)
X = pca2.transform(d1)
print(pca2.explained_variance_ratio_)

# Reduccion de dimensionalidad:
# Proyeccion PCA 2D (datos sin escalar):
pca3 = PCA(n_components=2)
pca3.fit(d)
X = pca3.transform(d)

fig2 = plot.figure(1, figsize=(10, 6))
ax = fig2.gca()
ax.scatter(X[:, 0], X[:, 1])
ax.set_xlabel('PC1', fontsize=15)
ax.set_ylabel('PC2', fontsize=15)
plot.show()

# Proyeccion PCA 2D (datos escalados):
pca4 = PCA(n_components=2)
pca4.fit(d1)
X = pca4.transform(d1)

fig3 = plot.figure(1, figsize=(10, 6))
ax = fig3.gca()
ax.scatter(X[:, 0], X[:, 1])
ax.set_xlabel('PC1', fontsize=15)
ax.set_ylabel('PC2', fontsize=15)
plot.show()
```

Figura 11. Proyección 2D PCA con los datos sin escalar



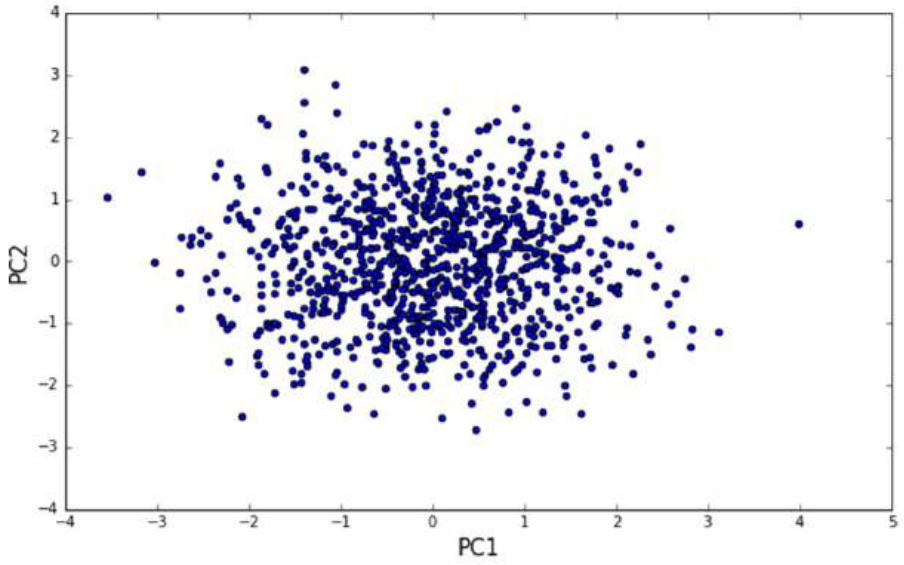
Ejemplo: PCA para imágenes

Una de las aplicaciones más interesantes de PCA es el análisis de imágenes. En este caso, PCA se aplica a una secuencia de imágenes con el objetivo de identificar las características principales que aparecen en las mismas. Las aplicaciones son numerosas, e incluyen aspectos como el reconocimiento de caras, el análisis de imágenes médicas o el procesado de información geográfica obtenida mediante técnicas de teledetección por satélite.

Matemáticamente, una imagen no es más que una matriz de $n \times m$ valores enteros. Cada uno de los componentes de la matriz representa un píxel de la imagen, y adopta valores enteros entre 0 y $2^k - 1$, donde k es el número de bits utilizados para representar cada píxel. En caso de que se utilicen 8 bits por píxel ($k = 8$), por ejemplo, los componentes de la matriz tomarán valores enteros en el rango [0 – 255]. Por lo tanto, una secuencia de imágenes no es más que un conjunto de NIM matrices cada una de ellas de tamaño $n \times m$. Para poder trabajar con PCA, los datos se organizan de forma que cada imagen sea almacenada en un único vector fila que contenga sus NPIX = $n \cdot m$ píxeles. De esta manera, la secuencia de imágenes se describe matemáticamente como una matriz de tamaño NIM × NPIX que adopta la forma

$$\begin{pmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,\mathrm{NPIX}} \\ a_{2,1} & a_{2,2} & \dots & a_{2,\mathrm{NPIX}} \\ a_{3,1} & a_{3,2} & \dots & a_{3,\mathrm{NPIX}} \\ & \dots & & \\ a_{\mathrm{NIM},1} & a_{\mathrm{NIM},2} & \dots & a_{\mathrm{NIM,NPIX}} \end{pmatrix}, \tag{19}$$

Figura 12. Proyección 2D PCA con los datos escalados



donde cada fila corresponde a una imagen de la secuencia y cada columna representa uno de los píxeles de la imagen.

Secuencias de Imágenes

Una secuencia de NIM imágenes de NPIX píxeles se puede representar como una matriz de datos con NPIX variables y NIM observaciones.

El problema principal con el que nos encontramos al aplicar PCA a una secuencia de imágenes es que normalmente el número de imágenes NIM es considerablemente menor que el número de píxeles NPIX de cada imagen. El problema radica en que para calcular PCA habría que diagonalizar la matriz de covarianza de tamaño NPIX $\times$ NPIX, lo que resulta tremendamente costoso incluso para imágenes de tamaño reducido (con imágenes de 128 $\times$ 128 píxeles tendríamos que diagonalizar una matriz de 16384 $\times$ 16384). Para solventar esta situación, se utiliza una interesante propiedad algebraica según la cual las matrices $A^T \cdot A$ (de tamaño NPIX $\times$ NPIX) y $A \cdot A^T$ (NIM $\times$ NIM) comparten los valores propios no nulos. En efecto, las condiciones de diagonalización de ambas matrices vienen dadas por

$$A^T \cdot A \cdot u_i = \lambda_i u_i \quad (20)$$

$$A \cdot A^T \cdot w_i = \lambda_i w_i. \quad (21)$$

Multiplicando la segunda ecuación por A^T por la izquierda tenemos

$$A^T \cdot A \cdot A^T \cdot w_i = A^T \lambda_i w_i, \quad (22)$$

e identificando 22 con la ecuación de 20 tenemos que los valores propios de $A^T \cdot A$ (u_i) y los de $A \cdot A^T$ (w_i) están relacionados de la forma

$$A^T \cdot w_i = u_i \quad (23)$$

por lo que podemos obtener los vectores propios de $A^T \cdot A$ diagonalizando la matriz $A \cdot A^T$ que, en el caso de imágenes, tiene un tamaño menor. Esta estrategia es la que se implementa en el código 3.8, en el que se aplica PCA a una secuencia de imágenes. El código supone que las imágenes tienen extensión *.jpg y que se encuentran en el mismo directorio en el que se ejecute el programa. Las líneas 13-23 construyen una lista Python con los nombres de los ficheros con extensión *.jpg encontrados en el directorio. Tras determinar el número de imágenes NIM y el tamaño de las mismas NPIX, se construye una matriz de datos de tamaño NIM $\times$ NPIX con las NIM imágenes desplegadas en forma de vectores fila de longitud NPIX (línea 33). A continuación se procede al centrado de los datos (líneas 41-43), se obtiene la matriz $M = A \cdot A^T$ (línea 45) y se calculan los vectores y valores propios de la matriz M (línea 46). Los vectores propios de $A^T \cdot A$ se obtienen utilizando la ecuación 23 (línea 47). Las líneas 49 ordenan los vectores propios en orden decreciente de sus valores propios, y la siguiente determina los valores propios de la matriz de datos A.

Código 3.8: aplicación de PCA a una secuencia de imágenes

```
1  import os
2  from PIL import Image
3  import numpy
4  import pylab
5  from pca_im import *
6  import matplotlib.pyplot as plt
7
8  ################################
9  # CONSTRUIR MATRIZ DE DATOS
10 ################################
11
12 # Obtener directorio actual en el que estan las imagenes
13 path= os.getcwd()
14 dirList=os.listdir(path)
15
16 # Construir una lista con los ficheros con extension '.jpg'
17 imlist = []
18 i = 0
19 for fname in dirList:
20     filename, filext = os.path.splitext(fname)
21     if filext == '.jpg':
22         imlist.append(fname)
23         i = i +1
24
25 # Abrir la primera imagen y obtener su tamaño en pixeles
26 im = numpy.array(Image.open(imlist[0]))
27 m,n = im.shape[0:2]
28
29 NPIX = n*m    # Numero de pixeles de cada imagen:
30 NIM = len(imlist) # Numero de imagenes a analizar:
31
32 # Crear una matriz de tamaño NIM x NPIX:
33 A = numpy.array([numpy.array(Image.open(imlist[i])).flatten()
34                  for i in range(NIM)],'f')
35
36 ################################
37 # ANALISIS PCA
38 ################################
39
40 # Centrado de los datos restando la media:
41 im_media = A.mean(axis=0)
42 for i in range(NIM):
43       A[i] -= im_media
44
45 M = dot(A,A.T) # matriz AA' (NIM x NIM)
46 lam,vec = linalg.eigh(M) # obtener los NIM vectores y valores propios
```

```
    de M = AA'
aux = dot(A.T,vec).T # aplicar w = A'v para obtener los vectores
    propios de A'A
V = aux[::-1] # ordenar vectores propios
S = sqrt(lam)[::-1] # ordenar valores propios de A

# Recomponer la imagen media:
im_media = im_media.reshape(m,n)

# Representar los autovalores PCA
pylab.plot(S[0:10],'o-')

# Obtener los dos primeros modos PCA y representarlos:
modo1 = V[0].reshape(m,n)
modo2 = V[1].reshape(m,n)

# Representar graficamente:
fig1 = pylab.figure()
fig1.suptitle('Imagen promedio')
pylab.gray()
pylab.imshow(im_media)
##
fig2 = pylab.figure()
fig2.suptitle('Primer modo PCA')
pylab.gray()
pylab.imshow(modo1)
##
fig3 = pylab.figure()
fig3.suptitle('Segundo modo PCA')
pylab.gray()
pylab.imshow(modo2)
pylab.show()
```

El análisis de imágenes con PCA trabaja con una matriz de datos NIM $\times$ NPIX, y puede interpretarse como la descomposición en componentes principales de NPIX variables de las que tenemos NIM observaciones. Los vectores propios indican las combinaciones lineales de píxeles que mayor variabilidad presentan en la secuencia. Así, un píxel que adopte un valor constante durante la secuencia de imágenes no será incluido en ninguno de los componentes con mayor valor propio. En cambio, aquellos píxeles o grupos de píxeles que presenten mayor variabilidad a lo largo de la secuencia darán lugar a los componentes principales de la descomposición.

Análisis PCA de una secuencia de caras: *eigenfaces* (autocaras)

Sin duda, el ejemplo paradigmático de PCA para imágenes es el análisis de una secuencia de caras. En este ejemplo vamos a ver cómo utilizar PCA para reducir la dimensionalidad en un problema en el que los datos son imágenes. Para ello vamos a utilizar una base de datos de imágenes de personajes famosos llamada Labeled Faces in the Wild, que se incluye junto a las librerías sklearn*.

*https://goo.gl/St43Qz

Lectura recomendada

A. M. Martinez; R. Benavente (junio, 1998). *The AR Face Database. CVC Technical Report* (núm. 24).

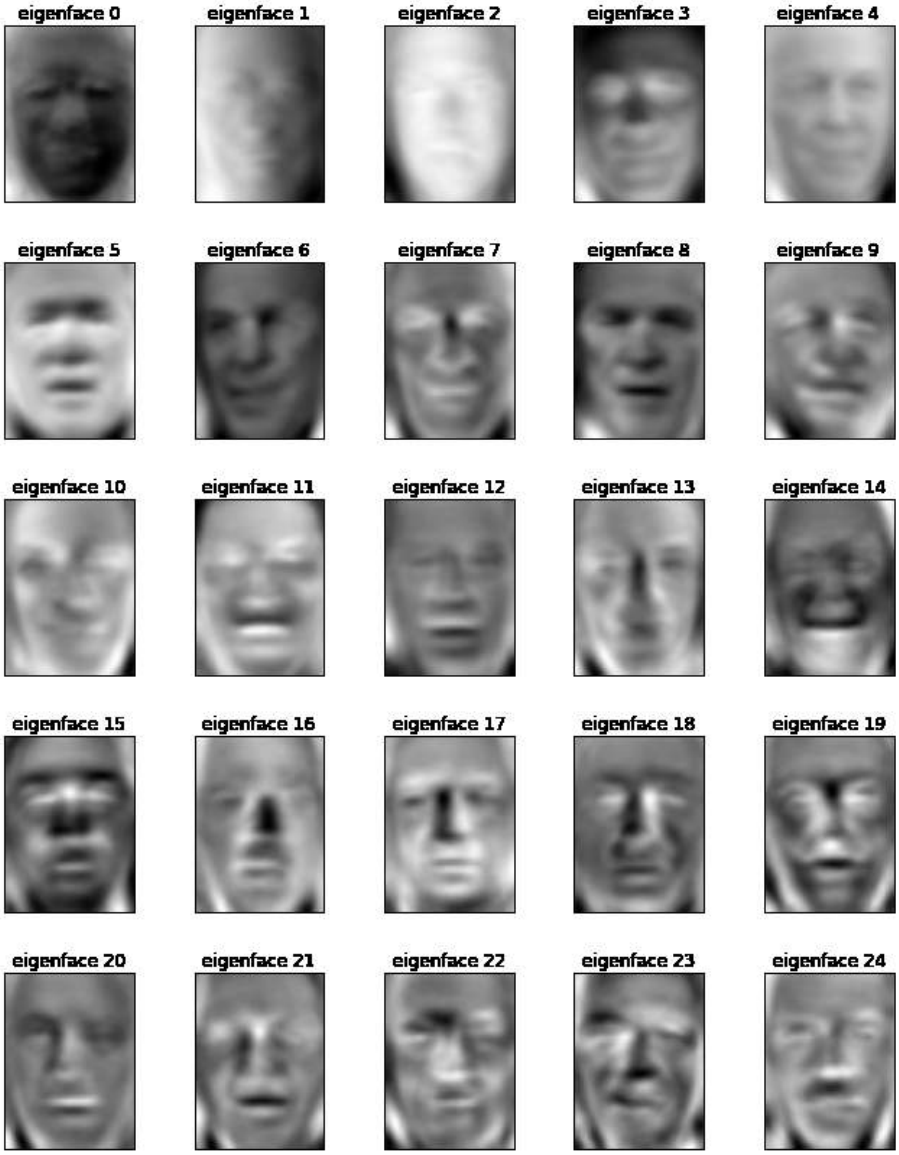
La base de datos es especialmente interesante para probar diferentes algoritmos de reconocimiento automático de caras basados en técnicas de inteligencia artificial. La base de datos contiene un total de 1288 imágenes de 50x37 píxeles. Al aplicar PCA a imágenes, cada uno de los 1850 píxeles de cada imagen corresponde a una de las variables originales del problema. La idea es

utilizar PCA para reducir la dimensionalidad de los datos y obtener una representación de las imágenes en un conjunto reducido de variables. El código 3.9 implementa el ejemplo: en primer lugar, procedemos a cargar los módulos de las librerías sklearn con las funciones para cargar las imágenes de la base de datos y las herramientas para realizar un análisis PCA, así como las librerías gráficas Matplotlib para representar los resultados (líneas 2-4). Tras leer los datos (línea 7) y cargarlos en la variable X (línea 13), esta tiene unas dimensiones de 1288x1850. En la línea 18 se aplica una descomposición PCA de los datos en un espacio de dimensionalidad reducida de ciento cincuenta variables (recordamos que inicialmente el número de variables era igual al número de píxeles $50 \times 37 = 1850$). Las *eigenfaces* más representativas se obtienen a partir de los vectores propios de la descomposición, que como sabemos representan las direcciones de máxima variabilidad en los datos. En la parte final del código los vectores propios se reordenan en forma de imágenes de tamaño 50x37 (línea 20) y a continuación se representan las veinticinco *eigenfaces* más representativas (figura 13).

Código 3.9: ejemplo de aplicación de PCA para el análisis de imágenes

```
1  # -*- coding: utf-8 -*-
2  # Cargar librerias necesarias:
3  from sklearn.datasets import fetch_lfw_people
4  from sklearn.decomposition import PCA
5  import matplotlib.pyplot as plt
6
7  # Obtener las imagenes de la base de datos:
8  lfw_people = fetch_lfw_people(min_faces_per_person=70, resize=0.4)
9
10 # Obtener numero de imagenes y size:
11 n_samples, h, w = lfw_people.images.shape
12
13 # Cargar los datos:
14 X = lfw_people.data
15
16 # numero de variables del espacio PCA reducido:
17 n_components = 150
18
19 pca = PCA(n_components=n_components, svd_solver='randomized',
20           whiten=True).fit(X)
21 eigenfaces = pca.components_.reshape((n_components, h, w))
22
23 # Representar las eigenfaces mas representativas:
24 n_row=5
25 n_col=5
26 eigenface_titles = ["eigenface %d" %i for i in range(eigenfaces.shape
       [0])]
27
28 plt.figure(figsize=(1.8 * n_col, 2.4 * n_row))
29 plt.subplots_adjust(bottom=0, left=.01, right=.99, top=.90, hspace=.35)
30 for i in range(n_row * n_col):
31     plt.subplot(n_row, n_col, i + 1)
32     plt.imshow(eigenfaces[i].reshape((h, w)), cmap=plt.cm.gray)
33     plt.title(eigenface_titles[i], size=12)
34     plt.xticks(())
35     plt.yticks(())
36 plt.show()
```

Figura 13. Representación de las veinticinco primeras *eigenfaces* del ejemplo



3.1.3. Análisis de componentes independientes (ICA)

Ejemplo de aplicación

Como ejemplo de aplicación de esta técnica, consideremos una fiesta en la que varias personas hablan de forma simultánea. En esta fiesta, cada persona percibe una señal acústica en la que se mezclan las voces del resto de los asistentes. Para que dos personas puedan mantener una conversación, es necesario que cada uno identifique las palabras emitidas por su interlocutor. Como cada persona recibe una mezcla de todos los asistentes, el cerebro debe realizar algún tipo de descomposición que le permita identificar la señal emitida por la persona con la que está hablando en ese momento. El cerebro realiza esta separación, además de por características acústicas (intensidad de la voz, tono) o visuales (movimiento de los labios), por el hecho de que cada persona emite una señal que, *estadísticamente*, es *independiente* de la del resto de los asistentes. En efecto, aunque las conversaciones compartan una temática común, el

hecho de que cada asistente es un sistema físico independiente garantiza que sus señales sean estadísticamente independientes del resto.

El análisis de componentes independientes (ICA) es una técnica que permite identificar fuentes estadísticamente independientes a partir de un conjunto de señales mezcladas. En otras palabras, ICA es una técnica de gran utilidad que permite descomponer una determinada señal en las fuentes *independientes* que la componen. A ICA también se la conoce como *separación ciega de fuentes**. En términos generales, ICA es una técnica que utiliza la *independencia estadística* para identificar las diferentes fuentes que contribuyen a una determinada mezcla de señales.

***En inglés, *Blind Source Separation* (BSS).**

Conviene resaltar aquí la importancia del término *independiente*. Las técnicas descritas anteriormente (SVD, PCA) descomponían en factores que no eran necesariamente independientes entre sí desde un punto de vista estadístico.

Ved también

En el subapartado siguiente explicaremos con mayor detalle qué se entiende por *estadísticamente independiente* desde un punto de vista matemático, lo que nos permitirá formular y comprender las bases de la técnica ICA.

Descripción de la técnica

Como hemos comentado anteriormente, ICA es un método que resulta útil cuando las fuentes a identificar provienen de sistemas distintos que emiten señales independientes. Esa es la primera hipótesis de aplicación de este método. La segunda condición necesaria es disponer de un número de señales mezcladas igual al número de componentes independientes que se desea identificar.

El subapartado siguiente está dedicado a aclarar algunos aspectos básicos sobre la estadística de señales, y en particular sobre los conceptos de independencia y correlación estadística. Volveremos al ejemplo de aplicación de la técnica ICA más adelante en este subapartado. Aquellos estudiantes que dispongan de conocimientos sobre estadística y variables aleatorias pueden dirigirse directamente a la descripción del código ICA en Python y al ejemplo de aplicación descrito en el subapartado 3.1.3.

Independencia y correlación estadística

En el lenguaje coloquial, a dos señales que no están correlacionadas se les suele llamar independientes. Desde el punto de vista matemático, la afirmación anterior no es cierta: independencia y correlación son propiedades estadísticas diferentes.

Independencia y falta de correlación

Dos variables estadísticamente independentes no presentan correlación, pero la ausencia de correlación no implica necesariamente independencia estadística. Independencia y falta de correlación son por tanto conceptos matemáticamente diferentes.

En concreto, la independencia estadística es una condición más restrictiva que la ausencia de correlación, ya que dos señales independientes necesariamente estarán descorrelacionadas pero dos señales descorrelacionadas no tiene porqué ser independientes.

Para formular este concepto de forma precisa hay que trabajar con las *densidades de probabilidad* de cada una de las señales. Supongamos, pues, que conocemos los n valores $\{x_1,x_2,\ldots,x_n\}$ que una señal $x(t)$ toma en un conjunto discreto de tiempos $\{t_1,t_2,\ldots t_n\}$. El *histograma* de valores de $x(t)$ se construye a partir del número de instantes t_i en que la señal toma valores en un determinado intervalo. Si los valores máximo y mínimo de la señal vienen dados por x_{min} y x_{max} respectivamente, dividiremos el rango de valores que toma la señal $x(t)$ en m intervalos de tamaño $\Delta x = (x_{max} - x_{min})/m$. El intervalo k-ésimo viene dado por

$$b_k = [x_{min} + (k-1)\Delta x, x_{min} + k\Delta x], k = 1 \ldots m, \tag{24}$$

de forma que el primer intervalo es $b_1 = [x_{min}, x_{min} + \Delta x]$ y el último $b_m = [x_{max} - \Delta x, x_{max}]$. Si la señal $x(t)$ toma un total de h_k valores en el intervalo b_k, el histograma de $x(t)$ vendrá dado por $\{h_1,h_2,\ldots h_m\}$. El código 3.10 indica cómo calcular histogramas y función de densidad de probabilidad en Python. En este ejemplo se parte de un conjunto de datos gaussianos con media $\mu = 200$ y desviación típica $\sigma = 30$.

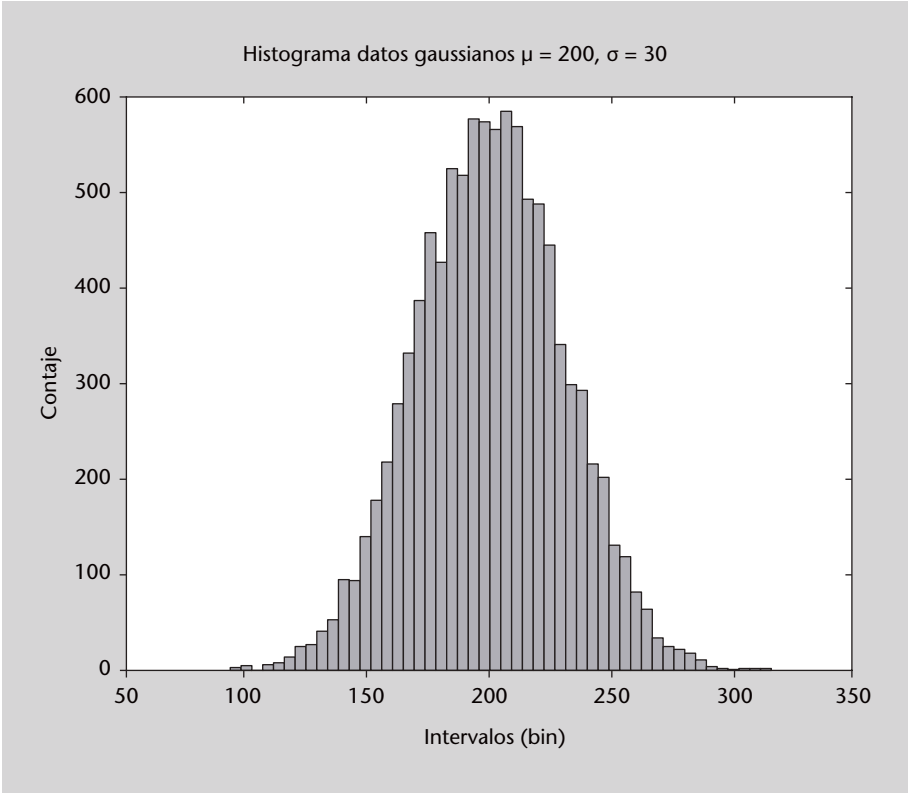
Código 3.10: cálculo de histograma y función densidad de probabilidad en Python

```
1  import numpy as np
2  import matplotlib.mlab as mlab
3  import pylab
4
5  # datos gaussianos media mu desviacion estandar sigma:
6  mu, sigma = 200, 30
7  y_datos = mu + sigma*np.random.randn(10000)
8
9  # histograma recuento de valores en cada bin:
10 nbins = 50 #numero de intervalos
11
12 fig1 = pylab.figure()
13 n, bins, patches = pylab.hist(y_datos, nbins, normed=0, facecolor='
       green', alpha=0.75)
14 pylab.title(r'$\mathrm{Histograma\ datos\ gaussianos}\ \mu=200,\ \sigma
       =30$')
15 pylab.ylabel('Contaje')
16 pylab.xlabel('Intervalos (bin)')
17 pylab.show()
18
19 # histograma normalizado de los datos (densidad de probabilidad)
20 fig2 = pylab.figure()
21 p, bins, patches = pylab.hist(y_datos, nbins, normed=1, facecolor='blue
       ', alpha=0.75)
22
23 # Ajuste de los datos con una funcion de densidad
24 # de probabilidad gaussiana:
25 y_pdf = mlab.normpdf(bins, mu, sigma)
26 l = pylab.plot(bins, y_pdf, 'r--', linewidth=1)
27 pylab.ylabel('Probabilidad')
28 pylab.xlabel('Intervalos (bin)')
29 pylab.title(r'$\mathrm{Densidad de probabilidad}\ \mu=200,\ \sigma=30$'
       )
30 pylab.grid(True)
31 pylab.show()
32
```

```
33 # Comprobar que la integral de la funcion densidad de probabilidad
34 # es igual a 1:
35
36 Integral = sum(p*np.diff(bins))
37 print(Integral)
```

La figura 14 representa el número de datos que toman valores dentro de cada intervalo (contaje).

Figura 14. Histograma de los datos gaussianos del ejemplo 3.10

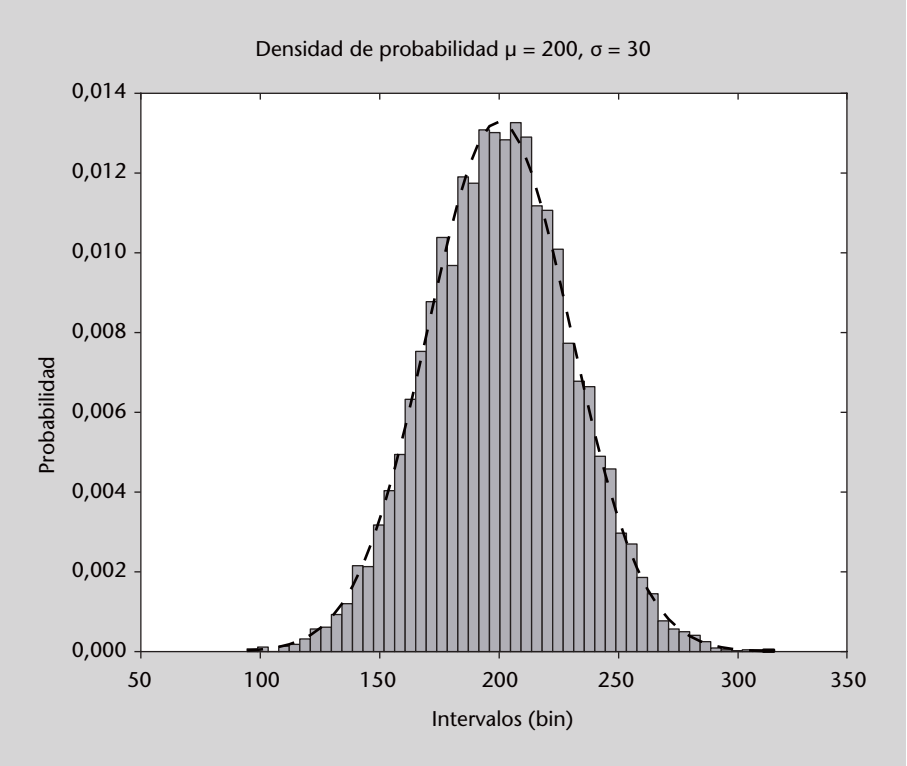


En la figura 15 el histograma de los datos ha sido normalizado al número de valores, de forma que lo que se representa es la función de densidad de probabilidad. La línea discontinua indica la forma que tiene una distribución de probabilidad gaussiana con media $\mu = 200$ y desviación típica $\sigma = 30$.

La probabilidad de que la señal $x(t)$ tome un valor en el intervalo b_k vendrá dada por el cociente entre el número de veces que $x(t)$ toma valores en el intervalo b_k y el número total de valores de la señal, n. Nos referiremos entonces a la *función probabilidad* de $x(t)$ como

$$p_k = \frac{h_k}{n}. \qquad (25)$$

Figura 15. Densidad de probabilidad de los datos del ejemplo 3.10



Como cabría esperar, la suma de todas las probabilidades $\sum_{k=1}^{m} p_k = 1$ es un puesto que corresponde a la probabilidad de que la señal adopte un valor en algún intervalo b_k.

La *densidad de probabilidad* $\rho_x(k)$ en un intervalo b_k se define entonces como el número de valores que se han producido en dicho intervalo h_k normalizado con el área total del histograma

$$\rho_x(k) = \frac{h_k}{n\Delta x}. \tag{26}$$

En el límite en el que el tamaño del intervalo Δx se hace infinitesimalmente pequeño (o, equivalentemente, $m \to \infty$) y el número de valores de la señal n tiende a infinito, $\rho_x(k)$ da lugar a la *función densidad de probabilidad continua*

$$\rho_x(x) = \lim_{\Delta x \to 0, m \to \infty} \rho_x(k). \tag{27}$$

Volvamos al asunto de la independencia y a la ausencia de correlación estadística entre dos señales $x(t)$ e $y(t)$. De forma similar a lo que hicimos anteriormente, podemos dividir los rangos de valores que toman las variables x e y en m intervalos de tamaño Δx y Δy respectivamente. Entonces podemos definir la *densidad de probabilidad conjunta* $\rho_{xy}(x,y)$ de $x(t)$ e $y(t)$ a partir del

Ved también

El anexo de este módulo se dedica a repasar los conceptos básicos de estadística que son necesarios en este apartado.

número de veces que la señal x e y tomen valores en diferentes intervalos *de forma simultánea*. Esta función nos indica, por ejemplo, en cuántos instantes de tiempo $t_1,t_2,\dots t_n$ la señal x ha tomado valores en el primer intervalo y en el último, lo que nos permite conocer la probabilidad de que se produzcan un par de valores (x,y) a la vez.

Cuando las señales x e y son *estadísticamente independientes*, la densidad de probabilidad conjunta $\rho_{xy}(x,y)$ viene dada por el producto de las *densidades de probabilidad marginales* de cada una de las señales $\rho_x(x)$ y $\rho_y(y)$

$$\rho_{xy}(x,y) = \rho_x(x)\rho_y(y), \tag{28}$$

puesto que, según el teorema de Bayes, la probabilidad de que se dé el par de valores (x_0,y_0) es el producto de las probabilidades de que x_0 se dé en x y de que y_0 se dé en y de forma independiente. Cuando las señales no son independientes tenemos, en cambio que

$$\rho_{xy}(x,y) = \rho_x(x)\rho_y(y) + \rho_{xy}(x|y) \tag{29}$$

donde $\rho_{xy}(x|y)$ es la *probabilidad condicionada* de que se dé un valor de x habiéndose dado un valor y. De la expresión anterior se deriva que *los momentos centrales conjuntos* de cualquier orden r,s cumplen la condición

$$E[(x-\bar{x})^r(y-\bar{y})^s] = E[(x-\bar{x})^r]E[(y-\bar{y})^s] \tag{30}$$

para cualquier valor entero de r,s. En particular, para $r = s = 1$ tenemos que la *covarianza* $Cov(x,y) = E[(x-\bar{x})(y-\bar{y})]$ entre x e y es nula

$$Cov(x,y) = E[(x-\bar{x})(y-\bar{y})] = E[(x-\bar{x})]E[(y-\bar{y})] = 0, \tag{31}$$

puesto que los primeros momentos centrales de cada una de ellas son nulos

$$E[(x-\bar{x})] = \int_{-\infty}^{\infty}(x-\bar{x})\rho_x(x)dx = \int_{-\infty}^{\infty}x\rho_x(x)dx - \bar{x} = 0 \tag{32}$$

$$E[(y-\bar{y})] = \int_{-\infty}^{\infty}(y-\bar{y})\rho_y(y)dy = \int_{-\infty}^{\infty}y\rho_y(y)dy - \bar{y} = 0. \tag{33}$$

En efecto, cuando dos señales son estadísticamente independientes, su covarianza es nula.

A partir de la covarianza se puede definir la *correlación* entre las dos señales

$$Corr(x,y) = \frac{Cov(x,y)}{\sigma_x \sigma_y} \tag{34}$$

que da cuenta de la covarianza normalizada con las desviaciones estándar de cada una de las señales σ_x y σ_y.

El código 3.11 describe cómo calcular las matrices de covarianza y correlación entre dos variables.

Código 3.11: cálculo de matrices de covarianza y correlación entre dos conjuntos de datos

```
from scipy import stats
from numpy import *

# datos:
mu1, sigma1 = 200, 30
x = mu1 + sigma1*random.randn(3).T
mu2, sigma2 = 10, 5
y = mu2 + sigma2*random.randn(3).T

# matriz de covarianza:
covmat = cov(x,y,bias=1)

# matriz de correlacion:
corrmat = corrcoef(x,y)

>>> print(covmat)
[[ 902.94004704  -86.47788485]
 [ -86.47788485   26.52406588]]

>>> print(var(x), var(y))
902.940047043 26.5240658823

>>> print(corrmat)
[[ 1.          -0.55879891]
 [-0.55879891  1.        ]]

# relacion entre ambas:
>>> cov(x/std(x),y/std(y),bias=1)
array([[ 1.        , -0.55879891],
       [-0.55879891,  1.        ]])
```

Todo esto nos lleva a concluir que dos señales independientes tienen una correlación nula. Lo que no es necesariamente cierto es que dos señales con correlación nula sean independientes. Para que sean independientes, además de la correlación también deben ser nulos los momentos centrales conjuntos de cualquier orden que cumplan la ecuación 30.

Por ejemplo, para $r = s = 2$ tenemos, utilizando la ecuación 82 que

$$E[(x-\bar{x})^2(y-\bar{y})^2] = E[(x-\bar{x})^2]E[(y-\bar{y})^2] = Var[x]\,Var[y] \tag{35}$$

Información mutua

La información mutua es una medida de la relación estadística entre dos señales. De forma cualitativa, la información mutua puede entenderse como una función de correlación que tiene en cuenta relaciones lineales y no-lineales entre dos variables.

En concreto, lo que mide la información mutua es la probabilidad de que una variable tome un valor determinado condicionada a que la otra variable haya tomado otro. O lo que es lo mismo, la cantidad de información que tenemos de una de ellas en caso de conocer la otra.

Para eso introducimos el concepto de *entropía de Shannon* de una señal x, definida como el valor esperado del logaritmo de su densidad de probabilidad

$$I(x) = -E[\log(\rho_x(x))] = -\int \rho_x(x)\log(\rho_x(x))dx. \tag{36}$$

Cuanto mayor es la entropía de Shannon de una señal, menor es su contenido informacional.

De esta forma una señal aleatoria presentará mayor entropía y menor contenido informacional que una señal periódica. En el ámbito de las comunicaciones, la ecuación 36 permite cuantificar la información que se transmite a través de un determinado canal. De forma equivalente, es posible definir la *información mutua* entre dos señales como el valor esperado del logaritmo de su distribución conjunta

$$I(x,y) = -E[\log(\rho_{xy}(x,y))] = -\int\int \rho_{xy}(x,y)\log(\rho_{xy}(x,y))dxdy \tag{37}$$

En el caso de que x e y sean señales estadísticamente independientes, la probabilidad conjunta factoriza como el producto de las probabilidades marginales $\rho_{xy}(x,y) = \rho_x(x)\rho_y(y)$ (ecuación 28), y la información mutua viene dada por la

$$I(x,y) = -E[\log(\rho_{xy}(x,y))] = -E[\log(\rho_x(x))]E[\log(\rho_y(y))] = I(x)I(y) \qquad (38)$$

Cuando ambas señales tienen alguna relación, sea lineal o no lineal, la probabilidad condicional $\rho_{xy}(x|y)$ no es nula, por lo que se cumple la ecuación 29 y la información mutua toma valores positivos $I(x,y) = I(x)I(y) + I(x|y) > I(x)I(y)$.

ICA descompone una matriz de datos en componentes estadísticamente independientes.

Puesto que la definición incluye directamente la densidad de probabilidad conjunta, la entropía mutua tiene en cuenta todos los momentos conjuntos de las distribuciones y no solamente la covarianza, y por lo tanto, la información mutua es una medida de la independencia estadística entre dos señales.

Ejemplo: separación de fuentes independientes

Las diferentes implementaciones de ICA obtienen las señales independientes de una mezcla mediante un proceso de minimización de la información mutua. Adicionalmente, la mayoría de los métodos incluyen alguna estrategia que garantice que las fuentes, ademas de independientes, sean lo más simples posible (principio de parsimonia). Para ello utilizan diferentes medidas de complejidad computacional de las señales fuente obtenidas.

La versión que vamos a utilizar está contenida en la librerías sklearn, que utilizan el algoritmo FASTICA desarrollado por Aapo Hyvarinen. El código 3.12 describe la utilización de la técnica ICA en Python. El programa empieza definiendo dos señales sinusoidales con frecuencias diferentes. En este ejemplo introductorio, cada una de esta señales se corresponderían a las voces de dos de los participantes en la fiesta. Estas señales aparecen representadas en la figura 16 y serán consideradas como las fuentes originales a identificar mediante la técnica ICA.

Lectura recomendada

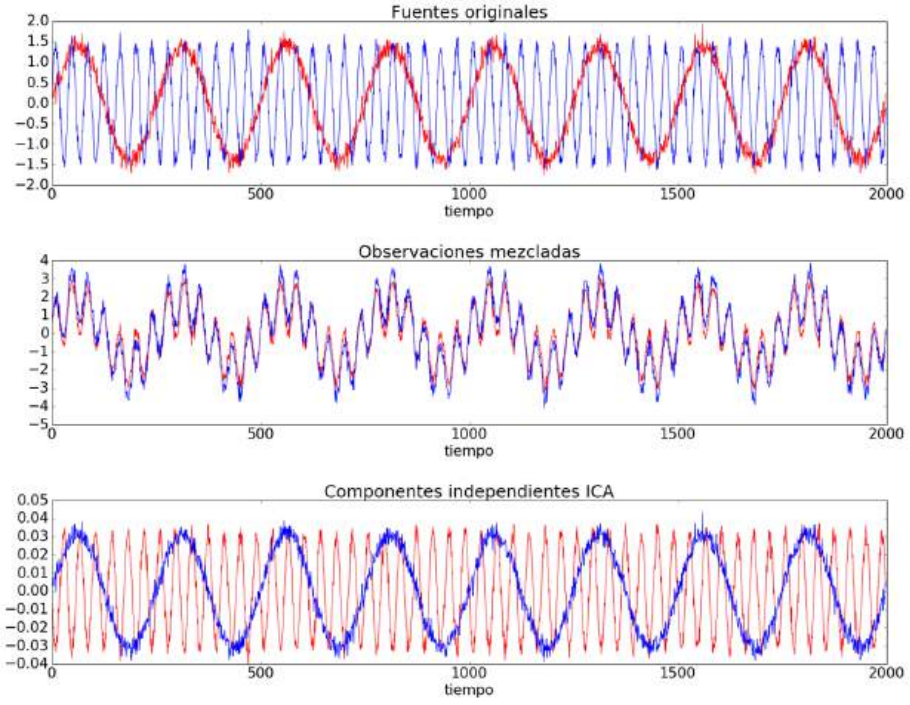
Aapo Hyvarinen.(1999). Fast and Robust Fixed-Point Algorithms for Independent Component Analysis IEEE Transactions on Neural Networks, 10(3):626-634, .
A. Hyvarinen, E. Oja (1997). A fast Fixed-Point Algorithms for Independent Component Analysis Neural Computation, 9(7)1483-1492.
http://www.cis.hut.fi/projects/ica/fastica/

Código 3.12: ejemplo de descomposición ICA en Python

```
1  # -*- coding: utf-8 -*-
2  import numpy as np
3  import matplotlib.pyplot as plt
4
5  # Generar señales sinteticas:
6  np.random.seed(0)
7  n_samples = 2000
8  time = np.linspace(0, 8, n_samples)
9
10 # Fuentes originales:
11 s1 = np.sin(2 *np.pi* time)  # Signal 1 : sinusoidal signal
12 s2 = np.sin(13 *np.pi* time)  # Signal 2 : sinusoidal signal
13
14 S = np.c_[s1, s2]
15
16 # Añadir ruido:
```

```
S += 0.1 * np.random.normal(size=S.shape)  # Add noise

S /= S.std(axis=0)  # Estandarizar datos
# Mix signals:
A = np.array([[1, 1], [1.5, 1.0]])  # Matriz de mezcla
X = np.dot(S, A.T)  # Observaciones mezcladas

# Descomposicion ICA :
from sklearn.decomposition import FastICA, PCA
ica = FastICA(n_components=3)
S_ = ica.fit_transform(X)  # Reconstruccion ICA

# representacion de resultados :
plt.figure()
plt.figure(figsize=(20,20))

models = [S, X, S_]
names = ['Fuentes originales','Observaciones mezcladas',
         'Componentes independientes ICA']
colors = ['red', 'blue', 'green']

for ii, (model, name) in enumerate(zip(models, names), 1):
    plt.subplot(4, 1, ii)
    plt.title(name, size=25)
    for sig, color in zip(model.T, colors):
        plt.plot(sig, color=color)
        plt.xlabel('tiempo', fontsize=21)
        plt.tick_params(labelsize=21)

plt.subplots_adjust(0.09, 0.04, 0.94, 0.94, 0.26, 0.46)
plt.show()
```

Figura 16. En el código 3.12, las fuentes originales son dos señales sinusoidales con frecuencias diferentes



A continuación se construyen los datos mezclados aplicando una matriz de mezclas A a las fuentes originales. En el ejemplo de la fiesta, los datos mezclados corresponden a la señal total mezclada que oiría cada uno de los asistentes. Como resultado, se obtienen dos señales de mezcla que son combinación lineal de las fuentes originales y que se representan en la figura 16.

La descomposición ICA se aplica a estas dos señales mezcladas con la intención de poder identificar las fuentes independientes subyacentes. El resultado de la descomposición son las dos señales representadas en la figura 16, y como puede verse, cada una de ellas corresponde a una de las dos fuentes originales de partida. El código puede utilizarse para explorar los resultados obtenidos con la técnica ICA al aplicarla a otro tipo de señales y con matrices de mezcla diferentes.

3.1.4. Factorización de matrices no-negativas (NMF)

Ejemplo de aplicación

NMF es una herramienta utilizada de forma habitual en aplicaciones de *minería de textos*, por lo que el ejemplo más representativo es el análisis de aparición de palabras en textos. El método NMF resulta útil para caracterizar conjuntos de datos que no tomen valores negativos, como por ejemplo, datos de recuento (número de votos por candidatura, aparición de palabras en documentos, frecuencia de eventos en una planta industrial, etc.). Supongamos que tenemos un conjunto de N textos que queremos caracterizar mediante un subconjunto de K palabras. El sistema puede representarse mediante una matriz de ocurrencias A de tamaño $N \times K$ cuyos componente i,j indicasen el número de veces que la palabra j aparece en el texto i.

$$\begin{array}{c} \\ \text{texto 1} \\ \text{texto 2} \\ \dots \\ \text{texto } N \end{array} \begin{array}{c} \begin{array}{cccc} \text{palabra 1} & \text{palabra 2} & \dots & \text{palabra } K \end{array} \\ \begin{pmatrix} a_{1,1} & a_{2,1} & \dots & a_{1,K} \\ a_{2,1} & a_{2,2} & \dots & a_{2,K} \\ \vdots & \vdots & \ddots & \vdots \\ a_{N,1} & a_{N,2} & \dots & a_{N,K} \end{pmatrix} \end{array}$$

Eligiendo las K palabras de forma estratégica podremos caracterizar los textos según el número de veces que aparezca cada palabra. Por ejemplo, en caso de que los textos provengan de una web de noticias, parece lógico que aquellos textos en los que más veces aparezca la palabra *internacional* o *Europa* estarán referidos a noticias de ámbito internacional, mientras que en los que aparezcan las palabras *alcalde* o *ayuntamiento* tendrán que ver con noticias de ámbito local. Evidentemente, también puede suceder que una noticia local sobre ayudas agrarias incluya la palabra *Europa*, o que una de ámbito internacional se refiera al alcalde de una determinada ciudad. El método NMF supera esta ambiguedad definiendo un conjunto de características que permitan agrupar los textos de forma eficiente. En resumen, NMF nos va a permitir agrupar los textos siguiendo un criterio que permita identificarlos mediante la aparición de palabras clave.

Descripción de la técnica

NMF realiza una descomposición en factores una matriz de datos no-negativa A de la forma

$$A \approx W \cdot F \tag{39}$$

donde las matrices no-negativas F y W son conocidas como *matriz de características* y *matriz de pesos* respectivamente. La matriz de características F, de tamaño $M \times K$, define M características y pondera cada una de ellas mediante la importancia que tiene cada una de las K palabras. Las características corresponden a temas genéricos que han sido definidos a partir de la agrupación de palabras en diferentes textos. La componente i,j de F representa la importancia de la palabra j en la característica i. Su aspecto sería el siguiente

$$\begin{array}{l} \\ \text{característica 1} \\ \text{característica 2} \\ \ldots \\ \text{característica } M \end{array} \begin{array}{c} \begin{array}{cccc} \text{palabra 1} & \text{palabra 2} & \ldots & \text{palabra } K \end{array} \\ \begin{pmatrix} f_{1,1} & f_{1,2} & \ldots & f_{1,K} \\ f_{2,1} & f_{2,2} & \ldots & f_{2,K} \\ \vdots & \vdots & \ddots & \vdots \\ f_{M,1} & f_{M,2} & \ldots & f_{M,K} \end{pmatrix} \end{array}$$

Por otra parte, la matriz de pesos W le atribuye un peso a cada una de las características en función de su importancia en cada uno de los N textos. Sus M columnas corresponden a las características y sus N filas a los textos analizados, de forma que la componente i,j de W indica la importancia que tiene la característica j en el texto i

$$\begin{array}{l} \\ \text{texto 1} \\ \text{texto 2} \\ \ldots \\ \text{texto } N \end{array} \begin{array}{c} \begin{array}{cccc} \text{característica 1} & \text{característica 2} & \ldots & \text{característica } M \end{array} \\ \begin{pmatrix} w_{1,1} & w_{1,2} & \ldots & w_{1,M} \\ w_{2,1} & w_{2,2} & \ldots & w_{2,M} \\ \vdots & \vdots & \ddots & \vdots \\ w_{M,1} & w_{M,2} & \ldots & w_{N,M} \end{pmatrix} \end{array}.$$

La ecuación 39 debe interpretarse como una transformación que permite definir M características de forma que los datos originales puedan factorizarse en la forma

$$\text{textos} \times \text{palabras} \approx \text{textos} \times \text{características} \cdot \text{características} \times \text{palabras}. \tag{40}$$

Factorización

NMF permite factorizar una matriz de datos definida positiva en el producto de dos matrices no-negativas con una estructura más simple.

Evidentemente, uno de los parámetros que deben definirse para aplicar NMF es el número de características M que se desean. Un valor demasiado alto resultaría en un etiquetado demasiado específico de los textos, mientras que un valor excesivamente pequeño agruparía los textos en unas pocas características demasiado genéricas y por tanto carentes de información relevante. La matriz de características F es equivalente a la matriz de vectores propios que obteníamos con otras técnicas de factorización de datos como PCA o ICA. La diferencia principal entre NMF y PCA radica en que en NMF la matriz F es no-negativa y por tanto la factorización de A puede interpretarse como una descomposición acumulativa de los datos originales.

Conviene precisar que la descomposición NMF no es única, es decir, dado un número de características M hay más de una forma de descomponer los datos en la forma de la ecuación 39. Lo ideal es aplicar el procedimiento e interpretar los resultados para obtener información adicional de los datos. El algoritmo que se encarga de realizar esta factorización no es trivial y escapa al alcance de este este libro. En términos generales, las diferentes implementaciones de NMF siguen un proceso de optimización multidimensional que minimiza el residuo R de la descomposición

$$R = W \cdot F - A. \tag{41}$$

Implementación en Python

El código siguiente describe cómo realizar una descomposición NMF utilizando las librerías sklearn. Primero se define una matriz de datos A, que en este caso puede interpretarse como datos de contaje de dos palabras en seis textos. Puesto que se escoge una descomposición NMF con dos características, A es descompuesta en dos matrices no negativas W y F de tamaños 6x2 y 2x2 respectivamente. La validez del resultado puede analizarse comprobando el residuo de la aproximación, es decir las diferencias entre la matriz de datos original A y la aproximación NMF $W \cdot F$.

Código 3.13: implementación de NMF en Python descrita en el libro *Programming Collective Intelligence*

```
1 import numpy as np
2 A = np.array([[3,4], [2, 5], [2.1, 1.1], [4.1, 1], [5, 1.8], [3, 4.5]])
3 from sklearn.decomposition import NMF
4 model = NMF(n_components=2, init='random', random_state=0)
5 W = model.fit_transform(A)
6 F = model.components_
7 # comprobar que A ~ W*F:
8 print(A-np.dot(W,F))
```

Otras implementaciones

Existen otras implementaciones que utilizan algoritmos de optimización basados en el método de gradientes proyectados, como la que puede obtenerse en en el libro *Programming Collective Intelligence* de T. Segaran o también en la siguiente dirección web: http://www.csie.ntu.edu.tw/~cjlin/nmf/, basada en el artículo: C. J. Lin. «Projected gradient methods for non-negative matrix factorization». *Neural Computation* (n.º 19(2007), págs. 2756-2779).

Lectura complementaria

T. Segaran (2007). *Programming Collective Intelligence*. EE. UU.: O'Really

Análisis de textos

Como hemos comentado, una de las aplicaciones más importantes de la técnica NMF es el análisis de documentos. Para poder construir la matriz de recuento de palabras, primero hay que realizar algunas operaciones de filtrado de los textos obtenidos. Supongamos que disponemos de un conjunto de ficheros de texto en formato .txt. Cada texto puede leerse mediante la rutina Python descrita en el código 3.14.

Código 3.14: lectura de ficheros de texto y creación de diccionarios en Python

```
# -*- coding: cp1252 -*-
import glob
import numpy
import string

def textos(path):
    # listar todos los ficheros con extensión .txt del directorio path:
    filelist = glob.glob(path)
    docs = []

    # bucle sobre todos los ficheros en el directorio:
    for kfile in filelist:
        fp = file(kfile) # abrir fichero
        txt = fp.read() # leer fichero
        fp.close() # cerrar fichero

        # filtrar caracteres:
        txt = txt.lower() # convertir a minusculas
        # substituir retornos de carro por espacios
        txt = txt.replace('\n',' ')
        # borrar caracteres ASCII que no sean el espacio
        #(ascii 32), o las letras a-z (ascii 97-122):
        char_ok = [32] + range(97,122)
        for i in range(256):
            if i not in char_ok:
                txt = txt.replace(chr(i),'')

        # construir una lista con las palabras del texto:
        palabras = txt.split()
        # crear un diccionario:
        dicc = { }
        for ipal in range(len(palabras)):
            pl = palabras[ipal];
            if len(pl)>4: # excluir palabras de menos de 5 caracteres
                if pl in dicc:
                    dicc[pl].append(ipal) # localizacion de las
                        apariciones
                else:
                    dicc[pl] = [ipal]
        # añadir numero de apariciones al final del diccionario
        for ipal in range(len(palabras)):
            pl = palabras[ipal];
            if pl in dicc:
```

```
43                  #dicc[pl].append(len(dicc[pl])) # numero de apariciones
44                  dicc[pl].insert(0,len(dicc[pl]))
45          # agregar diccionario de cada documento
46          docs.append(dicc)
47
48      return docs
```

El código 3.14 utiliza un *string* de entrada *path* en el que se indique el directorio en el que están ubicados los ficheros de texto. En las líneas 6-9 del código genera una lista *filelist* con los nombres de los ficheros de texto. El bucle de la línea 12 recorre todos los ficheros de texto y ejecuta la secuencia de acciones siguientes para cada uno de ellos: abrir fichero, leer fichero, cerrar fichero, eliminar todos los caracteres ASCII que sean diferentes a las letras a-z o el espacio (líneas 23-26). Una vez filtradas, se construye una lista con las palabras que aparecen en el texto mediante la función Python *split*.

El siguiente paso es crear un diccionario Python *dicc*, que consiste en una lista de las palabras que aparecen en el texto que va acompañada de las posiciones en las que aparece cada palabra (líneas 30-38). En el ejemplo sólo se han incorporado al diccionario las palabras con más de cuatro caracteres. Al principio de esta lista añadimos un nuevo valor que indica el número de veces que cada palabra aparece en el texto, por lo que cada entrada del diccionario final contiene la palabra, y a continuación un vector con el número de veces que aparece en el texto y las posiciones del texto en las que aparece. La última instrucción del bucle de ficheros incorpora el diccionario de cada texto en una variable biblioteca *docs* que agrupa todos los diccionarios (un diccionario por fichero de texto). La rutina retorna la variable *docs* con todos los diccionarios de todos los textos encontrados en la carpeta *path*.

El código 3.15 indica la instrucción necesaria para generar una biblioteca de diccionarios a partir de los ficheros de texto ubicados en la carpeta C:/textos/.

Código 3.15: llamada a la rutina textos para generar una biblioteca de diccionarios a partir de un conjunto de ficheros de texto

```
1 >>> docs = textos('C:\\textos\\*.txt')
2
3 >>> docs[0]
4 {'arrhythmogenic': [3, 2, 11, 135], 'catecholaminergic': [1, 147], '
      significant': [1, 80], 'family': [1, 52], 'providers': [1, 67], '
      abridge': [1, 125], 'correlations': [1, 154], 'outcomes': [1, 49],
       'helped': [1, 14], 'namely': [1, 137], 'determined': [1, 100], '
      decisions': [1, 72], 'occur': [1, 85], 'recognition': [1, 105], '
      current': [1, 127], 'polymorphic': [1, 148], 'realm': [1, 33], '
      knowledge': [1, 128], 'mutated': [1, 6], 'implementation': [1,
      114], 'technology': [1, 92], 'coupled': [1, 93], 'pathogenesis':
      [1, 151], 'research': [1, 1], 'better': [1, 55], 'their': [9, 8,
      7, 6, 5, 18, 48, 57, 65, 74], 'genetic': [3, 2, 35, 131], '
      clinical': [1, 87], 'identification': [1, 4], 'treatment': [1,
      116], 'which': [1, 110], 'inherited': [3, 2, 10, 134], 'forward':
      [1, 15], 'progress': [3, 2, 26, 79], 'tachycardia': [1, 150], '
      fatal': [1, 108], 'cause': [1, 9], 'possible': [1, 40], '
      potentially': [1, 107], 'practice': [1, 88], 'genotypephenotype':
      [1, 153], 'diseases': [5, 4, 3, 12, 109, 136], 'advances': [1,
      90], 'genes': [1, 7], 'background': [1, 132], 'symptomatic': [1,
      44], 'understanding': [3, 2, 16, 97], 'pathophysiology': [1, 19],
      'genetically': [1, 99], 'assists': [1, 102], 'understand': [1,
      56], 'discussed': [1, 157], 'members': [1, 53], 'improving': [1,
```

```
    96], 'article': [1, 123], 'conjunction': [1, 63], 'continued': [3,
     2, 0, 78], 'earlier': [3, 2, 104, 113], 'improve': [1, 47], '
    leads': [1, 111], 'short': [1, 141], 'decades': [1, 24], 'syndrome
    ': [5, 4, 3, 140, 143, 145], 'arrhythmias': [3, 2, 36, 101], '
    ventricular': [1, 149], 'patients': [1, 45], 'continue': [1, 83],
    'risks': [1, 58], 'allow': [1, 60], 'brugada': [1, 144], 'changes'
    : [1, 81]}
```

En la línea 3 se obtiene el diccionario correspondiente al primero de los textos encontrados en la carpeta. Cada entrada del diccionario consta de un campo para la palabra y de un vector en el que la primera componente indica el número de veces que aparece la palabra en el texto y los siguientes, las posiciones en las que aparece la palabra a lo largo del texto. Por ejemplo, la palabra *continued* aparece un total de tres veces en el primer documento, concretamente en las palabras primera [0], tercera [2] y en la que ocupa la posición 79 [78].

La rutina 3.16 lista las 10 palabras más frecuentes de cada uno de los textos analizados. Con esta información y la biblioteca de diccionarios, podemos construir la matriz de ocurrencias de palabras en textos.

Código 3.16:

```
1  from textos import *
2  import numpy
3  import operator
4
5
6  # obtener un conjunto de diccionarios:
7  docs = textos('C:\\textos\\*.txt')
8
9  # calcular la matriz de recuento de palabras
10 # tamaño N textos x K palabras
11
12 dicc_sort = []
13 # loop over dictionaries
14 for k in range(len(docs)):
15 # ordenar diccionario por numero de apariciones
16     ds = sorted(docs[k].iteritems(),key=operator.itemgetter(1),reverse=
           True)
17     dicc_sort.append(ds)
18 # Seleccionar un grupo de K palabras que
19 # aparezcan en los diferentes diccionarios:
20     print(k)
21 # listar las 10 palabras mas utilizadas en cada texto:
22     for kw in range(0,10):
23         print(ds[kw][0])
```

Consideremos una matriz de occurrencias de $K = 5$ palabras en $N = 4$ textos como la siguiente:

$$A = \begin{array}{c} \\ \text{texto 1} \\ \text{texto 2} \\ \text{texto 3} \\ \text{texto 4} \end{array} \begin{array}{c} \begin{array}{ccccc} \text{palabra 1} & \text{palabra 2} & \text{palabra 3} & \text{palabra 4} & \text{palabra 5} \end{array} \\ \begin{pmatrix} 14 & 1 & 3 & 2 & 1 \\ 1 & 2 & 0 & 14 & 1 \\ 15 & 1 & 3 & 2 & 4 \\ 0 & 1 & 3 & 21 & 0 \end{pmatrix} \end{array}$$

En este caso la primera palabra aparece mucho en los textos 1 y 3 y poco en los otros, mientras que los textos 2 y 4 comparten una alta aparición de la palabra 4. Las matrices *F*,*W* resultantes de aplicar el código 3.13 a la matriz anterior se indican en el código 3.17.

Código 3.17:

```
>>> F
matrix([[  1.56842028e+01,   1.10030496e+00,   3.18461101e+00,
           2.07222144e+00,   2.79548846e+00],
        [  8.29519007e-14,   6.26129490e-01,   7.91626659e-01,
           8.31254256e+00,   1.45667562e-01]])
>>> W
matrix([[  8.78583342e-01,   2.08132147e-02],
        [  5.58125546e-02,   1.66257600e+00],
        [  9.69522495e-01,   1.99310161e-05],
        [  5.33686016e-03,   2.53004685e+00]])
```

La interpretación de los resultados es la siguiente: La matriz *F* tiene dos filas que corresponden a cada una de las dos características que se han extraído por la técnica NMF. En esta matriz *F*, las columnas corresponden a cada una de las palabras. Analizando los valores de *F* que aparecen en el código 3.17, la primera conclusión a la que llegamos es que la primera característica tiene que ver principalmente con la primera palabra puesto que la componente *F*(1,1) de la matriz toma un valor elevado. Lo mismo sucede con la segunda característica, que aparece relacionada principalmente a la cuarta palabra (componente *F*(2,4) de la matriz *F*). Por lo tanto la matriz *F* nos describe dos características que permiten analizar los textos, una relacionada con la palabra 1 y otra con la palabra 4. Esta información de hecho es relevante puesto que pone de manifiesto algunas de las características de la matriz de ocurrencias inicial.

La matriz *W*, a su vez, pondera la importancia de cada una de las dos características en los cuatro textos analizados. La característica 1 (primera columna de *W*), por ejemplo, adopta valores más altos en los textos 1 y 3 (componentes *W*(1,1) y *W*(3,1)). De forma similar, la segunda característica (relacionada con la palabra 4), permite caracterizar los textos 2 y 4 (componentes *W*(2,2) y *W*(4,2)).

En resumen, la descomposición NMF nos ha permitido encontrar dos características que permiten agrupar los textos según los patrones de aparición de palabras en cada uno de ellos. En este ejemplo las características están relacionadas principalmente con una de las palabras, pero en general serán una combinación lineal de algunas de las palabras analizadas. Conviene recordar que la inicialización aleatoria de las matrices *W*,*F* que aparece en el código 3.13 ocasiona que los resultados obtenidos sean ligeramente diferentes cada vez que se ejecuta el código. También cabe notar que los resultados pueden ser mejorados mediante el ajuste de parámetros como el número máximo de iteraciones.

3.2. Discriminación de datos en clases

En los subapartados anteriores hemos visto que la técnica PCA sirve fundamentalmente para *caracterizar* un conjunto de datos a partir de las variables que presentan una mayor variabilidad. De igual forma, ICA se utiliza para *identificar* los diferentes mecanismos subyacentes que dan lugar a la estructura de los datos. El análisis de discriminantes sirve para determinar qué variables de un conjunto de datos son las que discriminan entre dos o más clases predefinidas.

En términos generales, puede decirse que PCA es útil para caracterizar, ICA para identificar y LDA para *discriminar*.

Dicotomía según la RAE

Según el diccionario de la Real Academia Española *dicotomía* es el método de clasificación en que las divisiones y subdivisiones solo tienen dos partes.

Aquí nos limitaremos al caso en el que se pretendan distinguir dos clases diferentes, por lo que lo trataremos como método de *dicotomía*. Nos limitaremos al caso de dos clases porque las expresiones matemáticas son más simples y permiten una comprensión más directa de la técnica. La extensión de la técnica a casos en los que haya que distinguir más de dos grupos no supone una excesiva dificultad adicional.

3.2.1. Análisis de discriminantes lineales (LDA)

Ejemplo de aplicación

Es frecuente que un conjunto de datos esté compuesto por observaciones que puedan ser clasificadas en dos grupos. Por ejemplo, imaginemos una base de datos con mediciones biomecánicas de atletas realizadas durante la ejecución de un ejercicio de salto de altura. Resultaría interesante saber qué variables biomecánicas de los saltadores permiten discriminar entre los atletas que alcanzan el podio del resto de participantes. La base de datos incluirá medidas de varias magnitudes relevantes como la velocidad de salida, la longitud de la primera zancada o la inclinación del cuerpo al realizar el último paso. Estas n variables biomecánicas se medirán para un conjunto de m saltadores durante campeonatos internacionales. Las mediciones darán lugar a un conjunto de n variables y m observaciones, y por tanto, a una matriz de datos de tamaño $m \times n$. A las observaciones deberemos añadir un vector de etiquetas que identifique cada observación como grupo 1 si el saltador ocupó el podio o como grupo 2 si finalizó el campeonato a partir de la cuarta posición.

Nuestro interés es saber qué variables o combinación de las mismas nos permiten identificar a los saltadores que ocupan el podio en competiciones internacionales. Un análisis PCA de estos datos nos permitiría identificar las variables que mejor expresen la variabilidad entre saltadores, pero éstas no son necesariamente las que mejor *discriminarán* entre los saltadores que presentan un

mejor rendimiento. Puede suceder, por ejemplo, que la máxima variabilidad se presente en la forma en la que se realiza el primer paso sin que ésta variable tenga un impacto decisivo en la altura final superada por el saltador y por tanto en el hecho de alcanzar el podio.

Descripción de la técnica

El método LDA parte de un conjunto de datos, al que llamaremos *conjunto de entrenamiento*, del que tenemos un conocimiento previo de la asignación de cada una de las observaciones a un grupo. En el ejemplo de los saltadores de altura, el conjunto de entrenamiento consistiría en un conjunto de m observaciones de n variables biomecánicas durante una ejecución del salto. Esto nos permite construir un conjunto n–dimensional de datos (n variables, m observaciones) a los que se les ha asignado una etiqueta de pertenencia a uno de los dos grupos (p. ej., etiquetas que identifican a los m saltadores como del grupo 1 (podio) o del grupo 2 (fuera del podio)).

Como en numerosas técnicas de inteligencia artificial, LDA se compone de dos fases: una primera de *entrenamiento* y una segunda de *predicción*. En la fase de entrenamiento se parte de los m datos del conjunto de entrenamiento y se genera un modelo lineal de las n variables que permita discriminar entre los dos grupos. En la fase de predicción, el modelo lineal se utiliza para predecir a qué grupo pertenece una nueva observación que no haya sido utilizada durante la fase de entrenamiento.

En el ejemplo anterior, la predicción nos permitiría estimar si un nuevo saltador será candidato al podio (pertenece al grupo 1) o no (grupo 2) a partir de una medición de sus n variables biomecánicas.

La formulación matemática de la técnica LDA pasa por modelizar probabilísticamente la distribución de datos en cada una de las clases. Nosotros nos limitamos al caso de dos clases a las que nos referiremos como α_1,α_2. Durante la fase de entrenamiento, los datos de entrenamiento de cada clase se utilizan para hacer una estimación estadística de los parámetros del modelo (media, varianza). El proceso de entrenamiento consiste por lo tanto en un proceso de *estimación estadística*, en el que se determinan parámetros de una cierta densidad de probabilidad.

La versión más simple del análisis de discriminantes asume que los datos de las clases α_1,α_2 están repartidos siguiendo una distribución de probabilidad normal. Eso se expresa mediante la densidad de probabilidad condicionada de que un dato $\mathbf{x}$ pertenezca a la clase α_i

$$p(\mathbf{x}|\alpha_i) = \frac{1}{(2\pi)^{N/2}\sigma_i} \exp\{-\frac{1}{2\sigma_i^2}(\mathbf{x}-\mathbf{m}_i)^T \cdot (\mathbf{x}-\mathbf{m}_i)\}, \quad (42)$$

donde N es la dimensionalidad de los datos (número de variables), y μ_i y σ_i son la media y desviación típica de los datos pertenecientes a la clase α_i.

Una vez modelizados los datos, se introduce una *función discriminante* $g(\mathbf{x})$ que permite asignar el dato $\mathbf{x}$ a una de las dos clases. La asignación de un nuevo dato $\mathbf{x}_0$ se realiza de la forma siguiente: si $g(\mathbf{x}_0) > 0$, el dato $\mathbf{x}_0$ se asigna a la clase α_1, mientras que si $g(\mathbf{x}_0) < 0$ será asignado al grupo α_2. En caso de que la función discriminante sea cero $g(\mathbf{x}_0) = 0$, el valor $\mathbf{x}_0$ se encontraría en la frontera de decisión y será igualmente asignable a cualquiera de las clases.

La función discriminante $g(\mathbf{x})$ permite definir la frontera de decisión de los datos para asignarlos a una clase. En el caso en que tengamos dos clases a discriminar, la función discriminante vendría dada por

$$g(\mathbf{x}) = g_1(\mathbf{x}) - g_2(\mathbf{x}), \quad (43)$$

donde $g_i(\mathbf{x})$ es la función de asignación a la clase i, que viene dada por la expresión

$$g_i(\mathbf{x}) = \ln p(\mathbf{x}|\alpha_i) + \ln P(\alpha_i),\ i = 1,2 \quad (44)$$

En la ecuación anterior, $p(\mathbf{x}|\alpha_i)$ es la densidad de probabilidad condicionada de que un dato $\mathbf{x}$ pertenezca a la clase α_i descrita en la ecuación 42, y $P(\alpha_i)$ es la *densidad de probabilidad a priori* a la que hacíamos referencia anteriormente. $P(\alpha_i)$ es por tanto la probabilidad con la que a priori asignaríamos un dato a la clase α_i. Sustituyendo la expresión 42 en la ecuación 44 e introduciendo ésta a su vez en 43 es fácil ver que la función discriminante de la ecuación viene dada por

$$g(\mathbf{x}) = -\frac{1}{\sigma_1^2}(\mathbf{x}-\mathbf{m}_1)^T \cdot (\mathbf{x}-\mathbf{m}_1) + \frac{1}{\sigma_2^2}(\mathbf{x}-\mathbf{m}_2)^T \cdot (\mathbf{x}-\mathbf{m}_2) - N\ln\frac{\sigma_1}{\sigma_2} + \ln P(\alpha_1) - \ln P(\alpha_2). \quad (45)$$

En el caso en el que no haya información a priori sobre la pertenencia a una de las dos clases, tendremos $P(\alpha_1) = P(\alpha_2) = \frac{1}{2}$ y la función discriminante se simplifica adoptando la forma

$$g(\mathbf{x}) = -\frac{1}{\sigma_1^2}(\mathbf{x}-\mathbf{m}_1)^T \cdot (\mathbf{x}-\mathbf{m}_1) + \frac{1}{\sigma_2^2}(\mathbf{x}-\mathbf{m}_2)^T \cdot (\mathbf{x}-\mathbf{m}_2) - N\ln\frac{\sigma_1}{\sigma_2}. \quad (46)$$

La ecuación 46 puede interpretarse de forma geométrica: la asignación del dato $\mathbf{x}$ a una u otra clase pasa por calcular a qué distancia se encuentra el dato $\mathbf{x}$ del centroide de cada clase i, que viene dada por

$$d_i(\mathbf{x}) = \frac{1}{\sigma_i^2}(\mathbf{x} - \mathbf{m}_i)^T \cdot (\mathbf{x} - \mathbf{m}_i). \quad (47)$$

La distancia $d_i(\mathbf{x})$ expresada en la ecuación anterior es conocida como *distancia de Mahalanobis*, y constituye una métrica que pondera la distancia del dato al centroide de la clase i $\mathbf{m}_i$ utilizando la varianza de los datos de la clase i, σ_i^2. La función discriminante $g(\mathbf{x})$ está compuesta por las distancias de Mahalanobis del dato a cada una de las clases, e incluye una corrección debida a la posible disparidad entre las varianzas de los datos de cada clase $-N \ln \frac{\sigma_1}{\sigma_2}$. Así, para asignar un dato $\mathbf{x}_0$ a la clase α_1, deberá cumplirse $g(\mathbf{x}_0) > 0$ o lo que es lo mismo, que $g_1(\mathbf{x}_0) > g_2(\mathbf{x}_0)$, y por lo tanto

$$\frac{1}{\sigma_1^2}(\mathbf{x}_0 - \mathbf{m}_1)^T \cdot (\mathbf{x}_0 - \mathbf{m}_1) + N \ln \sigma_1 < \frac{1}{\sigma_2^2}(\mathbf{x}_0 - \mathbf{m}_2)^T \cdot (\mathbf{x}_0 - \mathbf{m}_2) + N \ln \sigma_2. \quad (48)$$

Según la ecuación 48, cuando las varianzas de ambos grupos sean iguales $\sigma_1 = \sigma_2$, el discriminador lineal LDA asignará el dato $\mathbf{x}_0$ a la clase cuyo centroide esté más cerca del dato en distancia Mahalanobis. En este caso, la frontera de decisión será equidistante en distancia Mahalanobis a los centroides de los dos grupos. En caso de que exista una discrepancia entre las dos desviaciones típicas (p. ej., $\sigma_1 \neq \sigma_2$), el clasificador penalizará la asignación al grupo que presente mayor dispersión estadística mediante el término $N \ln \sigma_i$, $i = 1,2$, lo que ocasionará un desplazamiento efectivo de la frontera de decisión del problema. Además de la interpretación geométrica de la asignación de pertenencia a clases, el método puede entenderse desde un punto de vista estadístico ya que el nuevo dato es asignado al clúster al que su pertenencia es más *estadísticamente verosímil*.

Función de verosimilitud

Otras muchas técnicas estadísticas de reconocimiento de patrones utilizan la *función de verosimilitud* para definir los criterios de agrupamiento.

Ejemplo: separación de datos bidimensionales

Las dos fases de la técnica LDA se describen en el ejemplo descrito en el código 3.18. Se parte de un conjunto de entrenamiento con datos gaussianos ($n = 2$ variables, $m = 100$ observaciones) que han sido etiquetados como pertenecientes al grupo 1 o al grupo 2. Los datos etiquetados como grupo 1 tienen la misma desviación típica que los del grupo 2 ($\sigma_1 = \sigma_2 = 5{,}5$), pero presentan una media distinta ($\mu_1 = 10$, $\mu_2 = -10$). El conjunto completo de datos, junto a las etiquetas de pertenencia a cada grupo, se le pasan a la rutina LDA para que construya un modelo lineal que permita discriminar entre ambas clases. En este ejemplo utilizaremos el código LDA del paquete Python de inteligencia artificial *Scikit-Learn* que podemos obtener de http://scikit-learn.sourceforge.net. Este paquete utiliza a su vez los paquetes estándar de computación y representación gráfica de Python NumPy, SciPy, Matplotlib. La fase de entrenamiento corresponde a las líneas 26-28, en las que se genera un modelo lineal (fit) a partir de los datos *XT* y sus respectivas etiquetas de pertenencia a los dos gru-

pos, almacenadas en la variable *labelT*. La opción *priors = None* de la rutina LDA especifica que no se considera una mayor probabilidad de asignación *a priori* del nuevo dato a una de las dos clases. En ocasiones, un conocimiento previo de los datos nos permite favorecer la asignación de un nuevo dato a uno de los grupos. Este sesgo anticipado se expresa mediante la *densidad de probabilidad a-priori*, que determina una cierta probabilidad de asignación a los grupos antes de conocer el nuevo dato a clasificar.

Código 3.18: ejemplo de aplicación del análisis de discriminantes lineales (LDA)

```
# -*- coding: utf-8 -*-
import numpy as np
import pylab
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis

# Datos gaussianos:
mu1, sigma1 = 10, 5.5
X1 = mu1 + sigma1*np.random.randn(100,2)
mu2, sigma2 = -10, 5.5
X2 = mu2 + sigma2*np.random.randn(100,2)

# Representar graficamente los datos de entrenamiento:
fig1 = pylab.figure()
pylab.scatter(X1[:,0],X1[:,1],marker ='^',c='r')
pylab.scatter(X2[:,0],X2[:,1],marker ='o',c='b',hold='on')
pylab.legend(('Grupo 1', 'Grupo 2'))

# Concatenar los dos conjuntos de puntos:
XT = np.concatenate((X1,X2))

# Etiquetar los datos como tipo 1 o tipo 2:
label1 = np.ones(X1.shape[0])
label2 = 2*np.ones(X2.shape[0])
labelT = np.concatenate((label1,label2))

# Fase de entrenamiento:
clf = LinearDiscriminantAnalysis(n_components=2,priors=None)
clf.fit(XT, labelT)

#Fase de prediccion:
print clf.predict([[20, 0]]) #prediccion para el dato [20,0]
print clf.predict([[5, -20]])#prediccion para el dato [5,-20]

# Representacion de la prediccion de los datos [20,0] y [5,-20]:
pylab.scatter(20,0,s=100,marker ='x',c='k')
pylab.annotate('LDA grupo 1',xy=(20,-2),xycoords='data',
               xytext=(-50,-50),
               textcoords='offset points',
               arrowprops=dict(arrowstyle="->"))
pylab.scatter(-15,-5,s=100,marker ='x',c='k')
pylab.annotate('LDA grupo 2',xy=(-15,-5),xycoords='data',
               xytext=(-50,50),
               textcoords='offset points',
               arrowprops=dict(arrowstyle="->"))

# Prediccion de datos en una reticula:
fig2 = pylab.figure()
for i in range(-20,21,1):
    for k in range(-20,21,1):
        p = clf.predict([[i, k]])
        print i,k,p
        if p == 1:
            pylab.scatter(i,k,s=20,marker='o',c='r',hold='on')
        else:
            pylab.scatter(i,k,s=20,marker = 'o',c='b',hold='on')

```

```
57 pylab.axis([-20,20,-20,20])
58 pylab.text(5,5,'GRUPO 1',fontsize=25,fontweight='bold',color='k')
59 pylab.text(-10,-10,'GRUPO 2',fontsize=25,fontweight='bold',color='k')
60
61 pylab.show()
```

La fase de predicción permite utilizar el modelo lineal construido tras el entrenamiento para asignar un nuevo dato a uno de los dos grupos. En el código de ejemplo, los nuevos datos son (20,0) y (5, – 20), y son asignados por el clasificador lineal LDA a los grupos 1 y 2, respectivamente (líneas 31 y 32). El resultado de dicha asignación se representa en la figura 17, en la que los datos de entrenamiento son los círculos (etiquetados como grupo 1) y los triángulos (grupo 2). Los dos nuevos datos (20,0) y (5, – 20) para los que se realiza la predicción se indican en la figura 17 con una cruz, y el método LDA los ha asignado correctamente a los grupos 1 y 2, respectivamente. La frontera de decisión que determina la asignación de los datos a cada grupo puede obtenerse calculando la predicción LDA de los puntos de una retícula de tamaño (–20,20) en ambas variables.

Figura 17. Representación gráfica de la fase de entrenamiento del ejemplo LDA 3.18

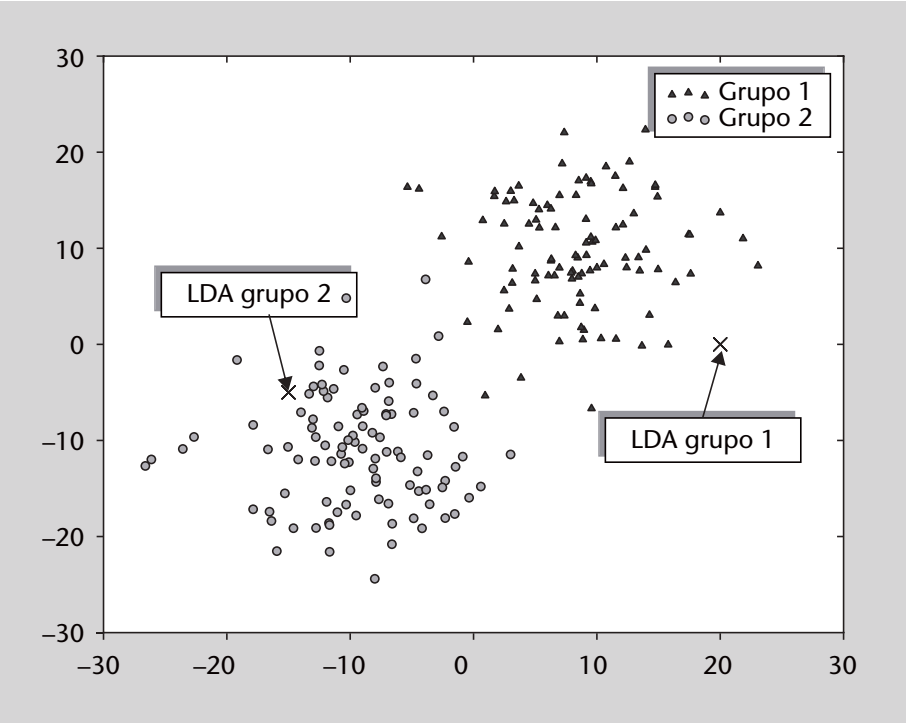


Figura 17

Los círculos azules han sido etiquetados como grupo 1 y los triángulos rojos como grupo 2 (datos de entrenamiento). Tras el entrenamiento, los nuevos datos [20,0] y [5, – 20] son correctamente asignados por el clasificador lineal LDA a los grupos 1 y 2, respectivamente.

En la figura 18, los puntos de la retícula que han sido asignados a los grupos 1 y 2 se representan con círculos de diferentes tonalidades. Como cabe esperar, la frontera es una función lineal que separa de forma equidistante los centroides de ambos grupos. Como veremos más adelante, la frontera es equidistante debido a que la desviación típica de los datos de cada uno de los grupos es la misma (i.e, $\sigma_1 = \sigma_2$).

Figura 18

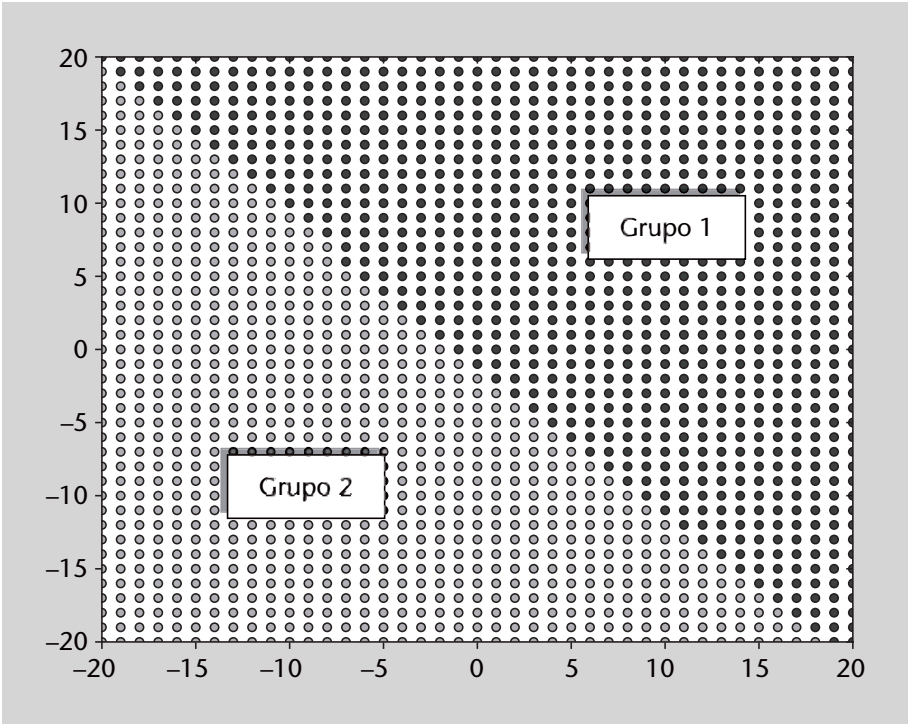


Figura 18

Predicción de la asignación a las dos clases mediante el modelo lineal generado por la técnica LDA a partir de los datos de entrenamiento en el ejemplo 3.18.

Conviene aclarar que, en muchos casos, técnicas como PCA no son útiles para realizar discriminación de datos en clases. Como ejemplo, consideremos los datos gaussianos multivariados de los dos grupos de la figura 19. Un análisis PCA de los datos de ambas clases proporcionaría una componente principal PCA a lo largo del eje de ordenadas (y), puesto que esta es la dirección, los datos presentan una mayor variabilidad. Si observamos la distribución de los datos de ambas clases, resulta evidente que una variable de este tipo no resulta útil para poder diferenciar los datos de ambos grupos, puesto que las diferencias entre el grupo 1 y el 2 es su ubicación en diferentes posiciones del eje de abcisas (x). En cambio, la línea vertical discontinua corresponde a la función discriminante obtenida al aplicar LDA a los datos, que sí resulta eficaz para separar de forma automática los datos en dos grupos que en gran medida coinciden con los grupos definidos inicialmente. En este caso, todos los datos del grupo 1 serían correctamente clasificados por la función discriminante, mientras que sólo tres puntos del grupo 2 serían asignados erróneamente al grupo 1.

3.3. Visualización de datos mutidimensionales

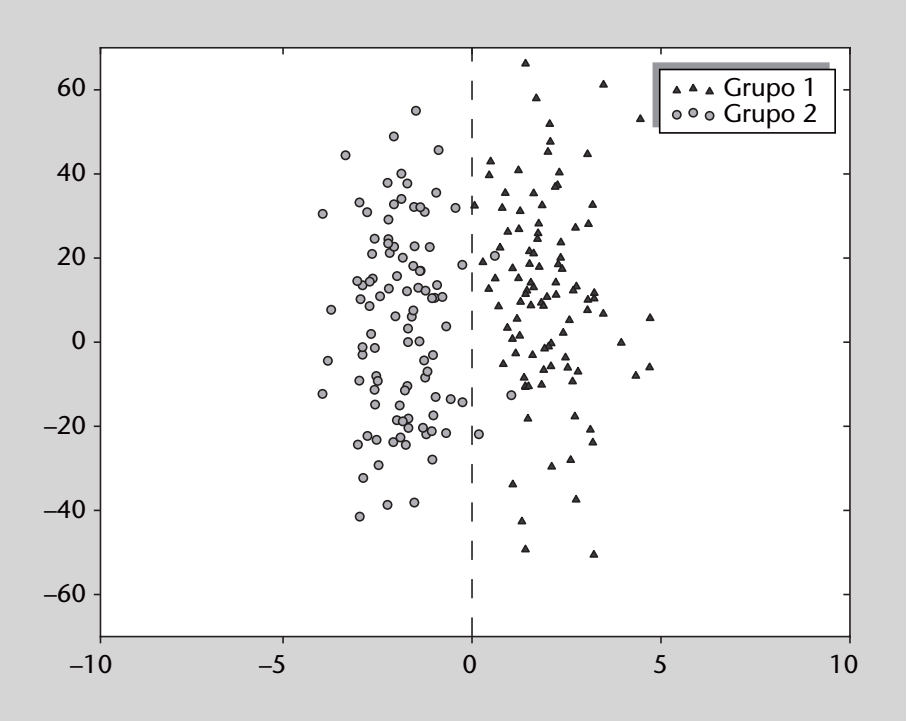
3.3.1. Escalamiento multidimensional (MDS)

Ejemplo de aplicación

En ocasiones deseamos visualizar datos multidimensionales en un plano. Imaginemos por ejemplo un conjunto de datos que contenga múltiples variables

sobre las preferencias musicales de un conjunto de personas. Las preferencias individuales de cada persona se representan mediante un conjunto de datos que define numerosos aspectos mediante *n* variables de carácter musical (grupos favoritos, el estilo de música, compositores, solistas, orquestas, grabaciones, etc.) y sociológico (edad, género, formación, salario, etc.). A las discográficas les resultará útil saber si existen relaciones entre los consumidores habituales de sus productos y otros colectivos desconocidos con los que puedan establecerse conexiones.

Figura 19. Comparación entre LDA y PCA



Resulta evidente que para establecer estas relaciones de forma manual no es posible trabajar en el espacio de dimensión *n*. Una de las posibilidades es proyectar los datos del espacio original a un espacio de dimensión 2 (plano), pero de forma que se *mantengan las distancias relativas entre cada uno de los individuos*. En otras palabras, personas que estaban lejos en el espacio original de dimensión *n* también estarán lejos en la proyección bidimensional de los datos, mientras que personas cercanas en el espacio original lo seguirán siendo en el espacio proyectado. Si las distancias entre individuos se conservan en el plano, será muy fácil ver qué personas están cercanas en el espacio original y por tanto quiénes son susceptibles de adquirir un determinado producto al que podrían ser afines. Podría suceder, por ejemplo, que los amantes incondicionales del rock sinfónico y los Rolling Stones, mayores de 40 años y con un alto nivel de estudios, sean un buen objetivo de mercado para una grabación inédita de una sonata de F. Schubert del pianista ruso S. Richter. Este tipo de relaciones permiten establecer nuevos nichos de mercado o explorar nuevas posibilidades en un estudio de marketing.

Descripción de la técnica

Como hemos indicado, el escalamiento multidimensional* se utiliza principalmente como herramienta de visualización de datos. En general, un análisis exploratorio previo es enormemente útil para establecer relaciones iniciales entre las variables, y por tanto, para obtener información cualitativa sobre las características estructurales de los datos. Cuando el conjunto de datos multidimensionales tiene un elevado número de variables, el análisis exploratorio se complica debido al número de variables a considerar. Por supuesto, siempre es posible visualizar los datos mediante proyecciones bidimensionales o tridimensionales en subespacios de variables representativas, pero el procedimiento no deja de ser laborioso y dificulta una visión general de los datos. MDS es una herramienta que permite proyectar los datos en un espacio de dimensionalidad reducida de forma que la distribución de los datos en el espacio proyectado mantenga una similitud con la distribución de los datos en el espacio original.

*En inglés *multidimensional scaling* (MDS).

En resumen, MDS busca un espacio de dimensionalidad reducida en el que se mantengan las distancias relativas entre los datos originales. Puntos cercanos en el espacio original son escalados de forma que también sean cercanos en el espacio MDS, lo que nos va a permitir una mejor visualización de los datos sin desvirtuar la estructura original de los mismos.

La estructura general del algoritmo MDS es la siguiente: en primer lugar se toman los datos en el espacio N-dimensional original (N variables, M observaciones) y se calcula la *matriz de distancias* entre cada una de las M observaciones. Esto da lugar a una matriz simétrica $M \times M$, cuyo componente i,j corresponde a la distancia euclídea entre los puntos $\mathbf{x}_i$ e $\mathbf{x}_j$. A continuación, las M observaciones son distribuidas aleatoriamente en un espacio bidimensional. Para cada par de puntos, se calcula una función error como la diferencia entre la distancia entre los puntos en el espacio original y la distancia en el espacio proyectado. Entonces se aplica un proceso iterativo, en el que la posición de cada punto en el espacio 2D es corregida para minimizar el error entre la distancia en el espacio original. El proceso continúa hasta que el error global no se ve afectado de forma sustancial por nuevos desplazamientos de los nodos.

El escalamiento multidimensional (MDS) permite visualizar datos multidimensionales en un espacio bidimensional.

Ejemplo: Proyección 2D de datos multidimensionales. Utilización de diferentes métricas

Un ejemplo de utilización de la rutina MDS incluida en las librerías sklearn se describe en el código 3.19. En este ejemplo se generan 30 datos, 4 dimensionales, aleatoriamente distribuidos en tres grupos distintos de igual tamaño

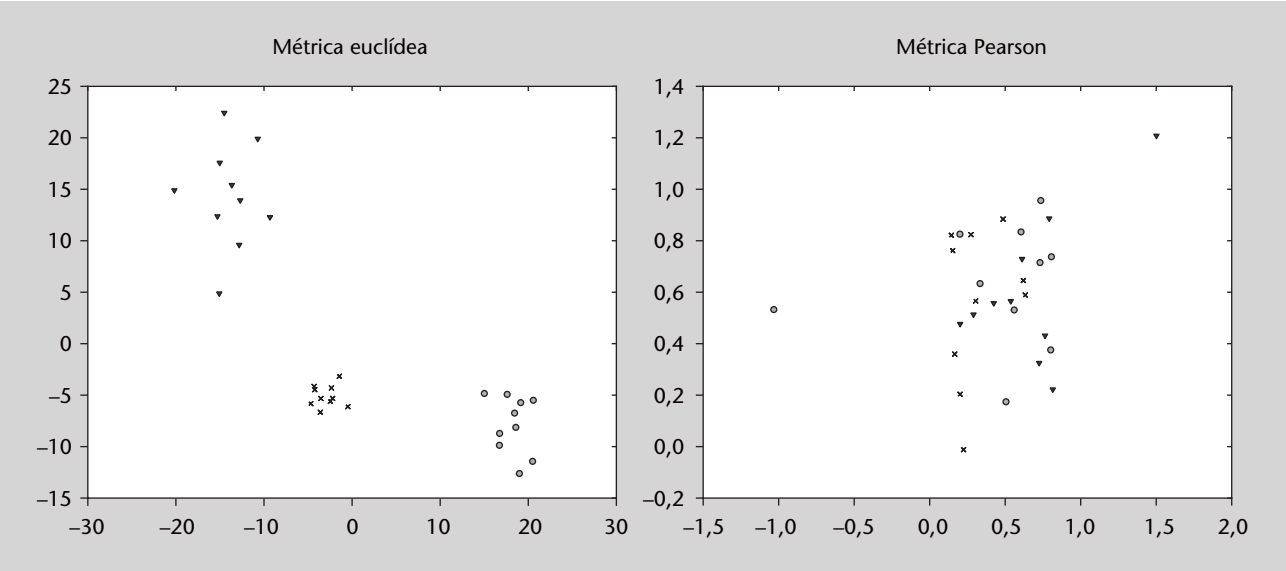
(líneas 1-15). A continuación se llama a la rutina MDS (línea 23), y previamente se define la métrica con la que deseamos trabajar para determinar la distancia entre puntos. En este ejemplo se ha escogido una distancia euclídea, ya que en el espacio original los datos no presentan ninguna correlación. La opción de distancia de Pearson resulta interesante en casos en los que los datos no se distribuyen por grupos separados en el espacio sino por la presencia de correlaciones internas entre grupos de datos.

Código 3.19: MDS con datos descorrelacionados distribuidos en un espacio

```
1  # -*- coding: utf-8 -*-
2  import numpy as np
3  import pylab as py
4
5  from sklearn import manifold
6  from sklearn.metrics import euclidean_distances
7
8  seed = np.random.RandomState(seed=3)
9
10 # Matriz de datos (N variables x M observaciones):
11 N = 4
12 M = 30
13
14 # Generar datos a partir de 3 clusteres diferentes:
15
16 X1 = 10 + 2*np.random.randn(M/3,N) # cluster 1 (dispersion media)
17 X2 = -10 + 5*np.random.randn(M/3,N) # cluster 2 (dispersion alta)
18 X3 = 1*np.random.randn(M/3,N) # cluster 3 (dispersion baja)
19
20 A = np.concatenate((X1,X2,X3))
21
22 # Tecnica MDS: Devuelve la posicion de los datos
23 # originales en un espacio 2D:
24 mds = manifold.MDS(n_components=2, max_iter=3000, eps=1e-9,
       random_state=seed,
25                    dissimilarity="precomputed", n_jobs=1)
26 similarities = euclidean_distances(A)
27 pos = mds.fit(similarities).embedding_
28
29 # Representacion de los datos en el espacio 2D MDS:
30 fig1 = py.figure()
31 for i in range(0,M/3,1):
32     py.scatter(pos[i][0],pos[i][1],marker='o',c='r')
33 for i in range(M/3,2*M/3,1):
34     py.scatter(pos[i][0],pos[i][1],marker='v',c='g')
35 for i in range(2*M/3,M,1):
36     py.scatter(pos[i][0],pos[i][1],marker='x',c='b')
37 fig1.suptitle('Metrica Euclidea')
38 py.show()
```

La proyección bidimensional MDS de los datos descritos en el ejemplo 3.19 utilizando una métrica euclídea se representan en la figura 20. Como puede observarse, los tres grupos definidos en el espacio original de cuatro dimensiones se mantienen agrupados en la proyección MDS 2D. Si utilizamos la distancia de correlación de Pearson, los resultados no son satisfactorios puesto que la distribución de los datos en el espacio proyectado no refleja la auténtica distribución de los mismos en el espacio original (panel derecho de la figura 20).

Figura 20. MDS ejemplo 3.19



El ejemplo 3.20 describe una aplicación de la técnica MDS a un conjunto de puntos formado por tres subconjuntos en los que las observaciones tienen una correlación interna. Cada uno de los grupos presenta distintos valores de correlación. En este caso, la distancia que resulta idónea es la distancia de Pearson.

Código 3.20: aplicación de MDS a un conjunto de datos que presentan correlaciones internas organizadas en tres grupos

```
1  from numpy import *
2  from scaledown import *
3  import pylab as py
4
5  # Matriz de datos (N variables x M observaciones):
6  N = 10
7  M = 30
8
9  # Generar datos a partir de 3 grupos con correlaciones diferentes:
10
11 Xinit1 = random.randn(1,N) # cluster 1 (correlacion baja)
12 X1 = Xinit1
13 for i in range(1,int(M/3),1):
14     X1 = concatenate((X1,100*Xinit1 + random.rand(1,N)))
15
16 Xinit2 = random.randn(1,N) # cluster 2 (correlacion media)
17 X2 = Xinit2
18 for i in range(1,int(M/3),1):
19     X2 = concatenate((X2,50*Xinit2 + random.rand(1,N)))
20
21 Xinit3 = random.randn(1,N) # cluster 3 (correlacion alta)
22 X3 = Xinit3
23 for i in range(1,int(M/3),1):
24     X3 = concatenate((X3,10*Xinit3 + random.rand(1,N)))
25
26 #Representacion correlaciones de cada grupo:
27 fig1 = py.figure()
28 py.scatter(X1[0],X1[1],marker='o',hold='on')
29 py.scatter(X2[0],X2[1],marker='x',hold='on')
30 py.scatter(X3[0],X3[1],marker='v',hold='on')
```

```
31
32 A = concatenate((X1,X2,X3))
33
34 # Escoger metrica de espacio original (Euclidea o Pearson):
35 metrica = euclidean
36 # metrica = pearson
37
38 # Tecnica MDS: devuelve la posicion de los datos
39 # originales en un espacio 2D:
40 mds = scaledown(A, metrica)
41
42 # Representacion de los datos en el espacio 2D MDS:
43 fig1 = py.figure()
44 for i in range(0,int(M/3),1):
45     py.scatter(mds[i][0],mds[i][1],marker='o',c='r')
46 for i in range(int(M/3),2*int(M/3),1):
47     py.scatter(mds[i][0],mds[i][1],marker='x',c='g')
48 for i in range(2*int(M/3),M,1):
49     py.scatter(mds[i][0],mds[i][1],marker='v',c='b')
50 fig1.suptitle('Metrica Euclidea')
51 py.show()
```

Las figuras 21 y 22 reflejan la distribución de los puntos en el espacio MDS bidimensional cuando se utiliza una métrica euclídea (figura 21) y una métrica de Pearson (figura 22). En este caso resulta evidente que la métrica de Pearson permite agrupar los tres subconjuntos de observaciones en tres grupos bien diferenciados en el espacio de dimensión reducida.

Figura 21. Proyección MDS 2D de los datos del ejemplo 3.20 utilizando una métrica euclídea

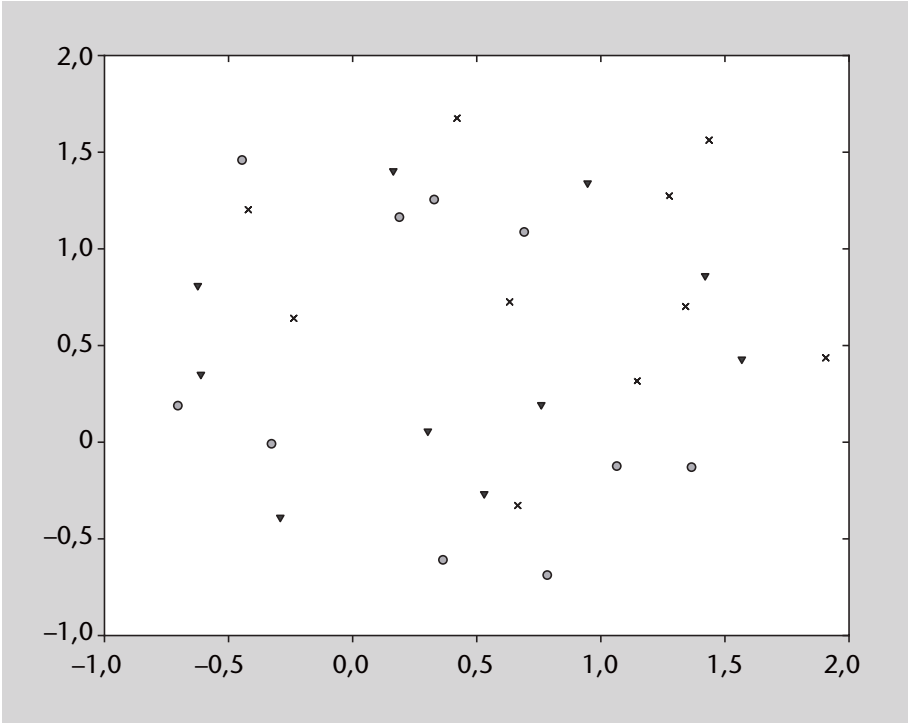
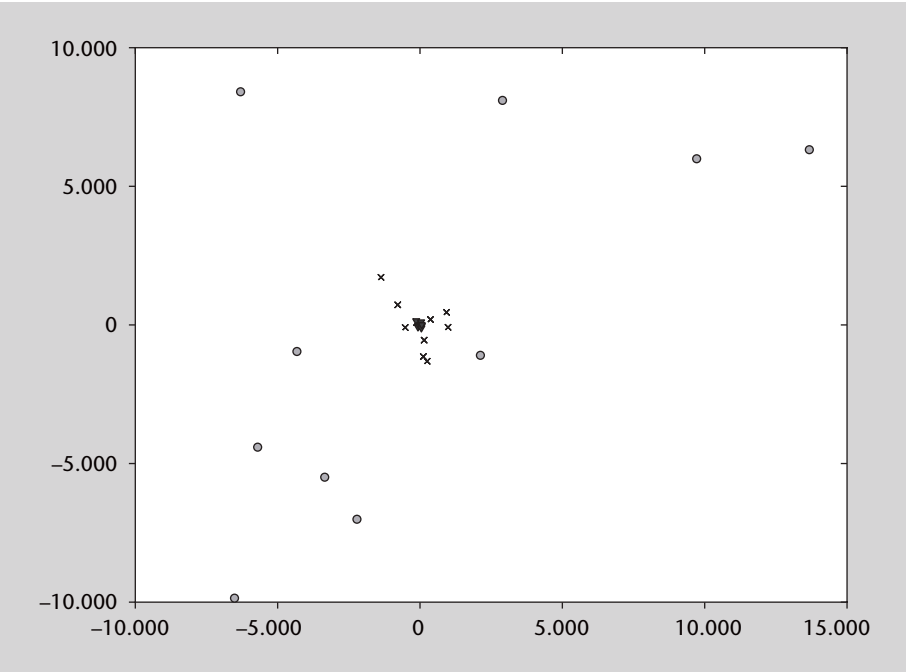


Figura 22. Proyección MDS 2D de los datos del ejemplo 3.20 utilizando una métrica de Pearson



4. Clasificación

4.1. Introducción

La clasificación es una de las tareas de reconocimiento de patrones en la que queremos etiquetar a un individuo a partir de ciertas propiedades que lo caracterizan; entendiendo como individuo una entidad de cualquier tipo. A continuación se muestran tres ejemplos de clasificación que utilizaremos a lo largo de este apartado:

- Clasificación de setas (fuente: problema «mushroom» del repositorio UCI (Frank y Asunción, 2010)). Supongamos que queremos realizar una aplicación para teléfonos móviles que ayude a los aficionados a la recolección de setas a discernir las setas venenosas de las comestibles a partir de sus propiedades: forma y color del sombrero y grosor y color del tronco.

- Clasificación de flores (fuente: problema «iris» del repositorio UCI (Frank y Asunción, 2010)). Supongamos que queremos realizar una aplicación para un almacén de flores, donde llegan a diario miles de productos. Disponemos de un sistema láser que nos suministra una serie de medidas sobre las flores y nos piden que el sistema las clasifique automáticamente para transportarlas mediante un robot a las diferentes estanterías del almacén. Las medidas que envia el sistema láser son: la longitud y anchura del sépalo y el pétalo de cada flor.

- Clasificación de documentos. Supongamos que queremos realizar una aplicación en la que dados los textos que van llegando a una agencia de noticias, los clasifique automáticamente para distribuirlos automáticamente a los clientes interesados. Los temas pueden ser: deporte, sociedad, política... o más específicamente: fútbol, motociclismo, atletismo... Los clientes pueden ser cadenas de televisión, radios, periódicos, revistas de toda índole...

Ved también

El problema de la clasificación de documentos, junto con sus alternativas, se describe con más profundidad en el subapartado 4.1.1.

La clasificación, desde el punto de vista del reconocimiento de patrones, involucra tres fases importantes: la definición de las clases, la representación de la información en forma de atributos y el aprendizaje mediante algoritmos.

La definición de clases viene dada en muchos problemas. En el ejemplo de la clasificación de setas tenemos que las clases son: venenosa y comestible. En

los otros dos ejemplos tenemos que tener en cuenta la granularidad de las clases. En la clasificación de flores podemos escoger entre familias de flores: rosa, clavel, margarita... o dentro de una familia, la flor específica: iris versicolor, iris setosa e iris virgínica. Este último será el que utilizaremos durante todo el apartado. En la clasificación de documentos también nos afecta la granularidad. En este caso, además, nos puede ocurrir que un documento tenga más de una etiqueta: una noticia sobre la relación sentimental entre un futbolista y una cantante puede ser de deporte y sociedad... La definición de clases tiene una relación directa con la aplicación final de la clasificación.

La representación de la información tiene que ver con como representamos las características de los individuos o ejemplos a etiquetar. Tiene una relación directa con el comportamiento general del clasificador que construyamos; es tan o más importante que el algoritmo de aprendizaje que utilicemos. Este tema se explica con más detalle en el apartado 3.

Este apartado está dedicado a la tercera de las fases, los algoritmos de aprendizaje para clasificación. Empezamos por desarrollar un poco más el problema de la clasificación de documentos (o categorización de textos) y por formalizar el problema. Una vez hecho esto damos una tipología de este tipo de algoritmos y entramos a describirlos. Para finalizar se dan los protocolos y medidas de evaluación.

4.1.1. Categorización de textos

La categorización de textos es uno de los problemas que se intenta resolver en el campo del procesamiento del lenguaje natural*. Este problema consiste en clasificar documentos con una o varias clases.

*** En inglés, *Text Categorisation* y *Natural Language Processing*.**

Un ejemplo práctico de este problema, al que se están dedicando muchos recursos y esfuerzos, es el tratamiento de textos por parte de las agencias de noticias. A estas empresas les suelen llegar varias noticias por segundo y tienen que etiquetar estas noticas en función de la temática: deportes, política, política internacional, sociedad... Una de las características de este problema es que es multietiqueta*, es decir, un ejemplo puede estar asociado a más de una etiqueta.

*** En inglés, *Multilabel*.**

Para representar los documentos en forma de atributos se suelen utilizar conjuntos de palabras*. Esta técnica viene del campo de la recuperación de la información**. Los conjuntos de palabras suelen aparecer en la bibliografía de dos formas: como conjuntos de palabras propiamente dichos (sobre todo cuando se trabaja a nivel de frase) o como conjuntos de palabras anotando el número de veces que aparecen en el documento (o su frecuencia de aparición).

*** En inglés, *bag of words* o *vector space model (VSM)*.**

**** En inglés, *Information Retrieval*.**

Cuando se trabaja con conjuntos de atributos, se suelen procesar los datos para eliminar las palabras sin contenido semántico (como preposiciones, ar-

tículos...), para eliminar signos de puntuación y para trabajar directamente con los lemas de las palabras (se elimina el género, número y tiempo verbal). Para poder realizar esto, se utilizan dos tipos de recursos: listas de palabras sin contenido semántico* y procesadores lingüísticos.

* En inglés, *lists of stop words*.

Enlace de interés

En la página web del profesor Lluís Padró (http://www.lsi.upc.edu/~padro) se pueden encontrar listas de palabras sin contenido semántico para el inglés, el castellano y el catalán. Las tenéis disponibles en el repositorio de la asignatura.

El segundo tipo de recurso son los procesadores lingüísticos. Suelen incluir tokenizadores, lematizadores, etiquetadores morfosintácticos, analizadores sintácticos superficiales... para diferentes lenguas. Para este problema, con el tokenizador y el lematizador ya cumpliremos con nuestras necesidades. Hay diversos procesadores lingüísticos libres disponibles en la red. Dos de ellos son el *FreeLing** y el *NLTK***.

* http://nlp.lsi.upc.es/freeling

** http://www.nltk.org

Un caso particular de la categorización de textos son los filtros anti spam. En este problema hay que pensar en cómo añadir como atributos los componentes de los correos, así como las direcciones de origen o el asunto.

El conjunto de datos más utilizado en categorización de textos es el Reuters-21578*. A los conjuntos de datos de textos se les denomina corpus. Este corpus ha sido generado a partir de datos de la agencia de noticias Reuters. Los documentos que contiene están sin procesar y tienen asociados una serie de clases o categorías. Este conjunto de datos también está disponible en el repositorio de la UCI (Frank y Asunción, 2010).

* http://bit.ly/1F8AFcO

El siguiente corpus* es un subconjunto del anterior. T. Joachims procesó una parte del corpus anterior calculando las frecuencias de aparición de las diferentes palabras en el documento. A esta forma de representar la información se le denomina VSM**. La tarea asociada a este corpus es distinguir si los documentos pertenecen o no a la clase: *adquisiciones corporativas*. El corpus está pensado para discernir si un documento es de esta clase o no. El conjunto de datos contiene 1.000 ejemplos positivos y 1.000 negativos en el conjunto de entrenamiento y 300 de cada en el conjunto de test. El número de atributos asciende a 9.947 y contienen información de la frecuencia relativa de las palabras en los documentos.

* http://svmlight.joachims.org

** Del inglés, *Vector Space Model*. También suele recibir el nombre de bolsa de palabras (del inglés, *bag of words*).

El tercer conjunto de datos es el *TechTC-100**, que ha sido recopilado por Gabrilovich y Markovitch. Este conjunto consta de una serie de páginas web que están asociadas a una ontología o jerarquía. Los datos están disponibles sin procesar y procesados. Este proceso consiste en generar el VSM para los documentos guardando el número de veces que aparece un término o palabra en cada documento. Las clases son las ramas de la jerarquía ODP** a la que pertenecen las páginas.

* http://techtc.cs.technion.ac.il/techtc100/techtc100.html

** http://www.dmoz.org

4.1.2. Aprendizaje automático para clasificación

En este subapartado vamos a ver la formalización del problema de clasificación tal y como se usa comúnmente en la bibliografía del campo del aprendizaje automático y el reconocimiento de patrones.

El objetivo del aprendizaje automático* para clasificación (o aprendizaje supervisado) consiste en inducir una aproximación (modelo o hipotesis) h de una función desconocida f definida desde un espacio de entrada X hacia un espacio discreto y desordenado $Y = 1,\ldots,K$, dado un conjunto de entrenamiento S.

* En inglés, *machine learning*.

Regresión

Cuando el espacio de salida es continuo, el problema a tratar será de *regresión* en lugar de clasificación.

El conjunto de entrenamiento $S = \{(x_1,y_1),\ldots,(x_n,y_n)\}$ contiene n ejemplos de entrenamiento, que corresponden a pares (x,y) donde $x \in X$ e $y = f(x)$. La componente x de cada ejemplo es un vector $x = (x_1,\ldots,x_n)$ de atributos con valores continuos o discretos que describen la información relevante o propiedades del ejemplo. Los valores del espacio de salida Y asociados a cada ejemplo son las clases del problema. Así, cada ejemplo de entrenamiento queda totalmente caracterizado por un conjunto de pares atributo–valor y una etiqueta de clase.

Clustering

Cuando no utilizamos las etiquetas de clase y_i en el conjunto de entrenamiento $S = \{x_1,\ldots,x_n\}$ hablamos de *clustering* o aprendizaje no supervisado. Este tema se describe en el subapartado 2.3 de este módulo.

Dado un conjunto de entrenamiento S, un algoritmo de entrenamiento induce un clasificador h que corresponde a una hipótesis acerca de la función f. Al hacer esto, el algoritmo puede escoger entre un conjunto de posibles funciones H llamado *espacio de hipotésis*. Los diferentes algoritmos de aprendizaje difieren en el espacio de hipotésis que tienen en cuenta (algunos tratarán funciones lineales, otros crearán hiperplanos que particionan dominios...); en el lenguaje de representación que utilizan (por ejemplo: árboles de decisión, conjuntos de probabilidades condicionales, redes neuronales...) y en el *sesgo* que usan al escoger la "mejor" hipotésis de entre las que son compatibles con el conjunto entrenamiento (por ejemplo: simplicidad, margen máximo...).

Partiendo de nuevos vectores x, utilizaremos h para predecir sus correspondientes valores de y para clasificar los nuevos ejemplos; esperando que coincidan con f en la mayoría de casos, o lo que es equivalente, que se produzcan el mínimo número de errores.

Lecturas complementarias

S. Kulkarni; G. Harman (2011). *An Elementary Introduction to Statistical Learning Theory*. USA: John Wiley and Sons, Inc.

La medida del error de los nuevos ejemplos (o ejemplos no vistos) se llama *error de generalización**. Resulta obvio que el error de generalización no se puede minimizar sin conocer a priori la función f o la distribución $P(X,Y)$. Esto hace que necesitemos un principio de inducción. La manera más usual de proceder consiste en intentar minimizar el error sobre el conjunto de entrenamiento (o error empírico). Este principio se conoce como *minimización empírica del riesgo*** y da una buena estimación del error de generalización en presencia de muchos ejemplos de entrenamiento.

* En inglés, *generalisation or true error*.

** En inglés, *empirical risk minimisation*.

No obstante, en dominios con pocos ejemplos de entrenamiento, forzar a que la minimización del error empírico tienda a cero puede llevar a que se sobreentrene* el conjunto de entrenamiento y a que se generalice de forma errónea. La complejidad de las funciones inducidas** y la presencia de ruido y ejemplos atípicos*** (mal clasificados o inconsistentes) en el conjunto de entrenamiento también puede aumentar el riesgo de sobreentrenamiento. Al definir un marco experimental es necesario establecer un compromiso entre el error experimental y la complejidad de los clasificadores inducidos para garantizar un error de generalización pequeño.

*** En inglés, *overfitting*.**

**** La complejidad del modelo se mide con la llamada dimensión de Vapnik-Chervonenkis.**

***** En inglés, *outliers*.**

Toda la nomenclatura utilizada en los dos párrafos anteriores viene de la *teoría del aprendizaje estadístico**. La forma clásica de tratar este tema es a partir del *sesgo*** y la varianza. El sesgo de un estimador corresponde a la diferencia entre su esperanza matemática y el valor que está estimando. Todo lo que restringe el espacio de búsqueda está relacionado con el sesgo. La complejidad del modelo se mide a través de la varianza, que mide la dispersión de los valores en torno a la esperanza. La complejidad del modelo está directamente relacionada con la varianza e inversamente relacionada con el sesgo. El error de generalización está formado por la suma del sesgo y la varianza del modelo. En la construcción de modelos siempre tenemos que tratar con un compromiso entre el sesgo y la varianza.

*** En inglés, *statistical learning theory*.**

**** En inglés, *bias*.**

A modo de ejemplo, la tabla 8 muestra las características de dos ejemplos del conjunto de datos de las setas descrito anteriormente. Los atributos del primer ejemplo $x = (x_1,x_2,x_3,x_4)$ son *cap-shape*, *cap-color*, *gill-size* y *gill-color* con valores *convex*, *brown*, *narrow* y *black* respectivamente; el valor de y es *poisonous*. El conjunto Y contiene las etiquetas {*poisonous*,*edible*}.

Tabla 8. Ejemplo del tratamiento de setas

class	cap-shape	cap-color	gill-size	gill-color
poisonous	convex	brown	narrow	black
edible	convex	yellow	broad	black

Fuente: problema «mushroom» del repositorio UCI (Frank y Asunción, 2010)

Toda la formulación que hemos visto hasta aquí corresponde a problemas monoetiqueta, en los que cada ejemplo sólo tiene asociada una única etiqueta. Para formalizar los problemas multietiqueta, como la categorización de textos, tenemos que cambiar las características de la función $f(x)$. En los problemas monoetiqueta esta función devuelve un único valor de $Y = \{1,\ldots,k\}$, donde k es el número de clases; en los problemas multietiqueta, la función $f(x)$ devuelve un conjunto de valores de Y, que puede ser vacío.

4.1.3. Tipología de algoritmos para clasificación

Podemos clasificar los algoritmos en función del principio de inducción que usan para adquirir los modelos o reglas de clasificación a partir de los ejemplos. Estos métodos están basados en: probabilidades, distancias, reglas o kernels. Esta tipología no tiene la intención de ser exhaustiva o única. Por su-

puesto, la combinación de diferentes paradigmas es otra posibilidad que se está utilizando en los últimos tiempos.

Los cuatro siguientes subapartados están enfocados a explicar estos paradigmas y algunos de sus algoritmos más característicos. El último subapartado nos presenta los protocolos de evaluación que se utilizan en los procesos de clasificación.

4.2. Métodos basados en modelos probabilísticos

Los métodos estadísticos suelen estimar un conjunto de parámetros probabilísticos, que expresan la probabilidad condicionada de cada clase dadas las propiedades de un ejemplo (descrito en forma de atributos). A partir de aquí, estos parámetros pueden ser combinados para asignar las clases que maximizan sus probabilidades a nuevos ejemplos.

4.2.1. Naive Bayes

Supongamos que queremos realizar una aplicación para teléfonos móviles que ayude a los aficionados a la recolección de setas a discernir las setas venenosas de las comestibles a partir de sus propiedades: forma y color del sombrero y grosor y color del tronco. La tabla 9 muestra ejemplos de este tipo de datos.

Tabla 9. Conjunto de entrenamiento

class	cap-shape	cap-color	gill-size	gill-color
poisonous	convex	brown	narrow	black
edible	convex	yellow	broad	black
edible	bell	white	broad	brown
poisonous	convex	white	narrow	brown
edible	convex	yellow	broad	brown
edible	bell	white	broad	brown
poisonous	convex	white	narrow	pink

Fuente: problema «mushroom» del repositorio UCI (Frank y Asunción, 2010)

El Naive Bayes es el representante más simple de los algoritmos basados en probabilidades. Está basado en el teorema de Bayes.

El algoritmo de Naive Bayes lo describen Duda y Hart en su forma más clásica en 1973.

El algoritmo de Naive Bayes clasifica nuevos ejemplos $x = (x_1, \ldots, x_m)$ asignándole la clase k que maximiza la probabilidad condicional de la clase dada la secuencia observada de atributos del ejemplo. Es decir,

$$\operatorname*{argmax}_k P(k|x_1, \ldots, x_m) = \operatorname*{argmax}_k \frac{P(x_1, \ldots, x_m|k)P(k)}{P(x_1, \ldots, x_m)} \approx \operatorname*{argmax}_k P(k) \prod_{i=1}^{m} P(x_i|k)$$

donde $P(k)$ y $P(x_i|k)$ se estiman a partir del conjunto de entrenamiento, utilizando las frecuencias relativas (estimación de la máxima verosimilitud*).

*** En inglés, *maximum likelihood estimation.***

Ejemplo de aplicación

La tabla 9 muestra un ejemplo de conjunto de entrenamiento en el que tenemos que detectar si una seta es venenosa o no en función de sus propiedades.

Durante el proceso de entrenamiento empezaremos por calcular $P(k)$ para cada una de las clases $k \in Y$. Así, aplicando una estimación de la máxima verosimilitud obtenemos $P(poisonous) = 3/7 = 0{,}43$ y $P(edible) = 4/7 = 0{,}57$. El segundo paso consiste en calcular $P(x_i|k)$ para cada pareja atributo-valor y para cada clase, de la misma forma que en el caso anterior. La tabla 10 muestra los resultados de este proceso. El 1 de la celda correspondiente a la fila *cap-shape: convex* y columna *poisonous* sale de que los tres ejemplos etiquetados como *poisonous* tienen el valor *convex* para el atributo *cap-shape*.

Tabla 10. Valores de $P(x_i|k)$

atributo-valor	poisonous	edible
cap-shape: convex	1	0,5
cap-shape: bell	0	0,5
cap-color: brown	0,33	0
cap-color: yellow	0	0,5
cap-color: white	0,67	0,5
gill-size: narrow	1	0
gill-size: broad	0	1
gill-color: black	0,33	0,25
gill-color: brown	0,33	0,75
gill-color: pink	0,33	0

Con esto habremos terminado el proceso de entrenamiento. A partir de aquí, si nos llega un ejemplo nuevo como el que muestra la tabla 11 tendremos que aplicar la fórmula $\text{argmax}_k P(k) \prod_{i=1}^{m} P(x_i|k)$ para clasificarlo.

Tabla 11. Ejemplo de test

class	cap-shape	cap-color	gill-size	gill-color
poisonous	convex	brown	narrow	black

Fuente: problema «mushroom» del repositorio UCI (Frank y Asunción, 2010)

Empezamos por calcular la parte del valor *poisonous*. Para ello tendremos que multiplicar entre sí las probabilidades: $P(poisonous)$, $P(cap-shape-convex|poisonous)$, $P(cap-color-brown|poisonous)$, $P(gill-size-narrow|poisonous)$ y $P(gill-color-black|poisonous)$; que equivale a un valor de 0,05. Realizando el mismo proceso para la clase *edible* obtenemos un valor de 0. Por tanto, la clase que maximiza la fórmula de las probabilidades es *poisonous*; con lo que el método está clasificando correctamente el ejemplo de test, dado este conjunto de entrenamiento.

Análisis del método

Uno de los problemas del Naive Bayes, que ha sido mencionado frecuentemente en la literatura, es que el algoritmo asume la independencia de los diferentes atributos que representan a un ejemplo.

A modo de ejemplo, es poco probable que el color del tallo de una seta no esté relacionado con el color de su sombrero.

Hasta ahora, hemos hablado exclusivamente de atributos nominales. En el caso de tener un problema representado con algún atributo continuo, como los numéricos, es necesario algún tipo de proceso. Hay diversas formas de hacerlo; una consiste en categorizarlos, dividiendo el continuo en intervalos. Otra opción consiste en asumir que los valores de cada clase siguen una distribución gaussiana y aplicar la fórmula:

$$P(x = v|c) = \frac{1}{\sqrt{2\pi\sigma_c^2}} \exp\left(-\frac{(v-\mu_c)^2}{2\sigma_c^2}\right)$$

donde x corresponde al atributo, v a su valor, c a la clase, μ_c al promedio de valores de la clase c y σ_c^2 a su desviación estándar.

El algoritmo presenta un problema numérico a tener en cuenta para ciertos conjuntos de datos. Cuando nos encontramos con un ejemplo de test que tiene alguna pareja atributo–valor que no ha aparecido en el conjunto de entrenamiento, no tendremos ningún valor para $P(x_i|k)$. La tabla 12 muestra un ejemplo para el conjunto de entrenamiento anterior; el valor "white" del atributo "gill–color" no aparece en el conjunto de entrenamiento. Para solucionar este problema se suele aplicar alguna técnica de *suavizado**.

* En inglés, *smoothing*.

Tabla 12. Ejemplo de test (suavizado)

class	cap-shape	cap-color	gill-size	gill-color
edible	bell	yellow	broad	white

Fuente: problema «mushroom» del repositorio UCI (Frank y Asunción, 2010)

Un ejemplo de técnica de suavizado para este algoritmo consiste en sustituir la $P(x_i|k)$ que contiene el contador a cero por $P(k)/n$ donde n corresponde al número de ejemplos de entrenamiento.

Otra característica a destacar es que cuando el conjunto de entrenamiento no está balanceado, tiende a clasificar los ejemplos hacia la clase que tiene más ejemplos dentro del conjunto de entrenamiento.

Conjunto de entrenamiento balanceado

Un conjunto de entrenamiento está balanceado cuando tiene el mismo número de ejemplos de cada una de las clases que contiene.

Lectura complementaria

Esta técnica de suavizado aparece en: H. T. Ng. Exemplar–based Word Sense Disambiguation: Some Recent Improvements. En las actas de la conferencia *Empirical Methods in Natural Language Processing, EMNLP*, 1997.

Dos de las ventajas que presenta el método son su simplicidad y eficiencia computacional.

No obstante, y a pesar de sus inconvenientes, este método ha sido muy utilizado históricamente y se han obtenido buenos resultados para muchos conjuntos de datos. Esto ocurre cuando el conjunto de entrenamiento representa bien las distribuciones de probabilidad del problema.

Implementación en Python

El programa 4.1 corresponde a la implementación, utilizando los módulos de la librería scikit-learn de Python, del algoritmo Naive Bayes.

Código 4.1: Naive Bayes en Python

```
1  import numpy as np
2  from sklearn.naive_bayes import GaussianNB
3  from sklearn.model_selection import train_test_split
4  from sklearn import preprocessing
5
6  le = preprocessing.LabelEncoder()
7
8  with open("mushroom.data.txt","r") as fopen:
9      lines = [l.strip().split(',') for l in fopen]
10
11 attrs = np.array([l[1:] for l in lines]).flatten()
12 le.fit(attrs)
13 encoded_attrs = le.transform(attrs)
14
15 X = np.reshape(encoded_attrs,(len(lines),-1))
16
17 y_labels = np.array([l[0] for l in lines])
18 le.fit(y_labels)
19 y = le.transform(y_labels)
20
21 X_train, X_test, y_train, y_test = train_test_split(X, y,
22                                                     test_size=0.33)
23
24 clf = GaussianNB()
25 clf.fit(X_train, y_train)
26
27 print("PRECISION:",clf.score(X_test,y_test))
```

Como se puede observar, para este ejemplo utilizamos el módulo *GaussianNB* de *sklearn.naive_bayes**, así como el archivo de datos del problema «mushroom» del UCI (Frank y Asunción, 2010). Este programa selecciona aleatoriamente como conjunto de tests el 33 % de ejemplos del archivo mediante el módulo *train_test_split*. El resto de ejemplos pertenecerá al conjunto de entrenamiento. También saca provecho del módulo *preprocessing* que, mediante el método *LabelEncoder()*, nos permite recodificar automáticamente las etiquetas nominales del conjunto de atributos, asignando valores numéricos en el rango comprendido entre 0 y el número de clases menos 1, lo cual es necesario para la aplicación de algunos métodos de scikit-learn y es considerado, en general, como una buena práctica.

* http://bit.ly/2gMOri0

Ved también

Los archivos de datos y el programa están disponibles en el repositorio de la asignatura.

4.2.2. Máxima entropía

Supongamos que queremos realizar una aplicación para teléfonos móviles que ayude a los aficionados a la recolección de setas a discernir las setas venenosas de las comestibles a partir de sus propiedades: forma y color del sombrero. La tabla 13 muestra ejemplos de este tipo de datos.

Tabla 13. Conjunto de entrenamiento

class	cap-shape	cap-color
poisonous	convex	brown
edible	convex	yellow
edible	bell	white
poisonous	convex	white

Fuente: problema «mushroom» del repositorio UCI (Frank y Asunción, 2010)

Algoritmo de máxima entropía

El algoritmo de máxima entropía apareció por primera vez en 1957 en dos artículos de E. T. Jaynes.

El algoritmo de máxima entropía proporciona una manera flexible de combinar evidencias estadísticas de diferentes fuentes. La estimación de probabilidades no asume conocimientos previos de los datos y se ha probado que es muy robusto.

El principio de máxima entropía se ha utilizado en muchos ámbitos, no es un algoritmo de clasificación. En este subapartado vamos a ver la adaptación del principio de la máxima entropía para crear un clasificador.

Lectura complementaria

En este subapartado vamos a utilizar la aproximación que se hace del principio de máxima entropía en: **A. L. Berger, S. A. Della Pietra; V. J. Della Pietra** (1996). "A Maximum Entropy Approach to Natural Language Processing". *Computational Linguistics* (vol. 1, núm. 22).

Dado un conjunto de entrenamiento $\{(x_1,y_1),\ldots,(x_n,y_n)\}$ donde n el el número de ejemplos de entrenamiento, nuestro trabajo será encontrar las probabilidades condicionadas $p(y|f)$, donde f son los atributos que nos permitan definir el modelo de clasificación.

El principio de máxima entropía consiste en definir un problema a partir de restricciones, y a partir de aquí definimos la distribución de probabilidades que sea más uniforme. Una medida matemática de la uniformidad de una distribución condicional $p(y|x)$ nos la da la entropía condicional:

$$H(p) = -\sum_{x,y} \tilde{p}(x)p(y|x)\log(p(y|x))$$

donde $\tilde{p}(x)$ corresponde a la distribución de probabilidad empírica.

Representaremos las parejas atributo–valor a partir de funciones $f_i(x,y)$ que asignarán un valor de 1 si el ejemplo x le corresponde como valor del atributo a la pareja representada por f_i, o 0 en caso contrario. A modo de ejemplo, si x_1 corresponde al primer ejemplo de la tabla 13 y f_1 al atributo correspondiente a la pareja (*cap-shape, convex*), la función quedaría como:

$$f_1(x,y) = \begin{cases} 1 & \text{si } x \text{ tiene } \textit{convex} \text{ como valor del atributo } \textit{cap-shape} \\ 0 & \text{si no lo tiene} \end{cases}$$

y el valor de $f_1(x_1,y_1)$ sería 1.

El valor esperado de f con respecto a la distribución empírica $\tilde{p}(x,y)$ es:

$$\tilde{p}(f) = \sum_{x,y} \tilde{p}(x,y) f(x,y)$$

y el valor esperado de f con respecto al modelo $p(y|x)$ que queremos:

$$p(f) = \sum_{x,y} \tilde{p}(x) p(y|x) f(x,y)$$

que nos genera la restricción:

$$p(f) = \tilde{p}(f)$$

o

$$\sum_{x,y} \tilde{p}(x,y) f(x,y) = \sum_{x,y} \tilde{p}(x) p(y|x) f(x,y)$$

Si tenemos el conjunto de restricciones $C = \{p \in P | p(f_i) = \tilde{p}(f_i), \forall 1 \leq i \leq n\}$, el principio de máxima entropía trata de buscar la distribución de probabilidad con entropía máxima:

$$p^* = \underset{p \in C}{\operatorname{argmax}} H(p)$$

Añadiremos las restricciones que siguen para garantizar que p son distribuciones de probabilidad condicional:

$$p(y|x) \geq 0, \forall x,y$$

$$\sum_{y} p(y|x) = 1, \forall x$$

Para atacar este problema, aplicaremos el método de los multiplicadores de Lagrange de la teoría de la optimización de restricciones:

Ved también

El método de los multiplicadores de Lagrange se estudia en el subapartado 5.2 de este módulo.

$$\zeta(p,\Lambda,\gamma) = -\sum_{x,y} \tilde{p}(x) p(y|x) \log(p(y|x)) +$$
$$+ \sum_i \lambda_i \left(\sum_{x,y} \tilde{p}(x,y) f_i(x,y) - \tilde{p}(x) p(y|x) f_i(x,y)\right) +$$
$$+ \gamma \sum_x p(y|x) - 1$$

De las soluciones a este problema tendremos:

$$p_\lambda(y|x) = \frac{1}{Z_\lambda(x)} \exp\left(\sum_i \lambda_i f_i(x,y)\right)$$

donde $Z_\lambda(x)$ corresponde a un factor de normalización:

$$Z_\lambda(x) = \sum_i \exp\left(\sum_i \lambda_i f_i(x,y)\right)$$

Uso en Python

En el ámbito matemático y estadístico, es bien conocida la equivalencia entre los modelos basados en máxima entropía y los modelos de **regresión logística**. La librería scikit-learn implementa estos modelos mediante el módulo *LogisticRegression*. El siguiente código muestra un ejemplo de uso.

Referencia bibliográfica

A. L. Berger; S. A. Della Pietra; V. J. Della Pietra (1996). «A Maximum Entropy Approach to Natural Language Processing». *Computational Linguistics* (vol. 1, n.º 22).

Código 4.2: *logisticRegression* / máxima entropía en Python

```
1  import numpy as np
2  from sklearn.linear_model import LogisticRegression
3  from sklearn.model_selection import train_test_split
4  from sklearn import preprocessing
5
6  le = preprocessing.LabelEncoder()
7
8  with open("mushroom.data.txt","r") as fopen:
9      lines = [l.strip().split(',') for l in fopen]
10
11 attrs = np.array([l[1:] for l in lines]).flatten()
12 le.fit(attrs)
13 encoded_attrs = le.transform(attrs)
14
15 X = np.reshape(encoded_attrs,(len(lines),-1))
16
17 y_labels = np.array([l[0] for l in lines])
18 le.fit(y_labels)
19 y = le.transform(y_labels)
20
21 X_train, X_test, y_train, y_test = train_test_split(X, y,
22                                                     test_size=0.33)
23
24 clf = LogisticRegression()
25 clf.fit(X_train, y_train)
26
27 print("PRECISION:",clf.score(X_test,y_test))
```

Podéis encontrar la documentación relativa a este paquete en la documentación oficial de scikit-learn*. Como se puede observar, el constructor del modelo puede invocarse sin recibir parámetros, ya que los valores que toma por defecto son útiles para la mayoría de los casos. No obstante, para este caso, hay que tener en cuenta que el método de los multiplicadores de Lagrange deriva en la resolución de un sistema de ecuaciones diferenciales. Debido a esto, el modelo ofrece algunas alternativas en cuanto al método de resolución que se aplicará (parámetro *solver* del constructor). Esto, dependiendo de las características de nuestro conjunto de datos de entrenamiento (tamaño, pertenencia o no a múltiples clases, etc.), puede ser determinante para obtener un buen rendimiento, tanto en tiempo como en precisión. Asimismo, también existen parámetros para limitar el error deseado en esta resolución, o el

* http://bit.ly/2bDrRAS

número máximo de iteraciones (parámetros *tol* y *max_iter*, respectivamente). Es recomendable consultar a fondo la documentación oficial antes de utilizar el método.

4.3. Métodos basados en distancias

Los métodos de esta familia clasifican los nuevos ejemplos a partir de medidas de similitud o distancia. Esto se puede hacer comparando los nuevos ejemplos con conjuntos (uno por clase) de prototipos y asignando la clase del prototipo más cercano, o buscando en una base de ejemplos cuál es el más cercano.

4.3.1. kNN

Supongamos que queremos realizar una aplicación para un almacén de flores, donde llegan a diario miles de productos. Disponemos de un sistema láser que nos suministra una serie de medidas sobre las flores y nos piden que el sistema las clasifique automáticamente para transportarlas mediante un robot a las diferentes estanterías del almacén. Las medidas que envia el sistema láser son: la longitud y anchura del sépalo y el pétalo de cada flor. La tabla 14 muestra ejemplos de este tipo de datos.

Tabla 14. Conjunto de entrenamiento

class	sepal-length	sepal-width	petal-length	petal-width
setosa	5,1	3,5	1,4	0,2
setosa	4,9	3,0	1,4	0,2
versicolor	6,1	2,9	4,7	1,4
versicolor	5,6	2,9	3,6	1,3
virginica	7,6	3,0	6,6	2,1
virginica	4,9	2,5	4,5	1,7

Fuente: problema «iris» del repositorio UCI (Frank y Asunción, 2010)

Uno de los algoritmos representativos de esta familia es el kNN (k vecinos más cercanos*). En este algoritmo la clasificación de nuevos ejemplos se realiza buscando el conjunto de los *k* ejemplos más cercanos de entre un conjunto de ejemplos etiquetados previamente guardados y seleccionando la clase más frecuente de entre sus etiquetas. La generalización se pospone hasta el momento de la clasificación de nuevos ejemplos**.

*** En inglés, *k nearest neighbours*.**

**** Por esta razón es por la que en ocasiones es llamado aprendizaje perezoso (en inglés, *lazy learning***

Una parte muy importante de este método es la definición de la medida de distancia (o similitud) apropiada para el problema a tratar. Esta debería tener en cuenta la importancia relativa de cada atributo y ser eficiente computacionalmente. El tipo de combinación para escoger el resultado de entre los *k* ejemplos más cercanos y el valor de la propia *k* también son cuestiones a decidir de entre varias alternativas.

Este algoritmo, en su forma más simple, guarda en memoria todos los ejemplos durante el proceso de entrenamiento y la clasificación de nuevos ejemplos se basa en las clases de los k ejemplos más cercanos*. Para obtener el conjunto de los k vecinos más cercanos, se calcula la distancia entre el ejemplo a clasificar $x = (x_1, \ldots, x_m)$ y todos los ejemplos guardados $x_i = (x_{i1}, \ldots, x_{im})$. Una de las distancias más utilizadas es la euclídea:

*** Por esta razón se lo conoce también como basado en memoria, en ejemplos, en instancias o en casos.**

$$de(x,x_i) = \sqrt{\sum_{j=1}^{m}(x_j - x_{ij})^2}$$

Ejemplo de aplicación

La tabla 14 muestra un conjunto de entrenamiento en el que tenemos que clasificar flores a partir de sus propiedades. En este ejemplo, aplicaremos el kNN para valores de k de 1 y 3, utilizando como medida de distancia la euclidiana.

El proceso de entrenamiento consiste en guardar los datos; no tenemos que hacer nada. A partir de aquí, si nos llega un ejemplo nuevo como el que muestra la tabla 15, tenemos que calcular las distancias entre el nuevo ejemplo y todos los del conjunto de entrenamiento. La tabla 16 muestra estas distancias.

Tabla 15. Ejemplo de test

class	sepal-length	sepal-width	petal-length	petal-width
setosa	4,9	3,1	1,5	0,1

Fuente: problema «iris» del repositorio UCI (Frank y Asunción, 2010)

Tabla 16. Distancias

0,5	0,2	3,7	2,5	6,1	3,5

Para el 1NN escogemos la clase del ejemplo de entrenamiento más cercano que coincide con el segundo ejemplo (distancia 0,2) que tiene por clase *setosa*. Para el 3NN escogemos los tres ejemplos más cercanos: primero, segundo y cuarto; con distancias respectivas: 0,5, 0,2 y 2,5. Sus clases corresponden a *setosa*, *setosa* y *versicolor*. En este caso asignaremos también a *setosa* por ser la clase más frecuente. En los dos casos el resultado es correcto.

Análisis del método

Un aspecto a tener en cuenta tiene que ver con la eficiencia computacional; este tiene que ver con que el algoritmo realiza todos los cálculos en el proceso de clasificación. Así, aun siendo un método rápido globalmente, tenemos que tener en cuenta que el proceso de clasificación no lo es. Esto puede llegar a ser crítico para aplicaciones de tiempo real que necesiten una respuesta rápida.

En el momento en que queramos aplicar este algoritmo a un problema, la primera decisión que tenemos que tomar es la medida de distancia y la segunda

el valor de *k*. Se suele escoger un número impar o primo para minimizar la posibilidad de empates en las votaciones. La siguiente cuestión a decidir es el tratamiento de los empates cuando la *k* es superior a uno. Algunos heurísticos posibles son: no dar predicción en caso de empate, dar la clase más frecuente en el conjunto de entrenamiento de entre las clases seleccionadas para votar... Se suele escoger el heurístico en función del problema a tratar.

Las distancias más utilizadas son la euclídea y la de Hamming, en función del tipo de atributos que tengamos. La primera se utiliza mayoritariamente para atributos numéricos y la segunda para atributos nominales o binarios. Otros tipos de distancias pueden ser utilizadas dependiendo del problema a tratar. Un ejemplo de esto es la distancia MVDM. Esta medida sustituye a la de Hamming en el tratamiento de atributos nominales y ha dado buenos resultados en problemas de procesamiento del lenguaje natural parecidos a la categorización de textos. Su inconveniente principal es su alto coste computacional.

Distancia de Hamming

La distancia de Hamming es $dh(x,x_i) = \sum_{j=1}^{m} \delta(x_j,x_{ij})$ donde $\delta(x_j,x_{ij})$ es la distancia entre dos valores que corresponde a 0 si $x_j = x_{ij}$ y a 1 si son diferentes.

MVDM

MVDM son las siglas del inglés, *modified value difference metric*. Propuesta por: S. Cost; S. Salzberg (1993). "A Wighted Nearest Neighbor Algorithm for Learning with Symbolic Features". *Machine Learning* (núm. 10)

Una variante del algoritmo muy utilizada es la utilización de ejemplos ponderados. Consiste en la introducción de una modificación en el esquema de votación de los *k* vecinos más cercanos, que hace la contribución de cada ejemplo proporcional a su importancia. Esta importancia se mide teniendo en cuenta la cercanía al ejemplo de test.

La ponderación de los atributos también se ha utilizado. Esto consiste en realizar un "ranking" de los atributos en función de su relevancia y hacer que su contribución afecte al cálculo de la distancia. Los pesos de los atributos se pueden calcular de diversas formas; una de ellas es la distancia RLM*****. A modo de ejemplo, aplicar este concepto a la distancia de Hamming, una vez calculados los pesos, da como resultado:

*** Distancia por Ramón López de Mántaras en: R. López de Mántaras (1991). "A distance-based attribute selection measure for decision tree induction". *Machine Learning* (núm. 6, págs. 81-92).**

$$dh(x,x_i) = \sum_{j=1}^{m} w_j \delta(x_j,x_{ij})$$

Otra forma de tratamiento de atributos es su selección, sobre todo en problemas en los que su número es importante. La selección de atributos consiste en reducir el número de atributos de los ejemplos. La idea es doble: selección de los más relevantes y eliminación de los que producen ruido. Una forma de realizar esto consiste en dividir el conjunto de entrenamiento en dos: uno de entrenamiento más pequeño y otro llamado de validación. Se aplica un algoritmo de optimización (como los descritos en el apartado 5) escogiendo como función objetivo el resultado de la clasificación sobre el conjunto de validación. El conjunto de entrenamiento de estos clasificadores es la parte del de entrenamiento que no pertenece al de validación. Una vez seleccionados los atributos, se entrena el clasificador final con todo el conjunto de entrenamiento y se prueba con el de test real. Otra forma de realizar esto consiste en aplicar alguno de los métodos descritos en el apartado 3, como el *PCA*.

Lectura recomendada

En la referencia siguiente se utilizan algoritmos genéticos para seleccionar atributos y el ajuste de parámetros de un algoritmo basado en kNN: B. Decadt; V. Hoste; W. Daelemans; A. van den Bosch (2004). GAMBL, Genetic Algorithm Optimization of Memory-Based WSD. En las actas del *International Workshop on Evaluating WSD Systems, Senseval* (núm. 3)

Una de las grandes ventajas de este método es la conservación de excepciones en el proceso de generalización, y uno de los grandes inconvenientes conocidos es la sensibilidad a procesos de selección de atributos.

Implementación en Python

El programa 4.3 corresponde a la implementación, utilizando los módulos de la librería scikit-learn de Python, del kNN básico. En esta versión se ha escogido como distancia la euclídea y 3 como valor de la *k*.

Código 4.3: kNN en Python

```
1  from sklearn.neighbors import KNeighborsClassifier
2  from sklearn.model_selection import train_test_split
3  from sklearn import datasets
4
5  iris = datasets.load_iris()
6  X = iris.data
7  y = iris.target
8
9  X_train, X_test, y_train, y_test = train_test_split(X, y,
10                                                     test_size=0.33)
11
12 clf = KNeighborsClassifier(n_neighbors=3, metric='euclidean')
13 clf.fit(X_train, y_train)
14
15 print("PRECISION:",clf.score(X_test,y_test))
```

Como se puede observar, para este ejemplo utilizamos el archivo de datos del problema «iris» del UCI (Frank y Asunción, 2010) pero, en esta ocasión, hacemos uso del módulo *datasets* que incorpora scikit-learn, donde además de este podremos encontrar otros conjuntos de datos listos para ser usados en problemas de aprendizaje automático. Podéis encontrar la documentación relativa al paquete KNeighborsClassifier en la documentación oficial de scikit-learn*. Como se puede observar, el constructor del modelo es invocado especificando los parámetros *n_neighbors* para el número de vecinos (*k*) que hay que tener en cuenta y *distance* para escoger la métrica de distancia. No obstante, existen otros parámetros que pueden hacer que el clasificador se adapte mejor a vuestro problema. Es recomendable consultar a fondo la documentación oficial antes de utilizar el método.

Repositorio

Los archivos de datos y el programa están disponibles en el repositorio de la asignatura.

*** http://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html**

4.3.2. Clasificador lineal basado en distancias

Una forma muy parecida a la del algoritmo anterior de abordar el problema de la clasificación de flores se basa en el uso de centros de masa o centroides. El modelo de clasificación de este algoritmo consta de un centroide, que representa cada una de las clases que aparecen en el conjunto de entrenamiento*. El valor de cada atributo del centroide se calcula como el promedio del valor del mismo atributo de todos los ejemplos del conjunto de entrenamiento que pertenecen a su clase. La fase de clasificación consiste en aplicar el 1NN con distancia euclídea, seleccionando como conjunto de entrenamiento los centroides calculados previamente.

*** Este método también recibe el nombre de método basado en centroides o en centros de masas.**

Ejemplo de aplicación

Aplicaremos como ejemplo el algoritmo a los ejemplos de la tabla 14. El proceso de entrenamiento consiste en calcular los centroides. La tabla 17 muestra el resultado. A modo de ejemplo, el 5 del atributo *sepal-length* del centroide *setosa* sale del promedio de 4.9 y 5.1 de la tabla 14.

Tabla 17. Centroides

class	sepal-length	sepal-width	petal-length	petal-width
setosa	5	3,25	1,4	0,2
versicolor	5,85	2,9	4,15	1,35
virginica	6,25	2,75	5,55	1,9

A partir de aquí, si nos llega un ejemplo nuevo como el que muestra la tabla 15, tenemos que calcular las distancias entre el nuevo ejemplo y todos los centroides para el 1NN. La tabla 18 muestra estas distancias. Al escoger el más cercano (distancia 0,2) que tiene por clase *setosa*, vemos que el ejemplo queda bien clasificado.

Tabla 18. Distancias

0,2	3,1	4,6

Análisis del método

Este algoritmo puede aplicarse utilizando un producto escalar y se ha utilizado mucho desde este otro punto de vista. En el subapartado 4.5.1 se plantea este nuevo punto de vista y se analiza en más profundidad.

Implementación en Python

El programa 4.4 corresponde a la implementación en Python del clasificador lineal.

Código 4.4: clasificador lineal basado en distancias en Python

```
1  from sklearn.neighbors import NearestCentroid
2  from sklearn.model_selection import train_test_split
3  from sklearn import datasets
4
5  iris = datasets.load_iris()
6  X = iris.data
7  y = iris.target
8
9  X_train, X_test, y_train, y_test = train_test_split(X, y,
10                                                     test_size=0.33)
11
12 clf = NearestCentroid() # Distancia euclidiana por defecto
13 clf.fit(X_train, y_train)
14
15 print("PRECISION:",clf.score(X_test,y_test))
```

Como se puede observar, para este ejemplo utilizamos el archivo de datos del problema «iris» del UCI (Frank y Asunción, 2010). Podéis encontrar la documentación relativa al paquete NearestCentroid en la documentación oficial de scikit-learn*. Como se puede observar, el constructor del modelo es invocado sin especificar parámetros, ya que por defecto toma la distancia euclídea como métrica de distancia. No obstante, existen otras métricas que pueden hacer que el clasificador se adapte mejor a vuestro problema. Es recomendable consultar a fondo la documentación oficial antes de utilizar el método.

Repositorio

Los archivos de datos y el programa están disponibles en el repositorio de la asignatura.

*** http://bit.ly/2gHztq9**

4.3.3. *Clustering* dentro de clases

Este algoritmo está a medio camino entre los dos anteriores. En este caso, abordaremos el problema de la clasificación de las flores teniendo las ventajas y/o inconvenientes de los dos.

Consiste en aplicar un algoritmo de categorización (cualquiera sirve) para calcular cierto número de centroides para cada una de las clases que aparece en el conjunto de entrenamiento. Una vez hecho esto, utiliza el kNN seleccionando todos los centroides como conjunto de entrenamiento y aplica el 1NN para la fase de clasificación.

Implementación en Python

El programa 4.5 corresponde a la implementación, utilizando los módulos de la librería scikit-learn de Python, del algoritmo que usa *clustering* dentro de clases. Como método de *clustering* se ha utilizado *el k-means**, pero funcionaría con cualquier otro método.

*** Se ha utilizado el algoritmo de *clustering* explicado en el subapartado 2.3.4 de este módulo.**

Código 4.5: *k-means* supervisado en Python

```
1  from sklearn.cluster import KMeans
2  from sklearn.neighbors import KNeighborsClassifier
3  from sklearn.model_selection import train_test_split
4  from sklearn import datasets
5  from functools import reduce
6  import numpy as np
7
8  k = 3 # numero de centroides por clase para k-means
9  m = 10 # maximo numero de iteraciones
10
11 iris = datasets.load_iris()
12 X = iris.data
13 y = iris.target
14
15 X_train, X_test, y_train, y_test = train_test_split(X, y,
16                                                    test_size=0.33)
17
18 Xcent = {} # Diccionario con los k centroides de cada clase
19
20 for c in set(y):
21         # Indice de elementos con clase c
22     yc = list(filter(lambda z: z[1] == c, enumerate(y_train)))
23     # Elementos cuyo indice corresponde con el de yc
```

```
    Xc = X_train[[i[0] for i in yc]]
    # Aplicamos k-means
    kmeans = KMeans(n_clusters=k,max_iter=m).fit(Xc)
    # Obtenemos centroides
    Xcent[c] = kmeans.cluster_centers_.tolist()

#Concatenamos listas de centroides por clase
X_train_kNN = reduce(lambda u,v: u+v, Xcent.values())
y_train_kNN = []
for c in Xcent.keys():
    y_train_kNN.extend([c]*k)

clf = KNeighborsClassifier(n_neighbors=1)
clf.fit(np.array(X_train_kNN), np.array(y_train_kNN))


print("PRECISION:",clf.score(X_test,y_test))
```

Como se puede observar, para este ejemplo utilizamos el archivo de datos del problema «iris» del UCI (Frank y Asunción, 2010). Podéis encontrar la documentación relativa al módulo *KMeans* en la documentación oficial de scikit-learn*. Como se puede observar, el constructor del modelo *k-means* es invocado de modo que se escoge una *k* de 3 y un máximo de diez iteraciones. No obstante, existen otras métricas que pueden hacer que el clasificador se adapte mejor a vuestro problema. Es recomendable consultar a fondo la documentación oficial antes de utilizar el método. En el paso final se hace uso del anteriormente estudiado KNeighborsClassifier, con $k = 1$, para aplicar un 1NN.

Repositorio

Los archivos de datos y el programa están disponibles en el repositorio de la asignatura.

*** http://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html**

4.4. Métodos basados en reglas

Estos métodos adquieren reglas de selección asociadas a cada una de las clases. Dado un ejemplo de test, el sistema selecciona la clase que verifica algunas de las reglas que determinan una de las clases.

4.4.1. Árboles de decisión

Supongamos que queremos realizar una aplicación para teléfonos móviles que ayude a los aficionados a la recolección de setas a discernir las setas venenosas de las comestibles a partir de sus propiedades: forma y color del sombrero y color del tronco.

Un árbol de decisión es una forma de representar reglas de clasificación inherentes a los datos, con una estructura en árbol *n*-ario que particiona los datos de manera recursiva. Cada rama de un árbol de decisión representa una regla que decide entre una conjunción de valores de un atributo básico (nodos internos) o realiza una predicción de la clase (nodos terminales).

El algoritmo de los árboles de decisión básico está pensado para trabajar con atributos nominales. El conjunto de entrenamiento queda definido por $S = \{(x_1,y_1),\ldots,(x_n,y_n)\}$, donde cada componente x corresponde a $x_i = (x_{i_1},\ldots,x_{i_m})$ donde m corresponde al número de atributos de los ejemplos de entrenamiento; y el conjunto de atributos por $A = \{a_1,\ldots,a_m\}$ donde $\mathrm{dom}(a_j)$ corresponde al conjunto de todos los posibles valores del atributo a_j, y para cualquier valor de un ejemplo de entrenamiento $x_{i_j} \in \mathrm{dom}(a_j)$.

El proceso de construcción del árbol es un proceso iterativo, en el que, en cada iteración, se selecciona el atributo que mejor particiona el conjunto de entrenamiento. Para realizar este proceso, tenemos que mirar la bondad de las particiones que genera cada uno de los atributos y, en un segundo paso, seleccionar el mejor. La partición del atributo a_j genera $|\mathrm{dom}(a_j)|$ conjuntos, que corresponde al número de elementos del conjunto. Existen diversas medidas para mirar la bondad de la partición. Una básica consiste en asignar a cada conjunto de la partición la clase mayoritaria del mismo y contar cuántos quedan bien clasificados y dividirlo por el número de ejemplos. Una vez calculadas las bondades de todos los atributos, escogemos el mejor.

Cada conjunto de la mejor partición pasará a ser un nuevo nodo del árbol. A este nodo se llegará a través de una regla del tipo *atributo* = *valor*. Si todos los ejemplos del conjunto han quedado bien clasificados, lo convertimos en nodo terminal con la clase de los ejemplos. En caso contrario, lo convertimos en nodo interno y aplicamos una nueva iteración al conjunto ("reducido") eliminando el atributo que ha generado la partición. En caso de no quedar atributos, lo convertiríamos en nodo terminal asignando la clase mayoritaria.

Para realizar el test, exploramos el árbol en función de los valores de los atributos del ejemplo de test y las reglas del árbol hasta llegar al nodo terminal, y damos como predicción la clase del nodo terminal al que lleguemos.

Ejemplo de aplicación

La tabla 19 es una simplificación de la tabla 9. Para construir un árbol de decisión a partir de este conjunto tenemos que calcular la bondad de las particiones de los tres atributos: *cap-shape*, *cap-color* y *gill-color*. El atributo *cap-shape* nos genera una partición con dos conjuntos: uno para el valor *convex* y otro para *bell*. La clase mayoritaria para el conjunto de convex es *poisonous* y la de bell es *edible*; su bondad es $bondad(cap-shape) = (3+2)/7 = 0{,}71$. Si realizamos el mismo proceso para el resto de atributos obtenemos: $bondad(cap-color) = (1+2+2)/7 = 0{,}71$ y $bondad(gill-color) = (1+3+1)/7 = 0{,}71$.

El siguiente paso consiste en seleccionar el mejor atributo. Hemos obtenido un empate entre los tres atributos, podemos escoger cualquiera de ellos; escogemos *cap-color*. Los nodos generados por el conjunto de *brown* y *yellow* son terminales y les asignamos las clases *poisonous* y *edible*, respectivamente. Esto es debido a que obtienen los dos conjuntos una bondad de 1. El nodo de *white*

Tabla 19. Conjunto de entrenamiento

class	cap-shape	cap-color	gill-color
poisonous	convex	brown	black
edible	convex	yellow	black
edible	bell	white	brown
poisonous	convex	white	brown
edible	convex	yellow	brown
edible	bell	white	brown
poisonous	convex	white	pink

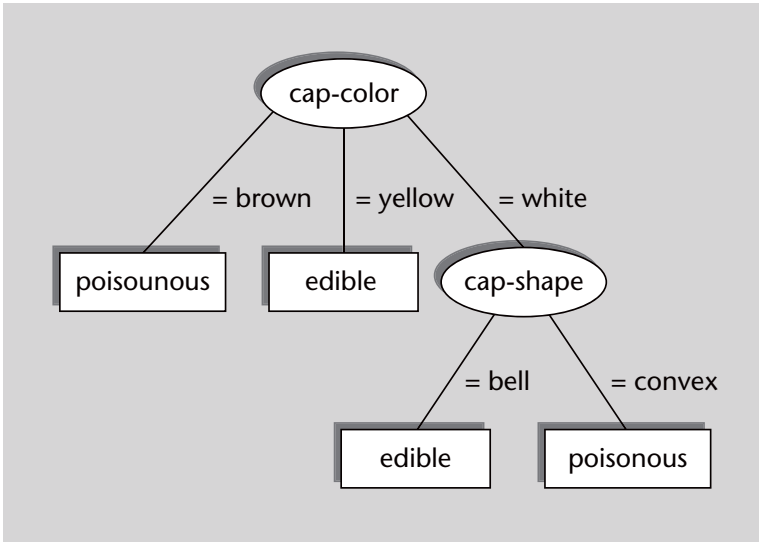
Fuente: problema «mushroom» del repositorio UCI (Frank y Asunción, 2010)

no queda con los datos que muestra la tabla 20. Este conjunto lo obtenemos de eliminar el atributo *cap-color* de los ejemplos que tienen el valor *white* para el atributo *cap-color*. Volvemos a iterar; la bondad de las nuevas particiones es: $bondad(cap-shape) = (2+2)/4 = 1$ y $bondad(gill-color) = (2+1)/4 = 0{,}75$. El mejor atributo es *cap-shape*, que genera dos nodos terminales con las clases *edible* para *bell* y *poisonous* para *convex*. La figura 23 muestra el árbol construido en este proceso.

Tabla 20. Conjunto de la segunda iteración

class	cap-shape	gill-color
edible	bell	brown
poisonous	convex	brown
edible	bell	brown
poisonous	convex	pink

Figura 23. Árbol de decisión



Para etiquetar un ejemplo de test como el que muestra la tabla 21 tenemos que recorrer el árbol, partiendo de la raíz, escogiendo las ramas correspondientes a los valores de los atributos de los ejemplos de test. Para este caso, miramos el valor del atributo *cap-color* y bajamos por la rama que corresponde al valor *brown*. Llegamos a un nodo terminal con classe *poisonous* con lo que se la asignaremos al ejemplo de test como predicción. Esta predicción es correcta.

Tabla 21. Ejemplo de test

class	cap-shape	cap-color	gill-color
poisonous	convex	brown	black

Fuente: problema «mushroom» del repositorio UCI (Frank y Asunción, 2010)

Análisis del método

Este algoritmo tiene la gran ventaja de la facilidad de interpretación del modelo de aprendizaje. Un árbol de decisión nos da la información clara de la toma de decisiones y de la importancia de los diferentes atributos involucrados. Por esta razón se ha utilizado mucho en diversos campos, como en el financiero.

Dos de los grandes inconvenientes que plantea son la elevada fragmentación de los datos en presencia de atributos con muchos valores y el elevado coste computacional que esto implica. Esto hace que no sea un método muy adecuado para problemas con grandes espacios de atributos; como pueden ser aquellos que contienen información léxica, como la categorización de textos o los filtros anti spam.

Otro inconveniente a tener en cuenta es que los nodos terminales correspondientes a reglas que dan cobertura a pocos ejemplos de entrenamiento no producen estimaciones fiables de las clases. Tiende a sobreentrenar el conjunto de entrenamiento*. Para suavizar este efecto se puede utilizar alguna técnica de poda**. Más adelante se explica una de ellas.

***Este efecto se conoce en inglés como *overfitting*.**

**** En inglés, *prunning*.**

Tratamiento de atributos numéricos

Supongamos que ahora queremos realizar una aplicación para un almacén de flores, donde llegan a diario miles de productos. Disponemos de un sistema láser que nos suministra la longitud del sépalo de las flores y nos piden que el sistema las clasifique automáticamente para transportarlas mediante un robot a las diferentes estanterías del almacén.

El algoritmo de los árboles de decisión fue diseñado para tratar con atributos nominales, pero existen alternativas para el tratamiento de atributos numéricos. El más común se basa en la utilización de puntos de corte. Estos son un punto que divide el conjunto de valores de un atributo en dos (los menores y los mayores del punto de corte).

Para calcular el mejor punto de corte de un atributo numérico, se ordenan los valores y se eliminan los elementos repetidos. Se calculan los posibles puntos de corte como el promedio de cada dos valores consecutivos. Como último paso se calcula el mejor de ellos como aquél con mejor bondad.

Tenemos que tener en cuenta que el tratamiento de atributos numéricos genera árboles (y decisiones) binarios. Esto implica que en los siguientes niveles del árbol tendremos que volver a procesar el atributo numérico que acabamos de seleccionar, a diferencia de los atributos nominales.

Ejemplo del cálculo del mejor punto de corte de un atributo numérico

A continuación se expone el cálculo del mejor punto de corte para el atributo *sepal-length* del conjunto de datos que muestran la tabla 14. La tabla 22 muestra estos cálculos: las columnas 1 y 2 muestra la clase y el atributo ordenados por el atributo, la 3 muestra los valores sin los repetidos, la 4 los puntos de corte como el promedio de cada dos valores consecutivos y la última columna muestra las bondades de los puntos de corte. El mejor punto de corte es 5,35 con una bondad de 66,7 %.

Tabla 22. Cálculos para el mejor punto de corte

clase	sepal-length	sin los repetidos	puntos de corte	bondad
setosa	4,9			
virgínica	4,9	4,9	5	(1 + 2)/6 = 0,5
setosa	5,1	5,1	5,35	(2 + 2)/6 = 0,67
versicolor	5,6	5,6	5,85	(2 + 1)/6 = 0,5
versicolor	6,1	6,1	6,85	(2 + 1)/6 = 0,5
virgínica	7,6	7,6		

Algoritmo ID3

El algoritmo ID3 es una modificación de los árboles de decisión que utiliza la ganancia de información* como medida de bondad para analizar los atributos. La ganancia de información es un concepto que está basado en la entropía. El uso de la entropía tiende a penalizar más los conjuntos mezclados.

*** En inglés, *Information Gain*.**

La entropía es una medida de la teoría de la información que cuantifica el desorden. Así, dado un conjunto *S*, su fórmula viene dada por:

$$H(S) = -\sum_{y \in Y} p(y) \log_2(p(y))$$

donde $p(y)$ es la proporción de ejemplos de *S* que pertenece a la clase *y*. La entropía será mínima (0) cuando en S hay una sola clase y máxima (1) cuando el conjunto es totalmente aleatorio.

A modo de ejemplo, la entropía del conjunto que muestra la tabla 19 viene dada por:

$$H(S) = -\frac{3}{7}\log_2\frac{3}{7} - \frac{4}{7}\log_2\frac{4}{7} = 0{,}985$$

La fórmula de la ganancia de información para el conjunto S y el atributo a_j queda como:

$$G(S,a_j) = H(S) - \sum_{v \in a_j} p(v)H(S_v)$$

donde $p(v)$ es la proporción de ejemplos de S que tienen el valor v para el atributo a_j y S_v es el subconjunto de S de los ejemplos que toman el valor v para el atributo a_j.

A modo de ejemplo, la ganancia de información del conjunto que muestra la tabla 19 y el atributo *cap_shape* viene dada por:

$$G(S,cs) = H(S) - \frac{5}{7}H(S_{cs=convex}) - \frac{2}{7}H(S_{cs=bell}) = 0{,}292$$

donde *cs* es *cap – shape*,

$$H(S_{cs=convex}) = -\frac{3}{5}\log_2\frac{3}{5} - \frac{2}{5}\log_2\frac{2}{5} = 0{,}971$$

y

$$H(S_{cs=bell}) = -\frac{0}{2}\log_2\frac{0}{2} - \frac{2}{2}\log_2\frac{2}{2} = 0$$

Existen otras variantes de árboles de decisión que utilizan otras medidas para computar la bondad de los atributos. Algunas de ellas son el uso de la varianza cuando nos encontramos atributos numéricos. Otra medida utilizada es la información mutua*, que está relacionada con la entropía cruzada.

* En inglés, *mutual information.*

Ved también

Sobre la información mutua podéis ver el subapartado 3.1.3.

Poda de los árboles de decisión

El objetivo de la poda de los árboles de decisión es obtener árboles que no tengan en las hojas reglas que afecten a pocos ejemplos del conjunto de entrenamiento. Es deseable no forzar la construcción del árbol para evitar el sobreentrenamiento*.

* En inglés, *overfitting.*

La primera aproximación para abordar la solución de este problema consiste en establecer un umbral de reducción de la entropía. Es decir, pararemos una rama cuando la disminución de entropía no supere dicho umbral. Esta aproximación plantea el problema de que puede ser que no se reduzca en un cierto paso pero sí en el siguiente.

Otra aproximación, que soluciona el problema de la anterior y que es la que se suele utilizar, consiste en construir todo el árbol y, en una segunda fase, eliminar los nodos superfluos (poda del árbol*). Para realizar este proceso de poda recorremos el árbol de forma ascendente (de las hojas a la raíz) y vamos mirando si cada par de nodos incrementa la entropía por encima de un cierto umbral si los juntáramos. Si esto pasa, deshacemos la partición.

* En inglés, *prunning*.

Implementación en Python

El programa 4.6 corresponde a la implementación en Python de los árboles de decisión básicos, utilizando los módulos de la librería scikit-learn de Python.

Código 4.6: árboles de decisión en Python

```
1  import numpy as np
2  from sklearn.tree import DecisionTreeClassifier
3  from sklearn.model_selection import train_test_split
4  from sklearn import preprocessing
5
6  le = preprocessing.LabelEncoder()
7
8  with open("mushroom.data.txt","r") as fopen:
9      lines = [l.strip().split(',') for l in fopen]
10
11 attrs = np.array([l[1:] for l in lines]).flatten()
12 le.fit(attrs)
13 encoded_attrs = le.transform(attrs)
14
15 X = np.reshape(encoded_attrs,(len(lines),-1))
16
17 y_labels = np.array([l[0] for l in lines])
18 le.fit(y_labels)
19 y = le.transform(y_labels)
20
21 X_train, X_test, y_train, y_test = train_test_split(X, y,
22                                                     test_size=0.33)
23
24 clf = DecisionTreeClassifier()
25 clf.fit(X_train, y_train)
26
27 print("PRECISION:",clf.score(X_test,y_test))
```

Podéis encontrar la documentación relativa al paquete DecisionTreeClassifier en la documentación oficial de scikit-learn*. Como se puede observar, el constructor del modelo puede invocarse sin recibir parámetros, ya que los valores que toma por defecto son útiles para la mayoría de los casos. No obstante, existe una serie de parámetros admitidos por el constructor que configuran de manera notable el árbol de decisión que se construye, como son la máxima profundidad del árbol, el número mínimo de instancias que ha de tener una hoja, el número mínimo de instancias para expandir un nodo, o el número máximo de hojas permitidas. Es recomendable consultar a fondo la documentación oficial antes de utilizar el método.

Ved también

Los archivos de datos y el programa están disponibles en el repositorio de la asignatura.

* http://bit.ly/1T5sf92

4.4.2. AdaBoost

Supongamos que queremos realizar una aplicación para teléfonos móviles que ayude a los aficionados a la recolección de setas a discernir las setas venenosas de las comestibles a partir de sus propiedades: forma y color del sombrero.

Este método está específicamente diseñado para combinar reglas. Se basa en la combinación lineal de muchas reglas muy sencillas pero muy precisas (llamadas *reglas débiles**), para crear un clasificador muy robusto con un error arbitrariamente bajo en el conjunto de entrenamiento. Estas reglas débiles se aprenden secuencialmente manteniendo una distribución de pesos sobre los ejemplos de entrenamiento. Estos pesos se van actualizando a medida que vamos adquiriendo nuevas reglas. Esta actualización depende de la dificultad de aprendizaje de los ejemplos de entrenamiento. Es decir, los pesos de los ejemplos serán directamente proporcionales a su dificultad de aprendizaje, y por tanto, la adquisición de nuevas reglas se irá dirigiendo a los ejemplos con mayor peso (y dificultad).

* En inglés, *weak rules* o *weak hypotheses*.

Lectura complementaria

El algoritmo AdaBoost fue propuesto en: Y. Freund; R. E. Schapire (1997). "A Decision–theoric Generalization of On-line Learning and an Application to Boosting". *Journal of Computer and System Sciences* (vol. 1, núm. 55).

Referencia bibliográfica

Y. Freund; R. E. Schapire (1999). "A Short Introduction to Boosting". *Journal of Japanese Society for Artificial Intelligence* (vol. 5, núm. 14, págs. 771-780).

AdaBoost básico

AdaBoost es un algoritmo que pretende obtener una regla de clasificación muy precisa combinando muchos clasificadores débiles, cada uno de los cuales obtiene una precisión moderada. Este algoritmo trabaja eficientemente con espacios de atributos muy grandes y ha sido aplicado con éxito a muchos problemas prácticos. A continuación se muestra el pseudocódigo del algoritmo:

algoritmo AdaBoost
entrada: $S = \{(x_i, y_i), \forall i = 1..m\}$
S es el conjunto de ejemplos de entrenamiento
D_1 es la distribución inicial de pesos
m es el número de ejemplos de entrenamiento
$y_i \in \{-1, +1\}$
$D_1(i) = \frac{1}{m}, \forall i = 1..m$
Se realizan T iteraciones:
para $t := 1$ **hasta** T **hacer**
Se obtiene la hipótesis débil $h_t : X \to \{-1, +1\}$
$h_t =$ `ObtenerHipótesisDébil`(X, D_t)
$\epsilon_t = \sum_{i:h_t(x_i) \neq y_i} D_t(i)$
$\alpha_t = \frac{1}{2} \ln(\frac{1-\epsilon_t}{\epsilon_t})$
Se actualiza la distribución D_t
$D_{t+1}(i) = \frac{D_t(i)\exp(-\alpha_t y_i h_t(x_i))}{Z_t}, \forall i = 1..m$
Z_t es un factor de normalización tal que D_{t+1} sea una
distribución, por ejemplo: $Z_t = \sum_{i=1}^{m} D_t(i)\exp(-\alpha_t y_i h_t(x_i))$
fin para
devuelve la combinación de hipótesis: $f(x) = \sum_{t=1}^{T} \alpha_t h_t(x)$
fin AdaBoost

Las hipotesis débiles se aprenden secuencialmente. En cada iteración se aprende una regla que se sesga para clasificar los ejemplos con más dificultades por parte del conjunto de reglas precedentes. AdaBoost mantiene un vector de pesos como una distribución D_t sobre los ejemplos. En la iteración t, el objetivo del algoritmo es encontrar una hipotesis débil, $h_t : X \rightarrow \{-1, +1\}$, con un error moderadamente bajo con respecto a la distribución de pesos D_t. En este marco, las hipotesis débiles $h_t(x)$ dan como predicciones valores reales que corresponden a valores de confianza. Inicialmente, la distribución de pesos D_1 es uniforme, y en cada iteración, el algoritmo de *boosting* incrementa (o decrementa) exponencialmente los pesos $D_t(i)$ en función de si $h_t(x_i)$ realiza una buena (o mala) predicción. Este cambio exponencial depende de la confianza $|h_t(x_i)|$. La combinación final de hipotesis, $h_t : X \rightarrow \{-1, +1\}$, calcula sus predicciones ponderando con pesos los votos de las diferentes hipotesis débiles,

$$f(x) = \sum_{t=1}^{T} \alpha_t \cdot h_t(x)$$

Para cada ejemplo x, el signo de $f(x)$ se interpreta como la clase predicha (el AdaBoost básico trabaja sólo con dos clases de salida, –1 o +1), y la magnitud $|f(x)|$ como una medida de la confianza de la predicción. Esta función se puede usar para clasificar nuevos ejemplos o para realizar un ranking de los mismos en función de un grado de confianza.

Ejemplo de aplicación

El algoritmo Adaboost tiene dos parámetros a determinar: el número máximo de iteraciones y la forma de construcción de las reglas débiles. En este subapartado vamos a ver la aplicación del algoritmo escogiendo como conjunto de entrenamiento los datos de la tabla 23 y como test el de la tabla 24. Fijamos el número máximo de iteraciones a 5 y como reglas débiles, reglas del estilo: *atributo* = *valor* y *atributo* ≠ *valor*.

Tabla 23. Conjunto de entrenamiento

class	cap-shape	cap-color
+1	convex	brown
–1	convex	yellow
–1	bell	white
+1	convex	white

Fuente: problema «mushroom» del repositorio UCI (Frank y Asunción, 2010)

Tabla 24. Ejemplo de test

class	cap-shape	cap-color
–1	bell	yellow

Fuente: problema «mushroom» del repositorio UCI (Frank y Asunción, 2010)

Como primer paso del algoritmo inicializamos los 4 elementos del vector de pesos D_1 a 0,25. A continuación, empezamos el proceso iterativo. Lo primero es calcular la regla débil. La tabla 25 muestra las diferentes reglas aplicables* y su error. La regla *cs* = *bell* asignará las predicciones: –1, –1, +1 y –1, respectivamente. Para calcular el error sumamos el peso $D_t(i)$ de los ejemplos cuyas predicciones sean diferentes a sus clases. En este caso sumaríamos el peso del primer, segundo y cuarto ejemplos (en total 0,75). El resto de casos los calcularíamos de manera similar.

*** Donde *cs* corresponde a *cap-shape* y *cc* a *cap-color*.**

Tabla 25. Reglas débiles

Regla	Error
cs = *bell*	0,75
cs ≠ *bell*	0,25
cs = *convex*	0,25
cs ≠ *convex*	0,75
cc = *brown*	0,25
cc ≠ *brown*	0,75
cc = *white*	0,5
cc ≠ *white*	0,5
cc = *yellow*	0,75
cc ≠ *yellow*	0,25

Escogemos como regla débil una de las que tengan error mínimo (en el ejemplo hemos escogido *cs* ≠ *bell*). Así, el error de la primera iteración es $\epsilon_1 = 0{,}25$ y por tanto $\alpha_1 = \frac{1}{2}\ln(\frac{1-\epsilon_1}{\epsilon_1}) = 0{,}5493$.

El paso siguiente consiste en actualizar el vector de pesos. Empezamos por calcular $D_1(i)\exp(-\alpha_1 y_i h_1(x_i))$. La tabla 26 muestra el valor de esta expresión para los 4 ejemplos y el valor de todas sus componentes. La última columna muestra los valores finales del vector de pesos para la segunda iteración.

Tabla 26. Cálculo de D_2

i	$D_1(i)$	α_1	y_i	$h1_(x_i)$	*numerador*	$D_2(i)$
1	0,25	0,5493	+1	+1	0,1443376	0,1667
2	0,25	0,5493	–1	+1	0,4330127	0,5
3	0,25	0,5493	–1	–1	0,1443376	0,1667
4	0,25	0,5493	+1	+1	0,1443376	0,1667

El resto de iteraciones se calcularían de forma similar. La tabla 27 muestra los resultados de las 5 iteraciones del entrenamiento.

Tabla 27. Evolución de los pesos

t	α_t	$regla_t$	D_t
1	0,549	*cs* ≠ *bell*	(0,25,0,25,0,25,0,25)
2	0,805	*cc* = *brown*	(0,167,0,5,0,167,0,167)
3	1,099	*cc* ≠ *yellow*	(0,999,0,3,0,099,0,499)
4	0,805	*cs* ≠ *bell*	(0,056,0,167,0,5,0,278)
5	0,805	*cc* = *brown*	(0,033,0,5,0,3,0,167)
6			(0,019,0,3,0,18,0,5)

El clasificador da una predicción correcta sobre el ejemplo que muestra la tabla 24. Para obtener la predicción tenemos que calcular:

$$\text{signo}(\textstyle\sum_{t=1}^{5} \alpha_t h_t(x)) =$$
$$= \text{signo}(0{,}549 \times (-1) + 0{,}805 \times (-1) +$$
$$+1{,}099 \times (-1) + 0{,}805 \times (-1) + 0{,}805 \times (-1)) =$$
$$= \text{signo}(-4{,}063) = -1$$

Análisis del método

Para garantizar la estabilidad del algoritmo tenemos que asegurarnos de que el error se mantenga entre 0 y 0,5. Si el error sube de esta medida, no podemos asegurar la convergencia del mismo.

Este algoritmo se ha aplicado con éxito a muchos problemas prácticos, incluidos la categorización de textos o el filtro anti spam.

Las ventajas que nos aporta este método son varias. En primer lugar, tiene pocos parámetros: el número máximo de iteraciones y la forma de las hipótesis débiles. La implementación de este algoritmo es relativamente sencilla.

Por otro lado, al finalizar el proceso, nos deja información muy útil. De las reglas generadas se puede extraer conocimiento que puede ser interpretado por humanos. Los pesos finales indican qué ejemplos no se han podido modelizar. Esto nos da información de posibles *outliers* o posibles errores de etiquetado.

El mayor inconveniente que tiene es su coste computacional. Se han dedicado esfuerzos a tratar este tema, como la versión LazyBoosting* en que en cada iteración sólo se explora un subconjunto aleatorio de las posibles reglas débiles. En este trabajo se reporta una reducción sustancial del coste computacional con una reducción mínima de los resultados.

*** G. Escudero; L. Màrquez; G. Rigau (2000). "Boosting Applied to Word Sense Disambiguation". In *Proceedings of the 12th European Conference on Machine Learning.***

El AdaBoost es un algoritmo teóricamente bien fundamentado. El error de generalización de la hipótesis final puede ser acotado en términos del error de entrenamiento. Este algoritmo está basado en la maximización del margen, lo que hace que tenga muchas similitudes con las máquinas de vectores de soporte. Estos dos conceptos estan tratados con mayor profundidad en el subapartado 4.5.

Lectura complementaria

El AdaBoost.MH fue propuesto en: R. E. Schapire; Y. Singer (2000). "Boostexter: A Boosting-based System for Text Categorization". *Machine Learning* (vol. 39, núm. 2-3)

El algoritmo básico sólo permite clasificar entre dos clases. Para tratar problemas multiclase, en lugar de binarios, tenemos que utilizar la variante AdaBoost.MH.

Implementación en Python

El programa 4.7 corresponde a la implementación en Python del AdaBoost binario básico con atributos nominales, utilizando los módulos de la librería scikit-learn de Python.

Código 4.7: AdaBoost en Python

```
1  import numpy as np
2  from sklearn.ensemble import AdaBoostClassifier
3  from sklearn.model_selection import train_test_split
4  from sklearn import preprocessing
5
6  le = preprocessing.LabelEncoder()
7
8  with open("mushroom.data.txt","r") as fopen:
9      lines = [l.strip().split(',') for l in fopen]
10
11 attrs = np.array([l[1:] for l in lines]).flatten()
12 le.fit(attrs)
13 encoded_attrs = le.transform(attrs)
14
15 X = np.reshape(encoded_attrs,(len(lines),-1))
16
17 y_labels = np.array([l[0] for l in lines])
18 le.fit(y_labels)
19 y = le.transform(y_labels)
20
21 X_train, X_test, y_train, y_test = train_test_split(X, y,
22                                                     test_size=0.33)
23
24 clf = AdaBoostClassifier()
25 clf.fit(X_train, y_train)
26
27 print("PRECISION:",clf.score(X_test,y_test))
```

Podéis encontrar la documentación relativa al paquete AdaBoostClassifier en la documentación oficial de scikit-learn*. Como se puede observar, el constructor del modelo puede invocarse sin recibir parámetros, ya que los valores que toma por defecto son útiles para la mayoría de los casos. No obstante, se pueden configurar aspectos como el número de clasificadores débiles por utilizar, o la tasa mínima de aprendizaje entre iteraciones, que pueden ajustar el modelo de manera más adecuada a vuestro problema. Es recomendable consultar a fondo la documentación oficial antes de utilizar el método.

Ved también

Los archivos de datos y el programa están disponibles en el repositorio de la asignatura.

* http://bit.ly/2mKJcwP

4.5. Clasificadores lineales y métodos basados en kernels

En este subapartado empezaremos por ver un algoritmo equivalente al clasificador lineal que se describe en el subapartado 4.3.2. En la versión de este subapartado sustituiremos la distancia euclídea por el producto escalar. Esta modificación nos permitirá introducir el uso de los kernels: una técnica muy utilizada en la última década para el tratamiento de conjuntos no lineales con algoritmos lineales. Para finalizar el subapartado, veremos el algoritmo de las máquinas de vectores de soporte.

4.5.1. Clasificador lineal basado en producto escalar

Supongamos que queremos realizar una aplicación para un almacén de flores, donde llegan a diario miles de productos. Disponemos de un sistema láser que nos suministra una serie de medidas sobre las flores y nos piden que el sistema las clasifique automáticamente para transportarlas mediante un robot a las diferentes estanterías del almacén. Las medidas que envía el sistema láser son: la longitud y la anchura del sépalo y el pétalo de cada flor.

Un clasificador lineal (binario) es un hiperplano en un espacio n-dimensional de atributos que puede ser representado por un vector de pesos w y un umbral* b, que está relacionado con la distancia del hiperplano al origen: $h(x) = \text{signo}(\langle w,x\rangle + b$, donde $\langle\cdot,\cdot\rangle$ representa el producto escalar. Cada componente del vector de pesos se corresponde con uno de los atributos.

* En inglés, *bias*.

El ejemplo más simple de clasificador lineal es el método basado en los centroides descrito en el subapartado 4.3.2. Una vez calculados los centroides p y n como el de los ejemplos con clases +1 y –1 respectivamente, podemos expresar la predicción de un nuevo ejemplo como:

$$h(x) = \begin{cases} +1, & si\ de(x,p) < de(x,n) \\ -1, & en\ otro\ caso \end{cases}$$

donde $de(x,p)$ corresponde a la distancia euclídea entre x y p.

Sabiendo que la distancia euclídea se puede expresar como $\|x-p\|^2$, la expresión anterior se puede reformular como:

$$h(x) = \text{signo}(\|x-n\|^2 - \|x-p\|^2)$$

Teniendo en cuenta que $\|a-b\|^2 = \langle a,a\rangle + \langle b,b\rangle - 2\langle a,b\rangle$, desarrollando el argumento de la función signo obtenemos:

$$\begin{aligned} &\|x-n\|^2 - \|x-p\|^2 = \\ &= \langle x,x\rangle + \langle n,n\rangle - 2\langle x,n\rangle - \langle x,x\rangle - \langle p,p\rangle + 2\langle x,p\rangle = \\ &= 2\langle x,p\rangle - 2\langle x,n\rangle - \langle p,p\rangle + \langle n,n\rangle = \\ &= 2\langle x,p-n\rangle - \langle p,p\rangle + \langle n,n\rangle \end{aligned}$$

Para finalizar este desarrollo, obtendremos la relación con w y b:

$$h(x) = \text{signo}(2\langle x,p-n\rangle - \langle p,p\rangle + \langle n,n\rangle) =$$
$$= \text{signo}(\langle x,p-n\rangle - \tfrac{1}{2}(\langle p,p\rangle - \langle n,n\rangle)) = \text{signo}(\langle w,x\rangle - b)$$

Y por tanto:

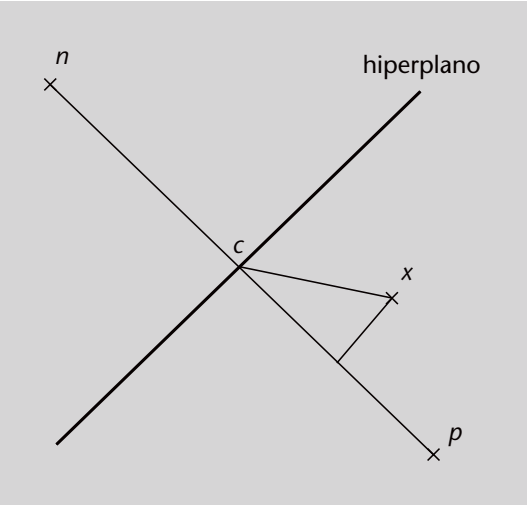
$$w = p - n$$
$$b = \tfrac{1}{2}(\langle p,p\rangle - \langle n,n\rangle).$$

Producto escalar

El producto escalar es la proyección de un vector sobre el otro. Si tenemos un punto x como el de la figura 24 y los dos centroides p para los ejemplos de la clase +1, n para los –1 y c como el punto medio entre ambos, podemos aprovechar el producto escalar para obtener la predicción. La predicción para este punto vendrá dada por el signo del producto escalar de los vectores $c \to p$ y $c \to x$:

$$h(x) = \text{signo}(\langle p-c,x-c\rangle)$$

Figura 24. Representación del producto escalar



Desarrollando esta expresión obtenemos:

$$y = \text{signo}(\langle p-c,x\rangle + \langle c,c-p\rangle) = \text{signo}(\langle w,x\rangle + b)$$

Y por tanto:

$$w = p - c = p - \frac{p+n}{2} = \frac{p-n}{2}$$

$$b = \langle c, c-p \rangle = \langle \tfrac{p+n}{2}, \tfrac{p+n}{2} - p \rangle = \langle \tfrac{p+n}{2}, \tfrac{n-p}{2} \rangle =$$
$$= \frac{\langle p,n \rangle - \langle p,p \rangle + \langle n,n \rangle - \langle p,n \rangle}{4} = \frac{\langle n,n \rangle - \langle p,p \rangle}{4}$$

La magnitud del producto escalar nos está dando una medida de la distancia del punto a clasificar al hiperplano, que se puede interpretar como una medida de la confianza de la predicción.

Fijaos en que con los dos desarrollos hemos llegado al mismo resultado*. Existe una relación directa entre las distancias euclídeas y el producto escalar.

*** Notad que multiplicar por un valor positivo no cambia el resultado de la función** signo.

Ejemplo de aplicación

La tabla 28 muestra un conjunto de entrenamiento en el que tenemos que clasificar flores a partir de sus propiedades. Los centroides se calculan a partir de los promedios de los diferentes atributos y corresponden respectivamente a: $p = (5{,}0,\ 3{,}25,\ 1{,}4,\ 0{,}2)$ y $n = (5{,}85,\ 2{,}9,\ 4{,}15,\ 1{,}35)$.

Tabla 28. Conjunto de entrenamiento

class	sepal-length	sepal-width	petal-length	petal-width
+1	5,1	3,5	1,4	0,2
+1	4,9	3,0	1,4	0,2
-1	6,1	2,9	4,7	1,4
-1	5,6	2,9	3,6	1,3

Fuente: problema «iris» del repositorio UCI (Frank y Asunción, 2010)

Una vez obtenidos los centroides, pasamos a obtener los valores del hiperplano w y b.

$$w = \frac{p-n}{2} = (-0{,}425,\ 0{,}175,\ -1{,}375,\ -0{,}575)$$

$$b = \frac{\langle n,n \rangle - \langle p,p \rangle}{4} = 6{,}02875$$

A partir de este momento, si queremos clasificar el ejemplo que muestra la tabla 29 evaluamos la fórmula:

$$y = \text{signo}(\langle w,x \rangle + b) =$$
$$= \text{signo}(\langle (-0{,}425,\ 0{,}175,\ -1{,}375,\ -0{,}575), (4{,}9,\ 3{,}1,\ 1{,}5,\ 0{,}1) \rangle + 6{,}02875) =$$
$$= \text{signo}(2{,}36875) = +1$$

Tabla 29. Ejemplo de test

class	sepal-length	sepal-width	petal-length	petal-width
+1	4,9	3,1	1,5	0,1

Fuente: problema «iris» del repositorio UCI (Frank y Asunción, 2010)

Análisis del método

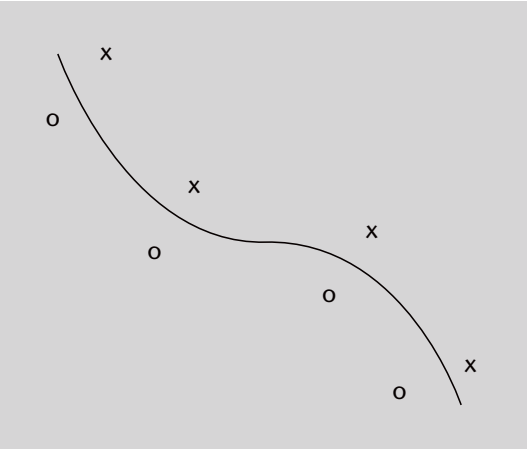
Este tipo de método ha sido muy utilizado en el área de la recuperación de la información*, ya que es una manera simple, eficiente y efectiva de construir modelos para la categorización de textos. Hay muchos algoritmos para entrenar este tipo de clasificadores (perceptrón, Widrow-Hoff, Winnow...). Un algoritmo de esta familia ha ganado mucha importancia en la última década: las máquinas de vectores de soporte. En el apartado 4.5.4 estudiaremos a fondo esta técnica y veremos ejemplos prácticos.

*** En inglés, *information retrieval*.**

4.5.2. Clasificador lineal con kernel

Los algoritmos lineales como el anterior están pensados para clasificar conjuntos linealmente separables. En el caso de un espacio de atributos que se pueda representar en el plano, el hiperplano correspondería a una linea recta. En muchos problemas reales, no se suele dar esta circunstancia; son conjuntos para los que no existen hiperplanos lineales que los puedan separar. La figura 25 muestra un ejemplo de uno de estos casos. Se han dedicado muchos esfuerzos en la literatura a crear algoritmos no lineales para abordar este tipo de problemas. El gran inconveniente que plantean la mayoría de estos métodos es su elevado coste computacional.

Figura 25. Ejemplo de conjunto no linealmente separable



Lecturas complementarias

J. Shawe-Taylor; N. Cristianini (2004). *Kernel Methods for Pattern Analysis*. UK: Cambridge University Press.

En las dos últimas décadas se llevan usando con mucha efectividad los así llamados *kernels* (o funciones núcleo). El uso de los mismos permite atacar problemas no lineales con algoritmos lineales. La estrategia que se sigue consiste en proyectar los datos a otros espacios de atributos en los que sí que exista

Ved también

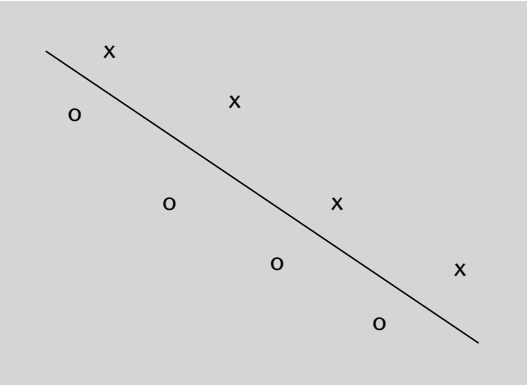
En el subapartado 4.5.4 se da una interpretación geométrica de los kernels y su uso.

un hiperplano lineal que los pueda separar*. Con esto conseguimos el coste computacional de los algoritmos lineales con sólo un pequeño incremento que conlleva la proyección de los datos a otro espacio de atributos.

*** En la literatura se le suele citar como el truco o estratagema del kernel, en inglés *kernel trick.***

Así, si conocemos una función ϕ que proyecte los puntos de la figura 25 a un espacio de atributos como el que muestra la figura 26, podremos encontrar un hiperplano lineal en el nuevo espacio que separe los dos tipos de puntos.

Figura 26. Ejemplo de proyección con cierta función ϕ de los puntos de la figura 25



De hecho, no se trabaja con las funciones tipo ϕ, sino con las así llamadas funciones kernel, $\kappa(x,z) = \langle\phi(x),\phi(z)\rangle$. Cuando se trabaja con kernels se suele utilizar una estructura modular en la que primero se proyectan los puntos (ejemplos de entrenamiento) dos a dos con la función κ, creando así una matriz cuadrada. A esta matriz se la conoce como matriz kernel*. Los algoritmos de aprendizaje pueden soler actuar sobre el kernel κ o sobre esta matriz directamente. La notación estándar para mostrar las matrices kernel es:

*** En inglés, *kernel matrix*.**

$$\begin{pmatrix} \kappa(x_1,x_1) & \kappa(x_1,x_2) & \cdots & \kappa(x_1,x_l) \\ \kappa(x_2,x_1) & \kappa(x_2,x_2) & \cdots & \kappa(x_1,x_l) \\ \vdots & \vdots & \ddots & \vdots \\ \kappa(x_l,x_1) & \kappa(x_l,x_2) & \cdots & \kappa(x_l,x_l), \end{pmatrix}$$

donde l corresponde al número de ejemplos de entrenamiento. Las matrices kernel han de cumplir que sean simétricas y semidefinidas positivamente. De manera equivalente, su función kernel asociada tiene que cumplir la misma propiedad.

Matriz semidefinida positiva

Se dice que una matriz simétrica es semidefinida positiva cuando todos sus valores propios son superiores o iguales a cero.

A partir de este momento vamos a desarrollar la formulación teniendo en cuenta que conocemos una supuesta función kernel κ, pero no conocemos su función ϕ relativa. Como en el primer desarrollo del algoritmo anterior, supongamos que p y n corresponden a los centroides de los ejemplos con

clases +1 y –1 respectivamente, y $|p|$ y $|n|$ al número de ejemplos de las clases +1 y –1 respectivamente. La predicción de un nuevo ejemplo x viene dada por:

$$h(x) = \text{signo}(\|\phi(x) - \phi(n)\|^2 - \|\phi(x) - \phi(p)\|^2).$$

Para continuar desarrollando esta expresión tenemos que tener en cuenta que $\kappa(n,n) = \langle\phi(n),\phi(n)\rangle$ es equivalente al promedio de las proyecciones dos a dos de los elementos con clase –1. Con p y los ejemplos de la clase +1 funciona de forma similar. Si desarrollamos el argumento de la función signo teniendo en cuenta estas observaciones, obtenemos:

$$\begin{gathered}
\|\phi(x) - \phi(n)\|^2 - \|\phi(x) - \phi(p)\|^2 = \\
\kappa(x,x) + \frac{1}{|n|^2}\sum_{\{i|y_i=-1\}}\sum_{\{j|y_j=-1\}}\kappa(x_i,x_j) - \frac{2}{|n|}\sum_{\{i|y_i=-1\}}\kappa(x,x_i) - \\
-\kappa(x,x) - \frac{1}{|p|^2}\sum_{\{i|y_i=+1\}}\sum_{\{j|y_j=+1\}}\kappa(x_i,x_j) + \frac{2}{|p|}\sum_{\{i|y_i=+1\}}\kappa(x,x_i) = \\
= \frac{2}{|p|}\sum_{\{i|y_i=+1\}}\kappa(x,x_i) - \frac{2}{|n|}\sum_{\{i|y_i=-1\}}\kappa(x,x_i) - \\
-\frac{1}{|p|^2}\sum_{\{i|y_i=+1\}}\sum_{\{j|y_j=+1\}}\kappa(x_i,x_j) + \frac{1}{|n|^2}\sum_{\{i|y_i=-1\}}\sum_{\{j|y_j=-1\}}\kappa(x_i,x_j).
\end{gathered}$$

Y de aquí obtenemos:

$$b = \frac{1}{2}\left(\frac{1}{|p|^2}\sum_{\{i|y_i=+1\}}\sum_{\{j|y_j=+1\}}\kappa(x_i,x_j) - \frac{1}{|n|^2}\sum_{\{i|y_i=-1\}}\sum_{\{j|y_j=-1\}}\kappa(x_i,x_j)\right)$$

$$h(x) = \text{signo}\left(\frac{1}{|p|}\sum_{\{i|y_i=+1\}}\kappa(x,x_i) - \frac{1}{|n|}\sum_{\{i|y_i=-1\}}\kappa(x,x_i) - b\right).$$

Notad que en estas expresiones resultantes no aparece ϕ por ningún lado. Esta forma en la que no aparece ϕ recibe el nombre de *formulación dual*, en contraposición a la *formulación primal*. Otra cuestión a tener en cuenta de cara a la implementación es que la b es una constante que se tiene que calcular una sola vez, ya que no aparece el ejemplo de test. No tendremos que recalcularlo para cada uno de los ejemplos de test.

Construcción de kernels

Hay un conjunto de reglas llamadas propiedades de la clausura que nos ayudan en la construcción de nuevos kernels. Si tenemos que κ_1 y κ_2 son kernels sobre $X \times X$, $X \subseteq \mathbb{R}^n$, $a \in \mathbb{R}^+$, $f(\cdot)$ una función real sobre X, $\phi : X \leftarrow \mathbb{R}^N$ con κ_3 un kernel sobre $\mathbb{R}^N \times \mathbb{R}^N$, y B una matriz simétrica semidefinida positiva de $n \times n$, las siguientes funciones son kernels:

1) $\kappa(x,z) = \kappa_1(x,z) + \kappa_2(x,z)$

2) $\kappa(x,z) = a\kappa_1(x,z)$

3) $\kappa(x,z) = \kappa_1(x,z)\kappa_2(x,z)$

4) $\kappa(x,z) = f(x)f(z)$

5) $\kappa(x,z) = \kappa_3(\phi(x),\phi(z))$

6) $\kappa(x,z) = x'Bz$

Kernels habituales

Hay dos kernels no lineales de aplicación general que aparecen en todas las implementaciones de métodos basados en kernels. A continuación damos su caracterización.

Si $\kappa_1(x,z)$ es un kernel sobre $X \times X$, donde $x,z \in X$, y $p(x)$ es un polinomio con coeficientes positivos y $\sigma > 0$, entonces las siguientes funciones también son kernels:

1) $\kappa(x,z) = p(\kappa_1(x,z))$

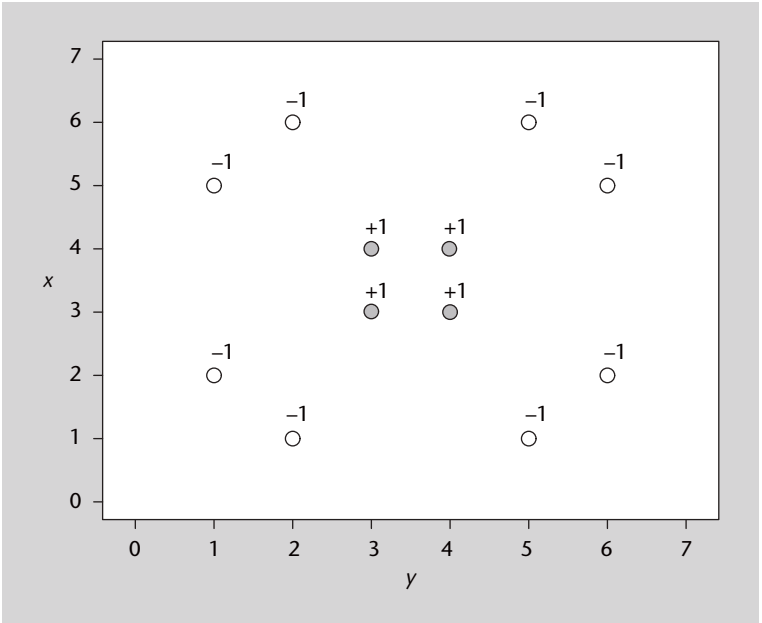
2) $\kappa(x,z) = \exp\left(-\frac{\|x-z\|^2}{2\sigma^2}\right)$

El primer kernel es conocido como *kernel polinómico* y el segundo como *kernel gaussiano* o *RBF kernel*.

Ejemplo de aplicación

El conjunto de datos que muestra la figura 27 no es linealmente separable. Si calculamos los centroides y aplicamos el conjunto de *training* como conjunto de test, logramos una precisión del 66 %. ¿Cómo podemos separar este conjunto?

Figura 27. Ejemplo de conjunto linealmente no separable



Este conjunto de datos se puede aprender con un kernel radial

$$\kappa(x,z) = \exp\left(-\frac{\|x-z\|^2}{2\sigma^2}\right)$$

que en ocasiones aparece como $\kappa(x,z) = \exp\left(-\gamma\|x-z\|^2\right)$.

Si escogemos como conjunto de entrenamiento los puntos de la figura 27, como ejemplo de test el punto $x_t = (3,6)$ y un kernel radial con $\gamma = 1$, obtenemos:

$$b = 0{,}1629205972066024$$

$$h(x_t) = \text{signo}(-0{,}2057373019348639) = -1$$

La predicción de –1 es coherente con la posición del punto (3,6) en la figura 27.

Análisis del método

La definición de kernels está a medio camino entre la representación de la información y el algoritmo de aprendizaje. Los kernels están muy relacionados con el espacio de atributos de los problemas. Es decir, no podemos decir que exista un kernel mejor que otro. Cada kernel funciona con los problemas a los que mejor se adapta por sus características intrínsecas. Para el problema de las flores que muestra la tabla 28, funciona mejor el kernel lineal que el kernel radial, y para el que muestra la figura 27, pasa exactamente lo contrario. En el apartado 4.5.4 estudiaremos a fondo esta técnica y veremos ejemplos prácticos aplicando estos kernels en máquinas de vectores de soporte.

El uso de kernels nos permite adaptar los algoritmos a espacios de atributos más complejos o específicos al problema. Con ellos se ha logrado mejorar el comportamiento de los algoritmos de aprendizaje para muchos problemas, abordar problemas que hasta el momento no tenían solución e ir más allá y crear nuevos algoritmos que tratan datos estructurados, como pueden ser los análisis sintácticos del lenguaje natural o el tratamiento de vídeos. Además, el acceso a estos espacios de atributos complejos y flexibles se consigue con un coste computacional (en espacio y tiempo) relativamente bajo.

Una de las grandes ventajas de estos métodos, además de su flexibilidad, es que están teóricamente muy bien fundamentados y se han demostrado características como su eficiencia computacional, robustez y estabilidad estadística.

En la bibliografía se habla del cuello de botella de la matriz kernel como del límite que impone el trabajar con ella para todo. Todo lo que queramos representar de los problemas se ha de poder expresar en forma de las funciones

kernel y por tanto de la matriz kernel. En algunos casos podemos encontrar limitaciones.

El uso de kernels no está acotado a los algoritmos de clasificación. Esta misma técnica se ha utilizado en métodos *clustering* como los que se describen en el subapartado 2.3 o en los algoritmos de selección de atributos descritos en el apartado 3. Así, existen algoritmos como el kernel PCA o el kernel *k-means*. Los algoritmos como el kernel PCA o el kernel CCA se están utilizando en problemas de visión como el reconocimiento de caras.

Una de las mayores ventajas del uso de kernels (funciones núcleo) es el diseño de kernels específicos para tareas concretas. En los últimos tiempos se está investigando mucho en la creación de kernels para tratar conjuntos, cadenas de texto, grafos o árboles, entre otros*. Algunos de estos kernels son los específicos para tratar texto**, imágenes o vídeos***.

* J. Shawe-Taylor; N. Cristianini (2004). *Kernel Methods for Pattern Analysis.* Reino Unido: Cambridge University Press.

** T. Joachims (2001). *Learning to Classify Text Using Support Vector Machines.* EE. UU.: Kluwer Academic Publishers.

*** F. Camastra; A. Vinciarelli (2008). *Machine Learning for Audio, Image and Video Analysis: Theory and Applications. (Advanced Information and Knowledge Processing).* Springer.

4.5.3. Kernels para tratamiento de textos

En este subapartado vamos a profundizar en uno de los kernels específicos, el kernel para el tratamiento de textos que nos servirá para abordar el problema de la categorización de textos.

Supongamos que queremos diseñar un clasificador para una agencia de noticias que sepa distinguir los documentos relacionados con adquisiciones corporativas.

El primer paso que tenemos que seguir para poder abordar este problema es pensar en la codificación de los datos. Una de las formas más simples para representar texto es mediante un conjunto de palabras*. A cada palabra se le puede asignar si aparece o no en el documento, el número de veces que aparece o la frecuencia relativa de la palabra en el documento. La frecuencia relativa evita el sesgo que el tamaño de los ejemplos (documentos) produce en los resultados.

* En inglés, *bag of words.*

En el conjunto de datos de que disponemos*, ya está preprocesado y el vector contiene la información de las frecuencias relativas. La tabla 30 muestra la primera parte del contenido del primer ejemplo de entrenamiento. El primer valor corresponde a la clase, que puede ser positiva o negativa. En este caso está indicando que el ejemplo pertenece a las adquisiciones corporativas. Las siguientes columnas corresponden a parejas *atributo* : *valor*. Así 6 : 0,0198 nos dice que el atributo número 6 tiene el valor 0,0198. Los atributos están ordenados y aquellos que no aparecen quiere decir que son valores nulos. A este tipo de representación de los atributos se le denomina representación dispersa**, en contraposición a la representación densa. En un archivo separado encontramos un índice de palabras donde podemos comprobar que el atributo 6

* Este conjunto de datos es un subconjunto del Reuters 21578 procesado por T. Joachims, que está disponble en: http://svmlight.joachims.org

** En inglés, *sparse.*

corresponde a la aparición de la palabra *said*. Los conjuntos de entrenamiento y test están separados en dos archivos: *train.dat* y *test.dat*.

Tabla 30. Ejemplo de representación de un documento

+1 6:0.0198403253586671 15:0.0339873732306071 29:0.0360280968798065 ...

Fuente: problema de «Text Categorisation» formateado por T. Joachims

A partir de aquí, podemos definir que la representación de un documento d queda de la forma:

$$\phi(d) = (tf(t_1,d), tf(t_2,d), \ldots, tf(t_N,d)) \in \mathbb{R}^N$$

donde t_i corresponde al término o palabra i, N al número de términos y $tf(t_i,d)$ a la frecuencia relativa del término t_i en el documento d.

Partiendo de esta definición, el conjunto de entrenamiento lo podemos expresar mediante la matriz siguiente:

$$D = \begin{pmatrix} tf(t_1,d_1) & tf(t_2,d_1) & \cdots & tf(t_N,d_1) \\ tf(t_1,d_2) & tf(t_2,d_2) & \cdots & tf(t_N,d_2) \\ \vdots & \vdots & \ddots & \vdots \\ tf(t_1,d_l) & tf(t_2,d_l) & \cdots & tf(t_N,d_l) \end{pmatrix}$$

donde cada fila corresponde a un documento y cada columna a la aparición de un término en todos los documentos.

A partir de aquí se pueden definir diferentes tipos de funciones kernel. La primera aproximación a una función de kernel es una de las que se usa para tratar conjuntos:

$$\kappa(d_1,d_2) = 2^{|A_1 \bigcap A_2|}$$

donde A_i corresponde al conjunto formado por las palabras de d_i.

En categorización de textos se suele utilizar la composición de kernels para ir creando kernels más sofisticados. Para ver un ejemplo de ello, vamos a ver cómo se combinaría el kernel anterior con el RBF estudiado anteriormente.

Si partimos de los kernels:

$$\kappa_2(d_1,d_2) = 2^{|A_1 \bigcap A_2|}$$

$$\kappa_1(x,z) = \exp(-\gamma\|x,z\|)$$

y de que

$$\|\phi_1(x) - \phi_1(z)\|^2 = \kappa_1(x,x) - 2\kappa_1(x,z) + \kappa_1(z,z),$$

el kernel resultado de su combinación es:

$$\kappa(d_1,d_2) = \exp(-\gamma(\kappa_2(d_1,d_1) - 2\kappa_2(d_1,d_2) + \kappa_2(d_2,d_2))).$$

Una segunda aproximación para abordar la creación de kernels para categorización de textos más utilizada que la de conjuntos es la «kernealización» del VSM*. Este kernel, a partir de la definición anterior de la matriz D, se formula como sigue:

*** Del inglés, *vector space model*. El kernel asociado recibe el nombre de *vector space kernel*.**

$$\kappa(d_1,d_2) = \sum_{i=1}^{N} tf(t_i,d_1)tf(t_i,d_2)$$

Ampliaciones de kernels para texto

Cuando se trabaja en clasificación de textos se suelen preprocesar los documentos. Uno de los procesos más comunes es eliminar las palabras sin contenido semántico como: preposiciones, artículos... Existen listas de estas palabras que nos ayudan a realizar esta tarea.

Repositorio

Tenéis disponible en el repositorio de la asignatura listas de palabras sin contenido semántico para inglés, castellano y catalán.

El segundo proceso que se suele utilizar es eliminar del diccionario aquellas palabras que no aparecen más de dos o tres veces en el conjunto de todos los documentos del corpus.

Diccionario

Se conoce como diccionario el archivo índice de palabras que aparecen en los documentos del conjunto de datos.

Con esto conseguimos reducir bastante el espacio de atributos y con ello mejorar la eficiencia computacional de los algoritmos esperando no degradar la precisión de los mismos.

En el segundo ejemplo de clasificación de textos que hemos visto no hemos tenido en cuenta el tratamiento del tamaño de los documentos. Esto va a sesgar el comportamiento de las predicciones hacia los documentos más grandes, ya que contienen más palabras y tienen más probabilidades de enlazarse entre ellos. Otra forma de tratar este tema, aparte de codificar frecuencias como en el primer problema tratado, es normalizar el kernel resultante. Esto se realiza como etapa final de la combinación de kernels a partir de la formulación siguiente:

$$\hat{\kappa}(x,y) = \langle \frac{\phi(x)}{\|\phi(x)\|}, \frac{\phi(x)}{\|\phi(x)\|} \rangle = \frac{k(x,y)}{\sqrt{k(x,x)k(y,y)}}$$

Para finalizar este subapartado vamos a describir una función kernel basada en uno de los algoritmos más utilizados de la recuperación de la información, el *tf-idf*. La idea que persigue este algoritmo es doble: por un lado aporta una ponderación de los términos o las palabras en función de su importancia; y por otro, aporta medidas de relación entre los términos, que nos ayudan a relacionar documentos que no comparten exactamente los mismos términos pero sí sinónimos.

tf-idf

scikit-learn 0.19.0 implementa el algoritmo *tf-idf* en el módulo *TfidfTransformer* de *sklearn.feature_extraction.text*.

Vamos a empezar retomando la definición de función kernel del VSM:

$$k(d_1,d_2) = \langle \phi(d_1),\phi(d_2)\rangle = \sum_{i=1}^{N} tf(t_i,d_1)tf(t_i,d_2)$$

Si queremos ponderar los términos con un peso, necesitamos una función $w(t)$ que nos lo proporcione. Existen diferentes alternativas, como pueden ser la información mutua o medidas basadas en la entropía. El *tf-idf* utiliza la función:

Ved también

Para saber más sobre la información mutua podéis ver el subapartado 3.1.3. Para saber más sobre medidas basadas en la entropía podéis ver el subapartado 4.4.1.

$$w(t) = \ln\left(\frac{l}{df(t)}\right)$$

donde l corresponde al número total de documentos y $df(t)$ al número de documentos que contienen el término t.

Si generamos un kernel a partir de esta función obtendremos:

$$\tilde{\kappa}(d_1,d_2) = \sum_{t} w(t)^2 tf(t,d_1)tf(t,d_2)$$

Si queremos añadir semejanzas semánticas entre términos, existen diversas formas de hacerlo. Una es explotar una ontología de libre distribución muy utilizada en el campo del procesamiento del lenguaje natural que se denomina WordNet*. Esta ontología contiene las diferentes palabras de la lengua organizadas en una jerarquía en la que cada nodo representa un concepto y contiene todas las palabras sinónimas que lo representan. Una palabra está en los diferentes nodos a los que pertenece en función de sus diferentes significados. Los nodos están interconectados por medio de la relación de hiperonimia/hiponimía**. Partiendo de esta ontología, la semejanza entre palabras tiene que ver con la inversa del camino más corto para llegar de una palabra a la otra.

***http://wordnet.princeton.edu**

**** A modo de ejemplo: felino es un hiperónimo de gato y gato es hipónimo de felino.**

En recuperación de información se juega con la matriz D para obtener un resultado parecido. Recordemos que la matriz de kernel se puede generar a partir de:

$$K = D \times D'$$

y que:

$$D = \begin{pmatrix} tf(t_1,d_1) & tf(t_2,d_1) & \cdots & tf(t_N,d_1) \\ tf(t_1,d_2) & tf(t_2,d_2) & \cdots & tf(t_N,d_2) \\ \vdots & \vdots & \ddots & \vdots \\ tf(t_1,d_l) & tf(t_2,d_l) & \cdots & tf(t_N,d_l) \end{pmatrix}$$

Definimos que dos términos están relacionados entre sí, si aparecen frecuentemente juntos en los mismos documentos. Esto lo podemos conseguir a partir de:

$$D' \times D$$

Fijaos en que la matriz $D'D$ tendrá un valor diferente de cero en la posición (i,j) si, y solo si, existe algún documento en el corpus en el que concurren los términos t_i y t_j.

A partir de esto, podemos definir un nuevo kernel:

$$\tilde{\kappa}(d_1,d_2) = \phi(d_1)D'D\phi(d_2)'$$

donde

$$(D'D)_{i,j} = \sum_d tf(i,d)tf(j,d)$$

Al uso de estas semejanzas semánticas se le conoce como *GVSM*, donde la *G* viene de «generalizado».

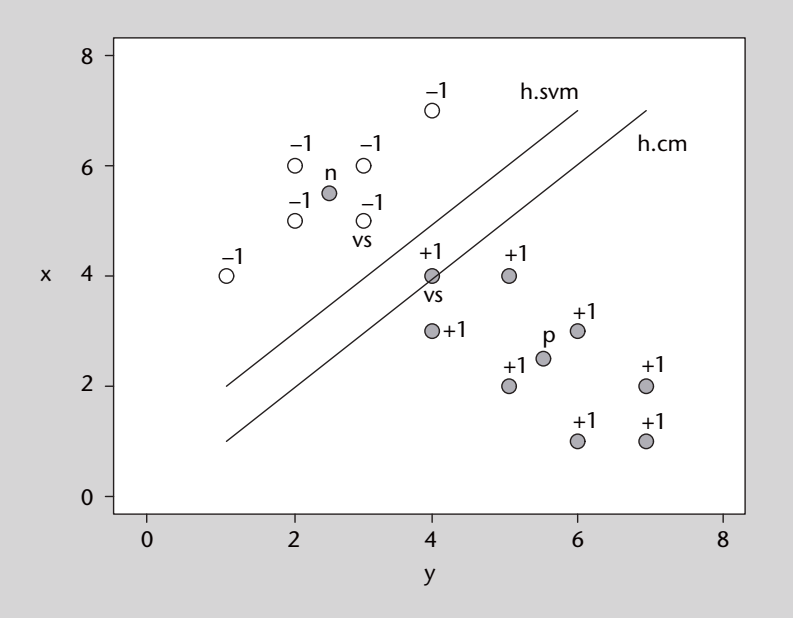
4.5.4. Máquinas de vectores de soporte

Las máquinas de vectores de soporte* (SVM) son un algoritmo que mejora el clasificador lineal buscando un mejor hiperplano que el que se genera con el clasificador lineal.

*** En inglés, *support vector machines*: N. Cristianini; J. Shawe-Taylor (2000). *An Introduction to Support Vector Machines* (SVMs). Reino Unido: Cambridge University Press.**

La figura 28 ilustra este concepto: los puntos marcados con +1 corresponden a los ejemplos positivos, los –1 a los negativos, *p* al centroide de los positivos, *n* al de los negativos y *h.cm* al hiperplano generado por los centroides (está ubicado a medio camino de los dos centroides).

Figura 28. Ejemplos de hiperplanos lineales y de las máquinas de vectores de soporte



Las máquinas de vectores de soporte son un algoritmo de optimización (como los descritos en el apartado 5) que escoge el hiperplano con margen máximo de entre todos los posibles hiperplanos que separan los ejemplos positivos de los negativos (el que tiene la misma distancia a los ejemplos positivos que a los negativos). Describe el hiperplano a partir de los llamados vectores de soporte. Estos suelen ser los puntos más cercanos al hiperplano y los que lo definen. En la figura 28, *h.svm* corresponde al hiperplano de margen máximo que encontrarían las SVM para este conjunto y los vectores de soporte están marcados con *vs*.

Interpretación geométrica

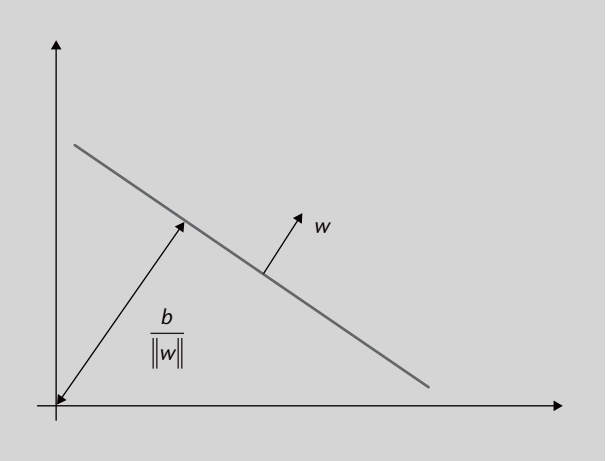
Este algoritmo combina la maximización del margen, como el AdaBoost, con el uso de los kernels descritos en el subapartado anterior y la posibilidad de flexibilizar el margen. Este subapartado pretende exponer estos conceptos de manera visual.

Muchos métodos de aprendizaje se basan en la construcción de hiperplanos. Un hiperplano es una figura geométrica que tiene una dimensión menos que el espacio en el que está ubicado. Ejemplos de hiperplanos son líneas en $\mathbb{R}^2$ o planos en $\mathbb{R}^3$. La figura 29 nos muestra un ejemplo de hiperplano en $\mathbb{R}^2$. Un hiperplano queda definido por un vector de pesos (w) y un umbral (b).

Consideremos ahora un ejemplo en $\mathbb{R}^2$. Tenemos que construir un clasificador para separar dos tipos de puntos. La figura 30 muestra tres hiperplanos válidos (en este caso líneas) que nos separarían los dos conjuntos de puntos.

Podríamos encontrar infinitos hiperplanos para este problema de clasificación. Una vez hemos escogido un hiperplano para clasificar un conjunto de

Figura 29. Ejemplo de hiperplano

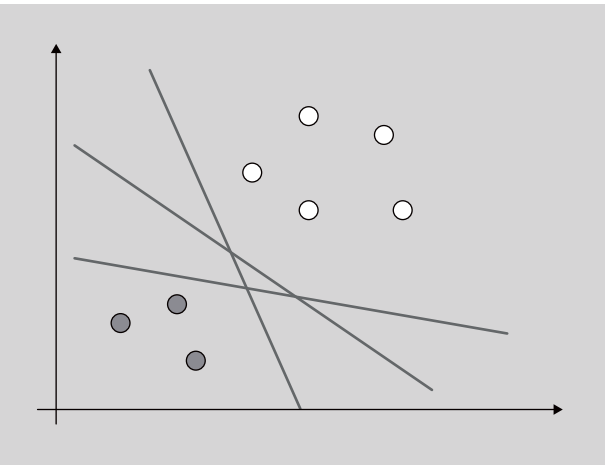


puntos, podemos crear una regla de clasificación a partir de él. Esta regla queda definida por la ecuación:

$$h(x) = signo\left(b + \sum_{i=1}^{N} w_i \cdot x_i\right)$$

donde N es el número de atributos y el vector $w = \{w_1, \ldots, w_N\}$ y b definen al hiperplano y $x = \{x_1, \ldots, x_N\}$ al ejemplo por clasificar.

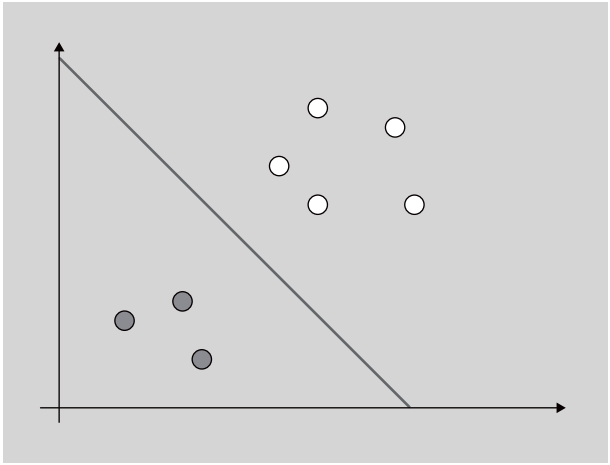
Figura 30. Ejemplo de hiperplanos posibles



Pero ¿cuál de estos infinitos hiperplanos nos servirá mejor como clasificador? Intuitivamente se puede decir que la mejor recta para dividir estos conjuntos sería algo parecido al de la figura 31.

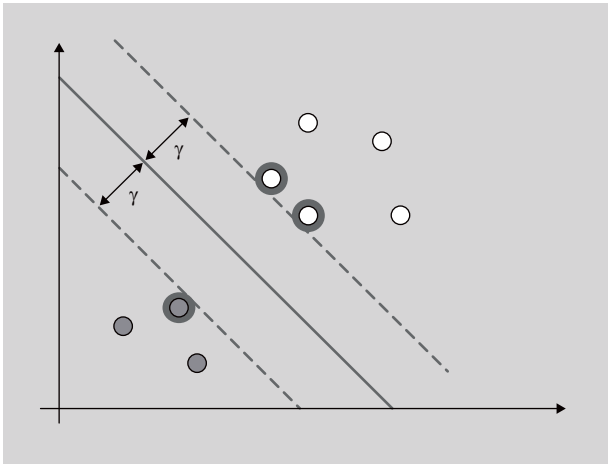
¿Por qué? Porque es la que deja más distancia a todos los puntos; es la recta que pasa más alejada de los puntos de las dos clases. A este concepto se le denomina margen. La figura 32 nos muestra la representación geométrica de este concepto; donde γ corresponde al margen y los puntos marcados a los vectores de soporte. Los vectores de soporte son aquellos puntos que están rozando el margen.

Figura 31. Mejor hiperplano



Las máquinas de vectores de soporte son un método de aprendizaje que se basa en la maximización del margen. ¿Cómo lo hace? La estrategia del método consiste en buscar los vectores de soporte. Este mecanismo de búsqueda parte de un espacio de hipótesis de funciones lineales en un espacio de atributos altamente multidimensional. En él se entrena con un algoritmo de la teoría de la optimización derivado de la teoría del aprendizaje estadístico. El subapartado siguiente describe este proceso formalmente. Este apartado está orientado a ver sus aplicaciones prácticas.

Figura 32. Margen



A partir de este momento vamos a utilizar el software svm-toy para ilustrar todos los conceptos. Este programa aplica las SVM con diferentes parámetros a un conjunto con dos atributos y representa gráficamente el resultado de la clasificación.

svm-toy

svm-toy es un programa incluido en la libsvm: Chih-Chung Chang; Chih-Jen Lin (2001). *LIBSVM: a library for support vector machines*. http://www.csie.ntu.edu.tw/~cjlin/libsvm/

Vamos a empezar con el caso anterior. Introduciremos una distribución parecida a la de la figura 32 (como muestra la figura 33). Los puntos se introducen con el botón del ratón. Hay que pulsar el botón de Change para cambiar de clase. Una vez introducidos los puntos, hay que especificar los parámetros. Para entrenar este caso especificaremos: $-t\ 0\ -c\ 100$. Es decir, un kernel lineal

Repositorio

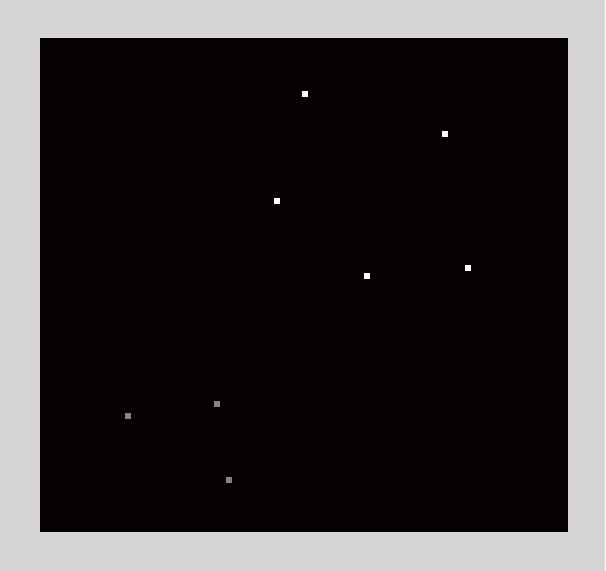
El software svm-toy está disponible en el repositorio de la asignatura.

(–*t* 0) y un valor de 100 a la rigidez (inversa de la flexibilidad) del margen. Este último concepto lo veremos más adelante.

Kernel lineal

Un kernel lineal es del estilo $u' \times v$ donde u' es nuestro vector de atributos y v es otro vector.

Figura 33. Configuración inicial para el svm-toy



Una vez establecidos los parámetros, pulsamos Run y nos aparecerá una pantalla como la que muestra la figura 34. Podemos observar cómo ha separado bien los dos conjuntos de puntos. Se puede intuir un margen claro y que los vectores de soporte son los tres puntos más cercanos al hiperplano (dos blancos y uno gris).

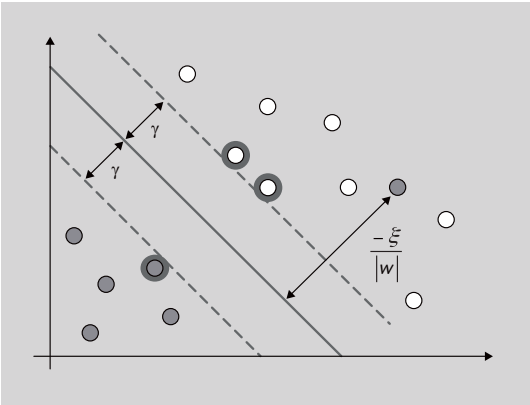
Figura 34. Kernel lineal con svm-toy

Cuando dado un problema de puntos podemos encontrar un hiperplano lineal (en el caso de $\mathbb{R}^2$, una recta) que divide perfectamente los dos conjuntos de puntos, tenemos un problema al que denominamos linealmente separable. El caso anterior cumple esta propiedad. Puede ocurrir (más bien suele ocurrir) que los conjuntos de datos no sean tan claramente separables. En estos casos nos solemos encontrar con dos tipologías de problemas:

- El primer problema aparece cuando hay puntos sueltos de las dos clases en la zona del margen, o puntos de una clase en la zona de la otra. El caso mostrado en el figura 35 no es linealmente separable. El punto gris que está entre los blancos normalmente es un error o un *outlier*. La solución pasa por generar un margen flexible*. El margen flexible admite errores en la separación a cambio de aumentar el margen. Estos errores están ponderados por la expresión $-\xi/|w|$. El uso de márgenes flexibles normalmente aumenta la generalización y disminuye la varianza del modelo. El parámetro $-c$ del svm-toy codifica la rigidez del margen. Cuanto menor sea el valor de este parámetro más flexible es el margen. O dicho de otro modo, la flexibilidad del margen es inversamente proporcional al valor del parámetro c. Este parámetro se escoge normalmente como una potencia de diez ($10^{\pm x}$).

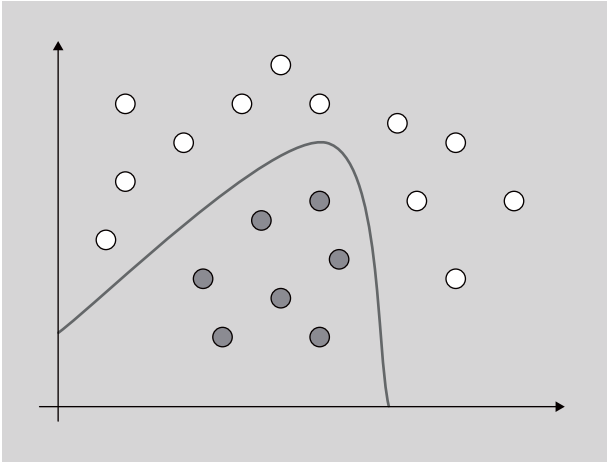
* En inglés, *soft margin*.

Figura 35. Margen flexible



- La segunda tipología de problemas aparece cuando los puntos no tienen una distribución que se pueda separar linealmente. Es decir, la estructura del problema sigue otro tipo de distribuciones. Para un problema como el que muestra la figura 36 nos iría bien una curva en lugar de una recta para separar los conjuntos de puntos. Esto se logra cambiando de kernel. Un kernel es una función de proyección entre espacios. Este mecanismo nos permite aumentar la complejidad del modelo, y por tanto la varianza.

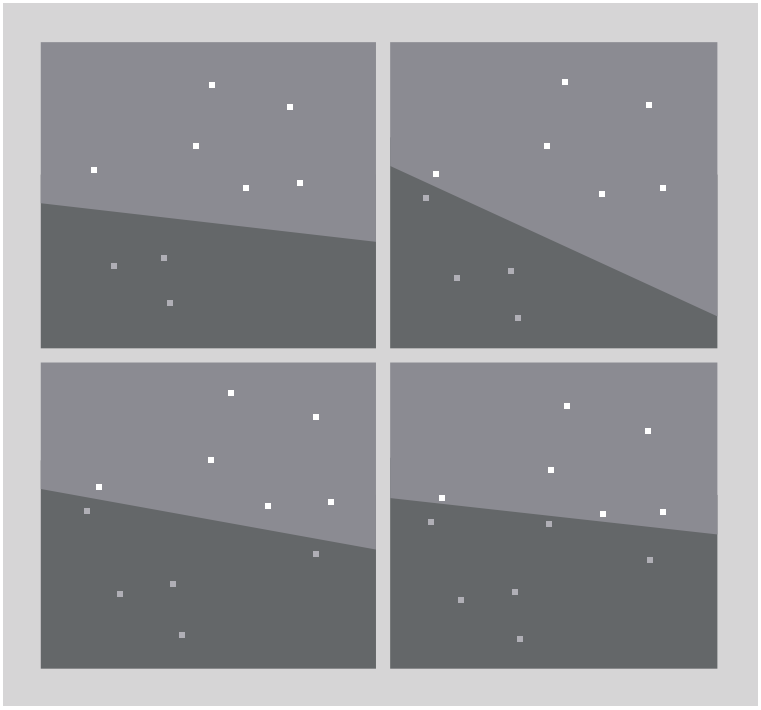
Figura 36. Conjunto linealmente no separable



De cara a construir modelos con las SVM, es muy interesante estudiar el comportamiento de estos dos parámetros: los kernels y los márgenes flexibles.

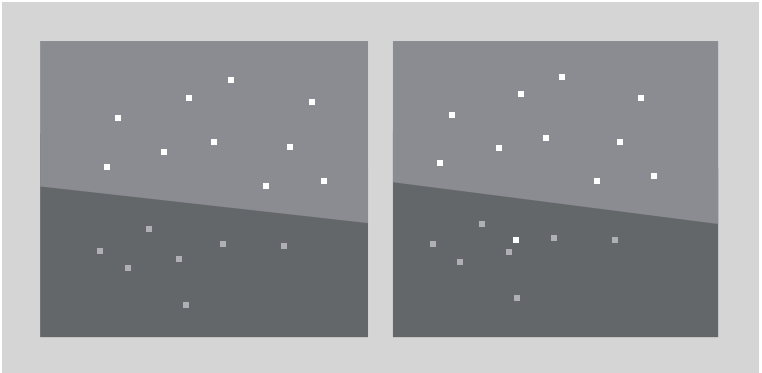
Cojamos ahora el conjunto de la figura 34 y empecemos a añadir puntos. Fijaos en cómo van cambiando el hiperplano y el margen en las imágenes de la figura 37 (tendremos que aumentar la rigidez del hiperplano escogiendo un valor de cien mil para el parámetro c). En la última imagen de la figura 37, la distancia de los vectores de soporte al hiperplano es muy inferior a la de la primera.

Figura 37. Margen flexible con svm-toy



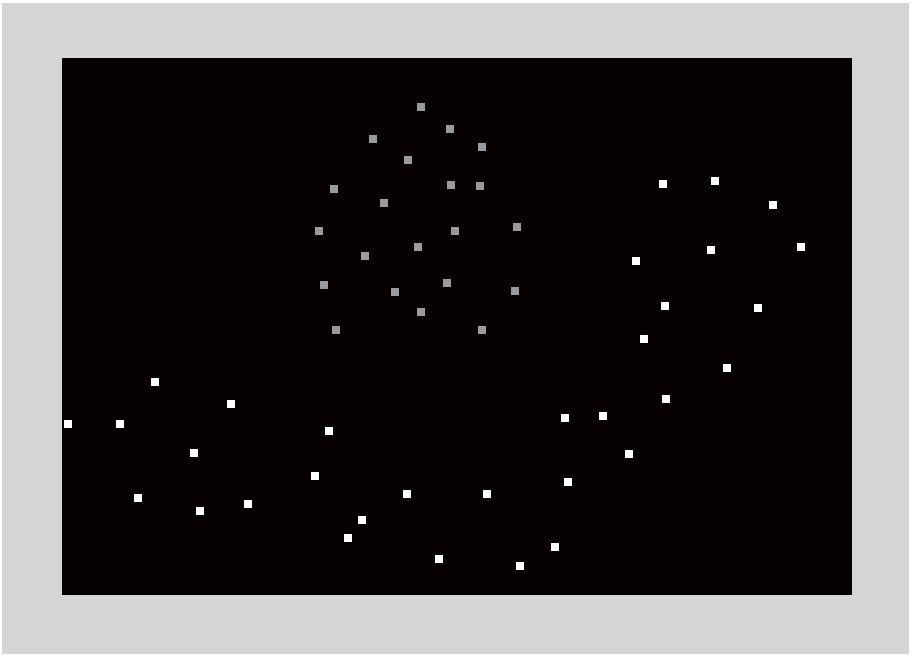
Fijaos ahora en la figura 38. En la segunda imagen tenemos un punto blanco en la zona de los grises. Gracias a haber creado un margen flexible, lo hemos dejado mal clasificado, pero tenemos un margen igual que el anterior. Las ventajas que obtenemos tienen que ver con el error de generalización. Seguramente este caso lo encontraríamos como un error en el conjunto de entrenamiento.

Figura 38. Margen flexible con svm-toy



En este punto, vamos a ver ejemplos de la aplicación de kernels. Para empezar, fijaos en el conjunto de puntos de la figura 39. ¿Qué hiperplano puede separar estos conjuntos de puntos? Un hiperplano lineal no los separaría. La aplicación de márgenes flexibles dejaría demasiados puntos mal clasificados, y aunque fueran pocos, una línea recta no es un modelo correcto para este problema. La solución para este caso pasa por utilizar un kernel para realizar una proyección a otro espacio.

Figura 39. Puntos para kernels con svm-toy



En la figura 40 tenéis dos aproximaciones con diferentes kernels polinómicos para separar estos conjuntos de puntos y, en la figura 41, dos con kernels radiales. Sus parámetros exactos son, respectivamente:

- polinómico de grado dos (*–t* 1 *–d* 2 *–c* 10000),
- polinómico de grado cuatro (*–t* 1 *–d* 4 *–c* 10000),
- radial con gamma 50 (*–t* 2 *–g* 50 *–c* 10000),
- radial con gamma 1000 (*–t* 2 *–g* 1000 *–c* 10000).

Kernel polinómico

Un kernel polinómico es del estilo: $(\gamma \times u' \times v + coef0)^{grado}$ donde u' es nuestro vector de atributos, v es otro vector y los demás son valores reales. Se le pueden pasar como parámetros: *–t* 1 *–d grado –g* γ *–r coef*0.

Kernel radial

Un kernel radial del estilo: $e^{-\gamma \times |u' \times v|^2}$ donde u' es nuestro vector de atributos, v es otro vector y los demás son valores reales. Se le pueden pasar como parámetros: *–t* 2 *–g*γ.

Figura 40. Kernels polinómicos con svm-toy

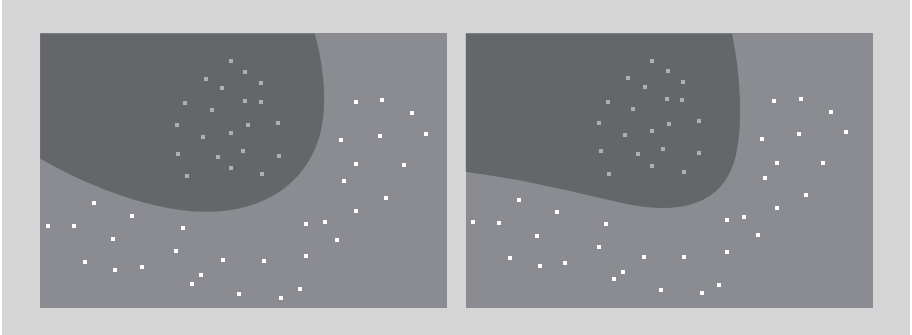
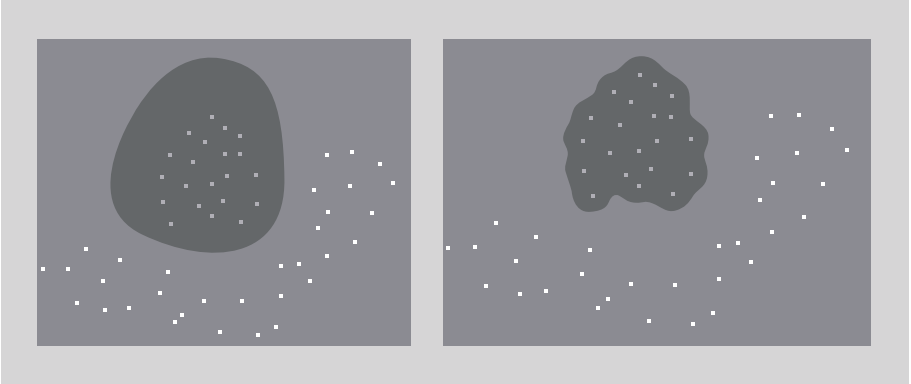


Figura 41. Kernels radiales con svm-toy



La figura 42 muestra un tercer ejemplo que trabajar. En este caso los kernels polinómicos no nos servirán. Separaremos los conjuntos de puntos utilizando kernels radiales. Fijaos en el efecto del parámetro gamma que muestra la figura 43:

- radial con gamma 1000 (*–t* 2 *–g* 1000 *–c* 10000),
- radial con gamma 10 (*–t* 2 *–g* 10 *–c* 10000).

Figura 42. Puntos para kernels radiales con svm-toy

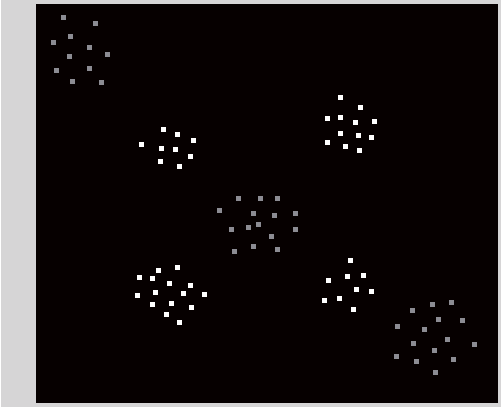
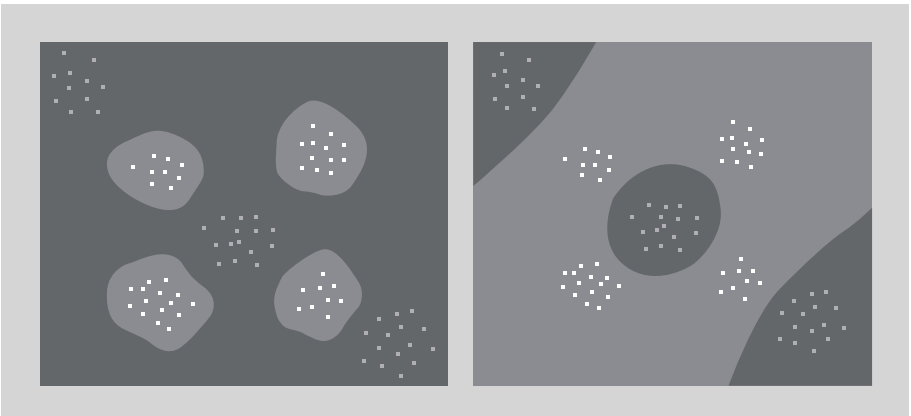


Figura 43. Kernels radiales con svm-toy



Formalización

Las máquinas de vectores de soporte se basan en el principio de la minimización del riesgo estructural* de la teoría del aprendizaje estadístico. En su forma básica, aprenden un hiperplano lineal de margen máximo que separa un conjunto de ejemplos positivos de uno de ejemplos negativos. Se ha demostrado que este sesgo de aprendizaje tiene muy buenas propiedades teniendo en cuenta la generalización de los clasificadores inducidos. El clasificador lineal queda definido por dos elementos: un vector de pesos w (con un componente por cada atributo) y un umbral b, que está relacionado con la distancia del hiperplano al origen: $h(x) = \langle x,w\rangle + b$ (figuras 29 y 31). La regla de clasificación $f(x)$ asigna +1 a un ejemplo x cuando $h(x) \geq 0$ y –1 cuando es menor. Los ejemplos positivos y negativos más cercanos al hiperplano (w,b) son los llamados vectores de soporte.

*** En inglés, *structural risk minimisation*. Descrita en: V. N. Vapnik (1998). *Statistical Learning Theory*. EE. UU.: John Wiley.**

Aprender el hiperplano de margen máximo (w,b) se puede abordar como un problema de *optimización cuadrática convexa** con una única solución. Esto consiste, en su formulación primal en:

*** En inglés, *convex quadratic optimisation*. Este es un algoritmo de optimización matemática como los que se describen en el subapartado 5.2.**

$$minimizar: \ ||w||$$

$$sujeto\,a: \ y_i(\langle w,x_i\rangle + b) \geq 1, \forall\ 1 \leq i \leq N$$

donde N es el número de ejemplos de entrenamiento y las y_i corresponden a las etiquetas.

Lo anterior se suele transformar a su formulación dual en que su solución se puede expresar como una combinación lineal de ejemplos de entrenamiento (vistos como vectores):

$$maximizar: \ \sum_{i=1}^{N} \alpha_i - \frac{1}{2}\sum_{i,j=1}^{N} y_i y_j \alpha_i \alpha_j \langle x_i,x_j\rangle$$

$$sujeto\,a: \ \sum_{i=1}^{N} y_i \alpha_i = 0, \alpha_i \geq 0, \forall\ 1 \leq i \leq N$$

donde los α_i son valores escalares no negativos determinados al resolver el problema de optimización.

De esta formulación, el vector ortogonal al hiperplano queda definido como:

$$h(x) = \langle w,x\rangle + b = b + \sum_{i=1}^{N} y_i \alpha_i \langle x_i,x\rangle$$

donde $\alpha_i \neq 0$ para todos los ejemplos de entrenamiento que están en el margen (vectores de soporte) y $\alpha_i = 0$ para los demás. $f(x) = +1$ cuando $h(x) \geq 0$ y -1 cuando es menor define el clasificador en su forma dual.

Aunque las máquinas de vectores de soporte son lineales en su forma básica, pueden ser transformadas en una forma dual, en la que los ejemplos de entrenamiento solo aparecen dentro de productos escalares, permitiendo el uso de funciones núcleo* para producir clasificadores no lineales. Estas funciones trabajan de manera muy eficiente en espacios de atributos de dimensionalidad muy elevada, donde los nuevos atributos se pueden expresar como combinaciones de muchos atributos básicos**.

* En inglés, *kernel functions*.

Si suponemos que tenemos una transformación no lineal ϕ de $\mathbb{R}^D$ a algún espacio de Hilbert $\mathfrak{F}$, podemos definir un producto escalar con una función núcleo tipo $\kappa(x,y) = \langle \phi(x), \phi(y) \rangle$ y obtener una formulación dual:

** Véase N. Cristianini; J. Shawe-Taylor (2000). *An Introduction to Support Vector Machines*. Reino Unido: Cambridge University Press.

$$maximizar: \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{N} y_i y_j \alpha_i \alpha_j \kappa(x_i, x_j)$$

$$sujeto\ a: \sum_{i=1}^{N} y_i \alpha_i = 0, \alpha_i \geq 0, \forall\ 1 \leq i \leq N$$

y obtener el vector ortogonal al hiperplano:

$$h(x) = \langle w, \phi(x) \rangle + b = b + \sum_{i=1}^{N} y_i \alpha_i \kappa(x_i, x)$$

Para algunos conjuntos de entrenamiento no es deseable obtener un hiperplano perfecto, es preferible permitir algunos errores en el conjunto de entrenamiento para obtener «mejores» hiperplanos (figuras 35 y 38). Esto se consigue con una variante del problema de optimización, llamada margen flexible, donde la contribución a la función objetivo de la maximización del margen y los errores en el conjunto de entrenamiento se pueden balancear mediante un parámetro normalmente llamado C. Este parámetro afecta a las variables ξ_i en la función (figura 35). Este problema queda formulado como:

$$minimizar: \frac{1}{2} \langle w, w \rangle + C \sum_{i=1}^{N} \xi_i$$

$$sujeto\ a: y_i(\langle w, x_i \rangle + b) \geq 1 - \xi_i, \xi_i \geq 0, \forall\ 1 \leq i \leq N$$

Análisis del método

Las máquinas de vectores de soporte fueron dadas a conocer en el año 92 por Boser, Guyon y Vapnik, pero no empezaron a ganar popularidad en el área del aprendizaje automático hasta finales de los noventa.

Referencia bibliográfica

B. Boser; I. Guyon; V. Vapnik (1992). «A training Algorithm for Optimal Margin Classifiers». En las actas del *Workshop con Computational Learning Theory*. COLT

Actualmente, son uno de los algoritmos de aprendizaje supervisados más utilizados en la investigación. Han demostrado ser muy útiles y se han aplicado a una gran variedad de problemas relacionados con el reconocimiento de patrones en bioinformática, reconocimiento de imágenes y categorización de textos, entre otros. Se ha probado que su sesgo de aprendizaje es muy robusto y que se obtienen muy buenos resultados en muchos problemas. Se están dedicando también muchos esfuerzos en la comunidad investigadora internacional a las posibilidades que ha abierto el uso de kernels.

Cuando aplicamos este algoritmo, hay una cuestión que tenemos que tener en cuenta: estamos combinando el uso de kernels con un algoritmo de optimización. Si aplicamos un kernel que no encaja bien con los datos, puede pasar que el algoritmo tarde en converger o que dé una mala solución. El tiempo que tarda en construir el modelo y el número de vectores que genera nos dan pistas de la bondad de la solución en la práctica y de la separabilidad del conjunto de datos con el kernel dado. El número de vectores de soporte es un número inferior al del conjunto de entrenamiento. La diferencia entre los dos conjuntos suele ser directamente proporcional a la bondad de la solución y del hiperplano. El tiempo de ejecución del proceso de aprendizaje suele ser inversamente proporcional a la bondad.

Tanto las SVM como los métodos descritos en este subapartado son binarios. Es decir, solo permiten separar entre dos clases. Cuando nos encontramos delante de un problema que tiene más de dos clases, tenemos que crear un marco para gestionar las predicciones de todas las clases a partir de los clasificadores binarios. Existen diversos mecanismos de binarización para abordar esta cuestión, aquí describiremos la técnica del uno contra todos*, que es una de las más utilizadas en la bibliografía.

*** En inglés, *one vs all.***

Esta técnica consiste en entrenar un clasificador por clase, cogiendo como ejemplos positivos los ejemplos de la clase y como negativos los ejemplos de todas las otras clases. El resultado de la clasificación es un peso (número real por clase). Para el caso monoetiqueta, se coge como predicción la clase con el peso mayor. Para el caso multietiqueta, se cogen como predicciones todas las clases con pesos positivos.

Lecturas complementarias

Para profundizar en el tema de la binarización podéis consultar:

E. Allwein; R. E. Schapire; Y. Singer (2000). «Reducing Multiclass to Binary: A Unifying Approach for Margin Classifiers». *Journal of Machine Learning Research* (núm. 1, págs. 113-141).

S. Har-Peled; D. Roth; D. Zimak (2002). «Constraint Classification for Multiclass Classification and Ranking». Actas del *Workshop on Neural Information Processing Systems*.

Uso en Python

Como muestra de la atención que las máquinas de vectores de soporte han atraído a la comunidad científica, tenemos el módulo *sklearn.svm* de *scikit-learn*, una vasta y completa librería de algoritmos relacionados con estas. En este apartado, utilizaremos el paquete SVC de este módulo, que proporciona un potente clasificador basado en la implementación LIBSVM de Chih-Chung Chang y Chih-Jen Lin* de máquinas de vectores de soporte. En la misma línea de los demás algoritmos de clasificación vistos en este capítulo, la aplicación de este método es muy sencilla. El programa 4.8 es un ejemplo de lo fácil que es emplearlo con diferentes kernels.

*** http://www.csie.ntu.edu.tw/~cjlin/libsvm**
Chih-Chung Chang; Chih-Jen Lin (2001). *LIBSVM: a library for support vector machines.*

Código 4.8: clasificador SVM con kernels lineal y RBF

```
1  from sklearn.svm import SVC
2  from sklearn.model_selection import train_test_split
3  from sklearn import datasets
4
5  iris = datasets.load_iris()
6  X = iris.data
7  y = iris.target
8
9  X_train, X_test, y_train, y_test = train_test_split(X, y,
10                                                     test_size=0.33)
11
12 kernels = {'linear','rbf'}
13
14 for k in kernels:
15     clf = SVC(kernel=k)
16     clf.fit(X_train, y_train)
17
18     print("PRECISION (",k,"):",clf.score(X_test,y_test))
```

Podéis encontrar la documentación relativa al paquete SVC en la documentación oficial de scikit-learn*. Como se puede observar, el constructor del modelo se puede invocar especificando como parámetro el kernel que se va a utilizar. Este no es el único parámetro que puede recibir el constructor, pues existe una serie de parámetros admitidos que nos permiten configurar el modelo muy significativamente, como son el grado del polinomio en el caso de kernels polinomiales, el coeficiente gamma en el caso de kernel RBF, y la tolerancia (o error mínimo por alcanzar como criterio de parada del algoritmo). Es recomendable consultar a fondo la documentación oficial antes de utilizar el método.

Ved también

Los archivos de datos y el programa están disponibles en el repositorio de la asignatura.

*** http://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html**

4.6. Protocolos de test

Este subapartado está dedicada a la validación de los procesos de aprendizaje, las medidas de evaluación de sistemas de clasificación y la significación estadística.

4.6.1. Protocolos de validación

La validación es el proceso por el que comprobamos cómo de bien (o mal) ha aprendido un conjunto de datos un algoritmo de clasificación.

La validación más simple, conocida como *validación simple*, consiste en dividir el conjunto de datos en dos: uno llamado de entrenamiento* y el otro de test. Se construye el modelo sobre el conjunto de entrenamiento y acto seguido se clasifican los ejemplos de test con el modelo generado a partir del de entrenamiento. Una vez obtenidas las predicciones se aplica alguna de las medidas de evaluación como las que se describen en el siguiente subapartado. Una cuestión importante a tener en cuenta cuando dividimos un conjunto de datos en los conjuntos de entrenamiento y test es mantener la proporción de ejemplos de cada clase en los dos conjuntos.

*** En inglés, *training*.**

En el caso de que tengamos un algoritmo de aprendizaje que necesita realizar ajustes de parámetros, se divide el conjunto de entrenamiento en dos conjuntos: uno de entrenamiento propiamente dicho y otro de validación*. Se ajustan los parámetros entre el conjunto de entrenamiento y el conjunto de validación y una vez encontrados los óptimos; juntamos el conjunto de entrenamiento con el de validación y generamos el modelo con el algorimo de aprendizaje. La validación la realizamos sobre el conjunto de test. Un error de método bastante usual es el utilizar el conjunto de test para realizar el ajuste de parámetros; esto no es estadísticamente correcto. Una de las técnicas más utilizadas para ajustar parámetros son los algorimos de optimización como los algoritmos genéticos que se describen en el subapartado 5.5.

*** El conjunto de validación para el ajuste de parámetros no tiene nada que ver con el proceso de validación de los algoritmos de aprendizaje.**

Un problema que se ha detectado en el uso de la validación simple es que según el conjunto de datos que tengamos, puede variar mucho el comportamiento del sistema en función de la partición de ejemplos que hayamos obtenido. Es decir, diferentes particiones de ejemplos conducen a diferentes resultados. En muchos casos, los investigadores dejan disponibles los conjuntos ya divididos en entrenamiento y test para que las comparaciones entre sistemas sean más fiables.

Para minimizar este efecto, se suele utilizar la *validación cruzada** en lugar de la simple. Este tipo de validación consiste en dividir un conjunto en k subconjuntos. Acto seguido, se realizan k pruebas utilizando en cada una un subconjunto como test y el resto como entrenamiento. A partir de aquí se calcula el promedio y la desviación estándar de los resultados. Con esto podemos ver mejor el comportamiento del sistema. Un valor muy utilizado de la k es 10. Una variante conocida de la validación cruzada es el *dejar uno fuera*. Este caso es como el anterior definiendo la k como el número de ejemplos del conjunto total. Nos quedaría el conjunto de test con un único ejemplo. Este método no se suele utilizar debido a su alto coste computacional.

*** En inglés, *cross-validation* o *k-fold cross-validation*.**

**** En inglés, *leave one out*.**

4.6.2. Medidas de evaluación

Uno de los problemas que nos encontramos al describir las medidas de evaluación es la gran diversidad de medidas que se utilizan en las diferentes áreas de investigación. Estas medidas suelen tener en cuenta las diferentes peculiaridades de los problemas de los diferentes campos. Otro problema es que en ocasiones estas medidas de evaluación reciben nombres diferentes en función del área. En este subapartado vamos a ver las medidas que se utilizan en los contextos de la clasificación, la recuperación de la información y la teoría de la detección de señales.

La mayoría de las medidas de evaluación se pueden expresar en función de la matriz de confusión o tabla de contingencia. La matriz de confusión contiene una partición de los ejemplos en función de su clase y predicción. La tabla 31 muestra el contenido de las matrices de confusiones para el caso binario. A modo de ejemplo, la celda *verdadero positivo* corresponde al número de ejemplos del conjunto de tests que tienen tanto la clase como la predicción positivas. A los *falsos positivos* también se los conoce como *falsas alarmas* o *error de tipo I*; a los *falsos negativos* como *error de tipo II*; a los *verdaderos positivos* como *éxitos*; y a los *verdaderos negativos* como *rechazos correctos*.

Tabla 31. Matriz de confusión

		clase real	
		positiva	negativa
predicción	positiva	verdadero positivo (tp)	falso positivo (fp)
	negativa	falso negativo (fn)	verdadero negativo (tn)

El *error* o *ratio de error* mide el número de ejemplos que se han clasificado incorrectamente sobre el total de ejemplos. Se pretende que los algoritmos de aprendizaje tiendan a minimizar esta medida. Su fórmula corresponde a:

$$error = \frac{fp + fn}{tp + fp + fn + tn}$$

La *exactitud** (a veces llamada también *precisión*) corresponde a los ejemplos que se han clasificado correctamente sobre el total de ejemplos. Esta medida es la complementaria de la anterior. Su fórmula corresponde a:

* En inglés, *accuracy*.

$$exactitud = \frac{tp + tn}{tp + fp + fn + tn}$$

La *precisión** o *valor predictivo positivo* corresponde a los ejemplos positivos bien clasificados sobre el total de ejemplos con predicción positiva. Su fórmula corresponde a:

* En inglés, *precision*.

$$precisión = \frac{tp}{tp + fp}$$

La *sensibilidad** corresponde a los ejemplos positivos bien clasificados sobre el total de ejemplos positivos. Su fórmula corresponde a:

* En inglés, *recall*.

$$sensibilidad = \frac{tp}{tp + fn}$$

El conjunto de precisión y sensibilidad puede verse como una versión extendida de la exactitud. Estas medidas vienen del campo de la recuperación de la información donde el valor de *tn* es muy superior al resto y sesga las medidas de evaluación. De allí nos llega también la medida $F1$, propuesta en 1979 por van Rijsbergen, que combina las dos anteriores:

$$F1 = 2 \times \frac{precisión \times sensibilidad}{precisión + sensibilidad} = \frac{2 \times tp}{2 \times tp + fn + fp}$$

Esta medida descarta los elementos negativos debido al sesgo producido por la diferencia entre el número de ejemplos negativos y el de positivos; en el ámbito de la recuperación de información llega a ser muy crítico.

Existen otras medidas que se utilizan a partir de la matriz de confusión, como puede ser la *especificidad*:

$$especificidad = \frac{tn}{fp + tn}$$

que se suelen utilizar en el área de la teoría de detección de señales. De aquí sale también un diagrama llamado ROC que se utiliza en varios ámbitos como el de la biotecnología.

Lectura complementaria

T. Fawcet (2006). An Introduction to ROC Analysis. *Pattern Recognition Letters*, 27, 861-874.

Todas las medidas de este subapartado se han descrito a partir del problema binario, pero también se pueden utilizar en problemas multiclase.

4.6.3. Tests estadísticos

Cuando tenemos las predicciones de dos clasificadores y hemos aplicado alguna de las medidas de evaluación; estadísticamente no es suficiente con mirar cuál de ellos ha obtenido un valor más alto; hay que mirar, además, si la diferencia es estadísticamente significativa.

Lectura complementaria

T. G. Dietterich (1998). "Approximate statistical tests for comparing supervised classification learning algorithms". *Neural Computation* (núm. 10, págs. 1895-1924).

Para realizar esto, hay dos formas de hacerlo en función del tipo de validación que hayamos utilizado. Si hemos utilizado la validación cruzada, tenemos que usar el *test de Student* y si hemos utilizado la validación simple el *test de McNemar*. Dados dos clasificadores F y G:

- el test de *Student* con una confianza de $t_{9,0,975} = 2,262$ o equivalente (depende del número de particiones) en la validación cruzada:

$$\frac{\overline{p} \times \sqrt{n}}{\sqrt{\frac{\sum_{i=1}^{n}(p(i)-\overline{p})^2}{n-1}}} > 2,262$$

  donde n es el número de particiones, $\overline{p} = \frac{1}{n}\sum_{i=1}^{n} p(i)$, $p(i) = p_F(i) - p_G(i)$, y $p_X(i)$ es el número de ejemplos mal clasificados por el clasificador X (donde $X \in \{F,G\}$).

- el test de *McNemar* con un valor de confianza de $\chi^2_{1,0,95} = 3,842$ con la validación simple:

$$\frac{(|A-B|-1)^2}{A+B} > 3,842$$

  donde A es el número de ejemplos mal clasificados por el clasificador F pero no por el G, y B es el número de ejemplos mal clasificados por el G pero no por el F.

Demšar, en el artículo "Statistical Comparisons of Classifiers over Multiple Data Sets", recomienda como alternativa el test no paramétrico de Wilcoxon:

Lectura complementaria

J. Demšar (2006). "Statistical Comparisons of Classifiers over Multiple Data Sets". *Journal of Machine Learning Research* (núm. 7, págs. 1-30).

$$z = \frac{T - \frac{1}{4}N(N+1)}{\sqrt{\frac{1}{24}N(N+1)(2N+1)}}$$

donde N es el número de conjuntos de datos y $T = min(R^+,R^-)$, donde:

$$R^+ = \sum_{d_i>0} rank(d_i) + \frac{1}{2}\sum_{d_i=0} rank(d_i)$$

$$R^- = \sum_{d_i<0} rank(d_i) + \frac{1}{2}\sum_{d_i=0} rank(d_i)$$

teniendo en cuenta que d_i es la diferencia de la medida de evaluación entre dos clasificadores sobre el conjunto de datos i y que realizamos un *ranking* sobre estas distancias. Los rankings tal que $d_i = 0$ se dividen equitativamente entre las dos sumas. En caso de ser un número impar de ellas, se ignora una. Para $\alpha = 0,05$, la hipótesis nula se rechaza cuando z es menor que –1,96.

4.6.4. Comparativa de clasificadores

En este subapartado vamos a ver dos formas de comparar clasificadores entre ellos.

Comparativas de más de dos clasificadores sobre múltiples conjuntos de datos

La forma clásica de realizar una comparativa de más de dos clasificadores sobre múltiples conjuntos de datos es utilizar el método estadístico ANOVA. Desafortunadamente, este estadístico realiza muchas suposiciones que usualmente no se cumplen en aprendizaje automático. Demšar nos recomienda un test alternativo, el de Friedman.

Lecturas complementarias

J. Demšar (2006). "Statistical Comparisons of Classifiers over Multiple Data Sets". *Journal of Machine Learning Research* (núm. 7, págs. 1-30).

Supongamos que tenemos k clasificadores y n conjuntos de datos. Para calcular el test de Friedman realizamos un ranking de los diferentes algoritmos para cada conjunto de datos por separado. En caso de empate promediamos los rankings. r_i^j corresponderá al ranking del algoritmo j sobre el conjunto de datos i. Bajo la hipótesis nula, el estadístico se calcula como:

$$\chi_F^2 = \frac{12n}{k(k+1)}\left(\sum_j R_j^2 - \frac{k(k+1)^2}{4}\right)$$

donde $R_j = \frac{1}{n}\sum_i r_i^j$ corresponde al promedio de los algoritmos sobre el conjunto de datos i.

Este estadístico se distribuye de acuerdo con χ_F^2 con $k-1$ grados de libertad, cuando n y k son suficientemente grandes ($n > 10$ y $k > 5$)*.

El estadístico siguiente** es una mejora del anterior (no tan conservador):

$$F_F = \frac{(n-1)\chi_f^2}{n(k-1)-\chi_f^2}$$

que se distribuye de sobre una distribución F con $k-1$ y $(k-1)(n-1)$ grados de libertad.

* Para valores menores, consultad los valores críticos exactos en: **J. H. Zar (1998).** ***Biostatistical Analysis (4.ª ed.).*** **Prentice Hall. D. J. Sheskin (2000).** ***Handbook of parametric and nonparametric statistical procedures.*** **Chapman & Hall/CRC.**

** **R. L. Iman; J. M. Davenport (1980). "Approximations of the critical region of the Friedman statistic".** ***Communications in Statistics.***

Medidas de acuerdo y kappa

En este subapartado vamos a ver dos de las medidas más útiles para comparar el comportamiento de diferentes clasificadores.

Dadas las predicciones de dos clasificadores F y G, el *acuerdo** (observado) se calcula como:

$$Pa = \frac{A}{N}$$

donde A corresponde al número de ejemplos para los que los dos han dado la misma predicción y N es el número total de ejemplos.

Y el estadístico *kappa* como:

$$\kappa(F,G) = \frac{Pa - Pe}{1 - Pe}$$

donde Pe (el *agreement* esperado) se calcula como:

$$Pe = \sum_{i=1}^{ns} \frac{P_F(i) \times P_G(i)}{N^2}$$

donde ns es el número de clases, y $P_X(i)$ es el número de predicciones del clasificador X (con $X \in \{F,G\}$) de la clase i.

* En inglés, *agreement*.

Ved también

En el subapartado 2.3.3 se estudia la creación de dendrogramas.

Este estadístico tiene en cuenta cómo se distribuyen las predicciones entre las diferentes clases los clasificadores. A partir de él, se puede estudiar cómo se comportan diferentes clasificadores entre ellos e incluso crear un dendrograma de los mismos. En un experimento realizado sobre un problema de lenguaje natural*, da como resultado un parecido notable entre los algoritmos del AdaBoost y las SVMs diferenciándolos del resto, por poner solo un ejemplo. El experimento parte de una tabla como la 32, donde c_i corresponde al clasificador i.

* Capítulo 7 del libro: E. Agirre; P. Edmonds (ed.) (2006) *Word Sense Disambiguation: Algorithms and Applications.* Springer.

Tabla 32. Matriz de kappa dos a dos

	c_1	c_2		c_{N-1}
c_2	$\kappa(c_1,c_2)$	–	–	–
c_3	$\kappa(c_1,c_3)$	$\kappa(c_2,c_3)$	–	–
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
c_N	$\kappa(c_1,c_N)$	$\kappa(c_2,c_N)$	–	$\kappa(c_{N-1},c_N)$

Conclusiones

Hay dos conceptos que afectan profundamente a todos los procesos de aprendizaje. El primero es la llamada maldición de la dimensionalidad*. Este concepto tiene que ver con la necesidad de ejemplos de entrenamiento necesarios para aprender en función de la complejidad de las reglas de clasificación; que aumenta de forma exponencial.

* En inglés, *curse of dimensionality*.

El segundo concepto es el sesgo inductivo*. Este concepto contempla que los diferentes algoritmos de aprendizaje favorecen diferentes tipos de reglas de clasificación en los procesos de aprendizaje.

*** En inglés, *inductive bias*.**

Sería deseable disponer de un algoritmo de aprendizaje que fuera mejor que todos los demás, independientemente del conjunto de datos a tratar. Desafortunadamente, esto no ocurre y hace que necesitemos diferentes algoritmos para diferentes tipos de conjuntos de datos. Estos resultados se han cuantificado y se los ha denominado, en inglés, *the no free lunch theorem* (Kulkarni y Harman, 2011).

Todo esto ha hecho que muchos autores consideren que el diseño de aplicaciones prácticas de procesos de aprendizaje pertenece tanto a la ingeniería y a la ciencia, como al arte.

5. Optimización

5.1. Introducción

La **optimización** de un problema P es la tarea de buscar los valores que aplicados a P satisfagan unas determinadas expectativas. Dependiendo de la naturaleza de P las expectativas que han de cumplir esos valores, o sea, lo que debe buscarse en la tarea de optimización puede tener características muy diferentes; en cualquier caso, se supone que son las características más deseables para quien propone el problema.

Así, se pueden aplicar técnicas de optimización para encontrar valores que minimicen los costes de una actividad, que minimicen el tiempo requerido para llevarla a cabo, que determinen la trayectoria óptima de un vehículo espacial, que gestionen de forma óptima un conjunto de recursos como una red de servidores, que maximicen los beneficios de una inversión o que encuentren la combinación de genes que más se asocia a una determinada patología, entre innumerables ejemplos posibles.

Para cada problema P se define una **función objetivo** f que mide de alguna forma la idoneidad de cada posible solución de P: coste, tiempo requerido, beneficio, combustible empleado, tiempo de respuesta de un sistema, productividad por hora, etc. El dominio de f, o sea el conjunto de puntos que pueden probarse como solución al problema, recibe el nombre de **espacio de soluciones** y se denota por la letra S.

El rango de f suele ser un valor escalar (entero o real, o un subconjunto), si bien en algunos casos puede tratarse de un valor de más dimensiones. Dependiendo de P y de la forma en que se defina f la tarea de optimización puede consistir en encontrar los puntos que minimicen f (coste, tiempo requerido, etc.) o los puntos que la maximicen (beneficio, productividad, eficiencia).

A menudo la función f se define como una **función de error**, que es la diferencia entre el resultado deseado y el obtenido por un punto $x \in S$. En esos casos el objetivo de la tarea de optimización siempre es, lógicamente, minimizar f.

En cualquier caso, hay que diferenciar el enunciado del problema P de la función objetivo f, pues a menudo conviene definir f de manera sustancialmente distinta a P para incrementar la efectividad de los métodos de optimización.

Si vemos f como el relieve geográfico de un país, con valles y montañas, es importante que f sea una función sin grandes llanuras, grandes zonas de valor homogéneo que no dan ninguna orientación sobre el mejor camino a tomar. El agua se estanca en las llanuras, sólo fluye y busca el mar en las pendientes. f debe definirse de forma que refleje los más pequeños cambios en la calidad de una solución, para que los métodos de optimización sepan que, poco a poco, están mejorando la solución propuesta. En el subapartado 5.8 se plantea un ejemplo, el coloreado de un mapa, en el que se aprecia claramente este requerimiento. Si f es el número de colores necesario para colorear un mapa sin que haya dos países contiguos del mismo color, tendremos una función objetivo con vastas llanuras. Sin embargo, si definimos f como el número de países contiguos del mismo color, tendremos una medida mucho más fina del progreso de la optimización, pues cada vez que se resuelva uno de esos "errores", f disminuirá un poco.

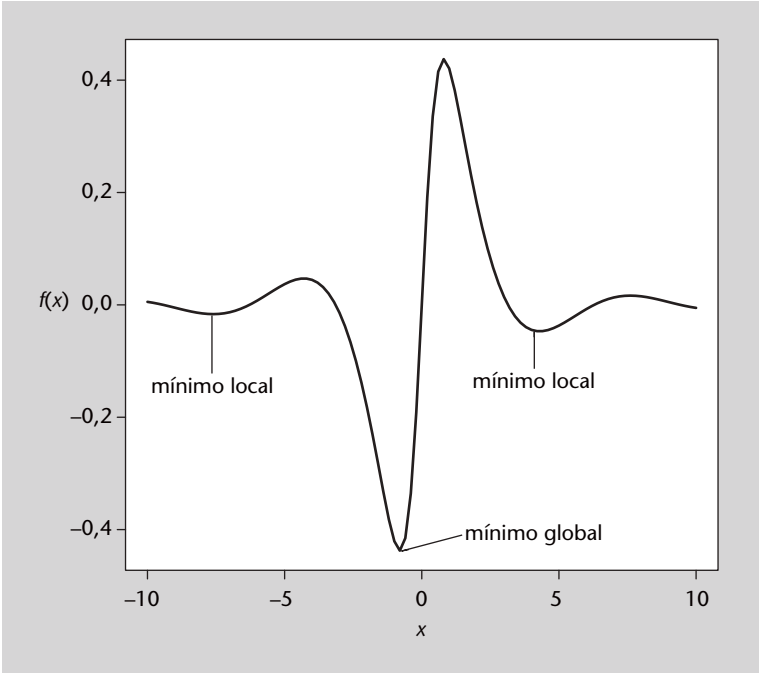
Este ejemplo nos lleva a una última consideración sobre f: en muchos problemas existen soluciones prohibidas, ya que incumplen algunas restricciones del problema; en el ejemplo anterior, un coloreado en el que dos países contiguos tienen el mismo color no es válido. Sin embargo muy a menudo conviene definir una f **relajada** respecto a P, que permita tales soluciones prohibidas como paso intermedio para alcanzar las soluciones óptimas deseadas. De lo contrario, el salto necesario para pasar de una solución válida a otra mejor puede ser tan grande que resulte, en la práctica, inalcanzable. Por ejemplo, permitiendo que dos países contiguos tengan el mismo color como paso intermedio para conseguir un coloreado válido. Si no se permite, el método tiene que obtener un nuevo coloreado del mapa en un solo paso, lo que casi seguro lo condenará al fracaso.

Siguiendo con la analogía geográfica del espacio de soluciones, y suponiendo que queremos minimizar f, lo que se busca es el valle con el punto más bajo (en el caso contrario buscaremos la montaña más alta). Dependiendo del problema es posible que exista un único valle con la solución óptima o, por el contrario, que haya numerosos valles con soluciones más o menos buenas. Si nos encontramos en el fondo de un valle nos parece que nuestra posición es la más baja, aunque es posible que haya otro valle más bajo aún en alguna otra parte. En un problema de optimización puede ocurrir lo mismo, que haya muchas zonas con soluciones relativamente buenas, si bien en otra zona de S se puedan encontrar soluciones aún mejores. En este caso se habla de **óptimos locales** y **óptimos globales**.

El objetivo de los métodos de optimización es encontrar el óptimo global y no "dejarse engañar" por los óptimos locales, aunque esto no siempre es fácil o posible.

El principal obstáculo es que, para pasar de un óptimo local a un óptimo global (o a un óptimo local mejor que el primero), es necesario pasar por puntos peores (salir de un valle para bajar a otro), y en principio eso se ve como un empeoramiento de la solución y por tanto se descarta. Puede verse un ejemplo en la figura 44.

Figura 44. Mínimos locales y mínimo global



Una última consideración se refiere al tamaño de *S*. En gran parte de los problemas reales, *S* tiene un tamaño muy grande, en especial si tiene muchas variables o dimensiones. En lo que se conoce como la **maldición de la dimensionalidad***, si tenemos un espacio de soluciones de una dimensión (una recta) de lado 1, con 100 puntos se puede explorar en intervalos de 0,01. Sin embargo, si tenemos un espacio de dos dimensiones y lado 1 (un cuadrado), serán necesarios 100^2 puntos para poder explorarlo con una resolución de 0,01; en un cubo se necesitan 100^3 puntos, y en un hipercubo de 10 dimensiones serán necesarios 100^{10} puntos. Por ese motivo una exploración exhaustiva de *S* suele ser impracticable, y ni siquiera métodos como el de Newton, que van subdividiendo el espacio de soluciones progresivamente, son prácticos.

*** En inglés, *curse of dimensionality*, término acuñado por Richard E. Bellman**

Así, en muchos casos no se conoce cuál es la solución óptima, de hecho, los métodos de optimización descritos en este apartado están diseñados para encontrar soluciones aceptablemente buenas, ya que no es posible saber si son las óptimas, en espacios de soluciones muy grandes (con muchas dimensiones).

Ved también

Una estrategia complementaria consiste en utilizar las técnicas de reducción de dimensionalidad vistas en el subapartado 3.1 de este módulo.

5.1.1. Tipología de los métodos de optimización

Existe una gran variedad de métodos de optimización debido a que cada problema requiere o permite estrategias diferentes. Básicamente se puede clasificar en las categorías siguientes:

- **Métodos analíticos**: consisten en realizar alguna transformación analítica de la función objetivo para determinar los puntos de interés. Como ejemplo se estudia el método de los multiplicadores de Lagrange en el subapartado 5.2. La característica de estos métodos es que se obtiene una solución exacta. Si es posible se deben utilizar, pero por desgracia en la mayoría de problemas reales no es posible obtener una solución analítica.

- **Métodos metaheurísticos**: se denomina **heurísticos** a las técnicas basadas en la experiencia o en analogías con otros procesos que resultan útiles para resolver un problema. En el área que nos ocupa, la de la optimización, los heurísticos son técnicas que suelen ayudar a optimizar un problema, habitualmente tomando soluciones aleatorias y mejorándolas progresivamente mediante el heurístico en cuestión. Los métodos que estudiaremos en este texto pertenecen a esta categoría (salvo el de Lagrange), pues son los que se consideran propios del área de la inteligencia artificial. Ejemplos: búsqueda aleatoria, descenso de gradientes, algoritmos genéticos y programación evolutiva, búsqueda tabú, recocción simulada, etc.

El prefijo "meta-" de los métodos metaheurísticos se debe a que tales heurísticos son aplicables a cualquier problema, pues los métodos no presuponen nada sobre el problema que pretenden resolver y por tanto son aplicables a una gran variedad de problemas. En general se hablará de "metaheurístico" o simplemente de "método" para hablar de cada técnica en general, y de "algoritmo" para la aplicación de una técnica a un problema concreto, si bien por motivos históricos esta nomenclatura no siempre se sigue (como en el caso de los algoritmos genéticos).

Los métodos metaheurísticos no garantizan la obtención de la solución óptima, pero suelen proporcionar soluciones aceptablemente buenas para problemas inabordables con técnicas analíticas o más exhaustivas.

5.1.2. Características de los metaheurísticos de optimización

La mayoría de metaheurísticos de optimización parten de una o más soluciones elegidas aleatoriamente que se van mejorando progresivamente. Si se parte de una única solución aleatoria y las nuevas soluciones propuestas por el heurístico dependen de ella, se corre el riesgo de quedarse estancado en un óptimo local. Por ese motivo tales métodos suelen denominarse **métodos locales**, en contraposición a los **métodos globales**, que suelen generar varias

soluciones aleatorias independientes y por tanto pueden explorar diferentes áreas del espacio de soluciones; así evitan, en la medida de lo posible, limitarse a un óptimo local determinado.

Otro aspecto común a los métodos metaheurísticos que se estudiarán en este apartado es su naturaleza iterativa, lo que lógicamente lleva a la cuestión del número de iteraciones que deberá ejecutar cada algoritmo. Cuando el óptimo global sea conocido (por ejemplo, en una función de error será 0) y sea posible alcanzarlo en un tiempo aceptable, se podrá iterar el algoritmo hasta llegar al óptimo global. Sin embargo, en la mayoría de problemas el óptimo global no es conocido o no se puede garantizar que se encontrará en un tiempo determinado. Por tanto, hay que fijar algún criterio de terminación del algoritmo. Entre los criterios más habituales se encuentran:

- Alcanzar un valor predeterminado de la función objetivo (por ejemplo, un error $< \epsilon$).
- Un número fijo de iteraciones.
- Un tiempo de ejecución total determinado.
- Observar una tendencia en las soluciones obtenidas (no se observa mejora durante n iteraciones).

Por último, los métodos metaheurísticos suelen caracterizarse por una serie de parámetros (hiperparámetros) que deben ajustarse de forma adecuada para resolver cada problema eficientemente. Al tratarse de heurísticos no hay reglas objetivas que determinen tales parámetros, y por tanto se recurre a valores orientativos obtenidos en estudios experimentales sobre diferentes tipos de problemas. De hecho, en algunos casos se aplica un algoritmo de optimización sobre los parámetros del algoritmo de optimización, lo que se conoce como **metaoptimización**.

5.2. Optimización mediante multiplicadores de Lagrange

Un fabricante de envases nos plantea la siguiente cuestión: ¿qué proporciones debe tener una lata cilíndrica para que contenga un volumen determinado utilizando la menor cantidad posible de metal? ¿Qué es mejor? ¿Que sea muy alta, muy ancha o igual de alta que de ancha?

La optimización mediante multiplicadores de Lagrange es un método de optimización matemática que permite encontrar el mínimo o el máximo de una función teniendo en cuenta una o más **restricciones**, que son otra función o funciones que se deben satisfacer.

En el ejemplo anterior, la función que se desea minimizar es la que determina la superficie de la lata, y la restricción es la que determina su volumen. Nótese que, si no se añade la restricción del volumen, la minimización de la función de la superficie dará como resultado una lata de altura y radio cero; un resultado correcto pero poco útil en la práctica.

5.2.1. Descripción del método

A continuación se describirá el método aplicándolo a funciones de dos variables; su generalización a más dimensiones es inmediata. Sea $f(x,y)$ la función que deseamos minimizar, y sea $g(x,y) = c$ la restricción que se aplica. Sin entrar en la explicación matemática, los puntos que buscamos son aquellos en los que los **gradientes** de f y g coinciden:

$$\nabla_{x,y} f(x,y) = \lambda \nabla_{x,y} g(x,y) \tag{49}$$

Gradiente de una función

El gradiente de una función f en un punto es un vector que apunta en la máxima pendiente de f. La función gradiente de f es un campo de vectores que apuntan, en cada punto, en la dirección de máxima pendiente de f, y se denota por ∇f.

Donde λ es una constante que es necesaria para igualar las longitudes de los vectores gradiente, ya que pueden tener la misma dirección pero longitudes diferentes. La función gradiente de una función se calcula mediante las derivadas parciales de la función para cada una de sus variables:

$$\nabla_{x,y} f(x,y) = \left\langle \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y} \right\rangle \tag{50}$$

$$\nabla_{x,y} g(x,y) = \left\langle \frac{\partial g}{\partial x}, \frac{\partial g}{\partial y} \right\rangle \tag{51}$$

Por lo tanto el problema se transforma en un sistema de ecuaciones que hay que resolver para obtener la solución deseada:

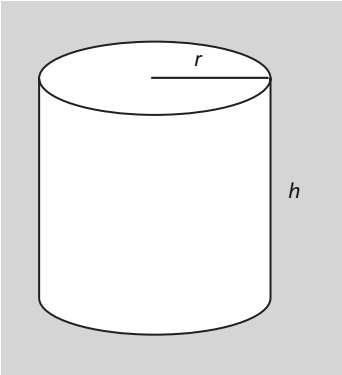
$$\left.\begin{array}{l} \frac{\partial f}{\partial x} = \lambda \frac{\partial g}{\partial x} \\ \frac{\partial f}{\partial y} = \lambda \frac{\partial g}{\partial y} \end{array}\right\} \tag{52}$$

En este sistema hay dos ecuaciones y tres variables: x, y y λ. En general obtendremos los valores para x e y en función de λ, que es el parámetro que nos da el grado de libertad necesario para elegir los valores deseados que cumplan la restricción impuesta; en otras palabras, nuestro problema de optimización se ha transformado en una función de una única variable, λ, que nos permite elegir el valor más adecuado a las especificaciones del problema.

5.2.2. Ejemplo de aplicación

Volvamos al problema de la lata cilíndrica cuya superficie deseamos minimizar para un determinado volumen. En la figura 45 se muestra la lata con las dos variables que la caracterizan, el radio r y la altura h.

Figura 45. Una lata cilíndrica



En este caso, la función que deseamos minimizar es la superficie de la lata (superficie del lado más superficial de las dos tapas), que es la que implica la cantidad de metal utilizado, y la restricción es el volumen contenido por la lata, luego en este problema f y g serán:

$$f(r,h) = 2\pi rh + 2\pi r^2 \tag{53}$$

$$g(r,h) = \pi r^2 h \tag{54}$$

Tras obtener las funciones gradiente y multiplicar ∇g por λ, hay que resolver el siguiente sistema de ecuaciones:

$$\left.\begin{array}{c} 2\pi rh = \lambda(2\pi h + 4\pi r) \\ \pi r^2 = \lambda 2\pi r \end{array}\right\} \tag{55}$$

Expresando r y h en función de λ se obtiene:

$$\left.\begin{array}{c} r = 2\lambda \\ h = 4\lambda \end{array}\right\} \tag{56}$$

Y de ahí se deduce que la relación entre radio y altura de una lata debe ser $h = 2r$ para utilizar la menor cantidad de material con relación al volumen que contiene.

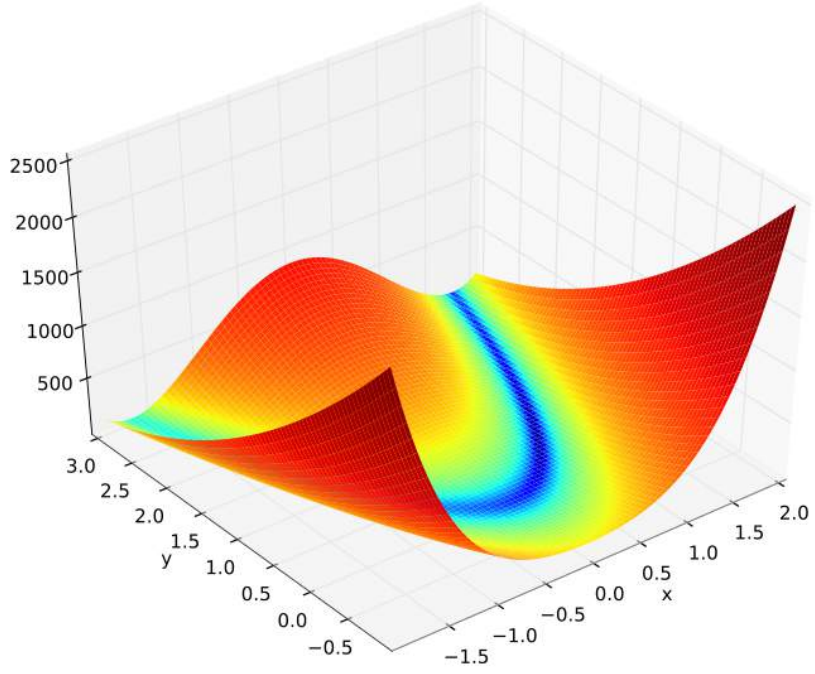
5.2.3. Análisis del método

La naturaleza analítica de este método condiciona sus ventajas e inconvenientes. Como ventaja destaca la posibilidad de obtener no una sino un rango de soluciones óptimas de forma relativamente rápida. Por otra parte, su principal limitación consiste en que sólo es aplicable a funciones diferenciables, lo que restringe el tipo de problemas en los que puede utilizarse este método.

5.3. Descenso de gradientes

La función de Rosenbrock, que puede verse en la figura siguiente, es una función clásica para probar métodos de optimización, ya que visualmente es muy intuitivo encontrar el mínimo y además se puede solucionar de forma analítica. Sin embargo, encontrar el mínimo global de forma heurística no es sencillo, ya que aunque es fácil llegar al «valle» (zona azul), no hay grandes diferencias entre sus diferentes puntos.

Figura 46. Función de Rosenbrock



Fuente: Wikipedia

Su formulación matemática es la siguiente:

$$f(x,y) = (a-x)^2 + b(y-x^2)^2 \tag{57}$$

para el caso $a = 1$ y $b = 100$, el mínimo de la función se encuentra en el punto (1,1) y su valor es 0.

Suponiendo que no sepamos encontrar el mínimo de forma analítica, ¿qué método podría usarse para encontrarlo?

5.3.1. Presentación de la idea

Los **gradientes** de una función f de n variables son las derivadas de f respecto a cada una de las variables de las que depende. Por ejemplo, si tenemos $f(x,y)$, sus gradientes serán las derivadas de f respecto a x y a y.

Para presentar la idea de la optimización mediante descenso de gradientes vamos a empezar con un caso muy sencillo, una función de una sola variable, con lo que lógicamente su gradiente es su derivada respecto a x. La función que utilizaremos es la siguiente:

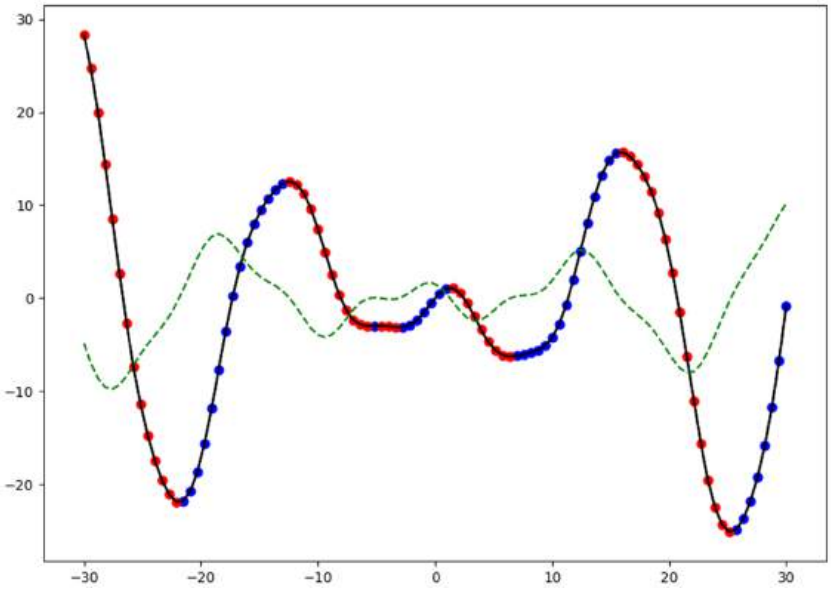
$$f(x) = \sin x + x\cos(1+\frac{x}{3}) \tag{58}$$

Su gradiente, es decir, su derivada respecto a su única variable (x) es la siguiente:

$$f'(x) = \cos x - x\sin(1+\frac{x}{3})/3 + \cos(1+\frac{x}{3}) \tag{59}$$

La figura 47 representa la función f (línea continua negra), su gradiente (línea de trazos verde) y el signo del gradiente (en rojo cuando es negativo, en azul si es positivo).

Figura 47. Gradiente (derivada) de una función



Derivada de una función

Recordemos que la derivada de una función es negativa cuando la función está decreciendo y positiva cuando está creciendo. En los máximos y mínimos la derivada es cero.

Si nuestro objetivo es alcanzar los mínimos de la función, si estamos en un punto rojo, es decir en una zona de bajada (decrecimiento) de la función, deberemos movernos hacia la derecha (aumentar la x). Igualmente, si estamos en un punto azul (zona de crecimiento de la función), nos interesará movernos hacia la izquierda (disminuir la x). Relacionando lo anterior con el hecho de que en las zonas de puntos rojos el gradiente de f respecto de x es negativo, se deduce que si estamos en una zona de puntos rojos tenemos que restarle el gradiente para desplazarnos hacia el mínimo; igualmente, si estamos en una zona de puntos azules en la que el gradiente es positivo, tenemos que restárselo a la x para desplazarnos hacia el mínimo. Es decir, en cualquiera de los dos casos hay que restar el gradiente a la x para acercarla a un mínimo:

$$x_{t+1} \leftarrow x_t - \gamma \nabla f(x_t) \tag{60}$$

donde x_t es el valor de x en la iteración t del algoritmo de optimización y ∇ es un operador que representa el gradiente de f respecto de sus variables. Por otra parte, γ es la llamada **tasa de aprendizaje**, que intuitivamente es un factor que controla la longitud de los «saltos» que va dando el algoritmo dentro del dominio de la función. Más adelante analizaremos la importancia de la tasa de aprendizaje.

Aplicando esta actualización iterativamente se garantiza que el algoritmo alcanzará un **mínimo local**, punto en el que el gradiente es cero y por tanto termina el algoritmo.

En funciones de dos o más variables el proceso es idéntico, solo que en ese caso el gradiente tiene un componente respecto a cada una de las variables de la función, lo que permite actualizar su valor según el mismo principio planteado para una variable.

5.3.2. Ejemplo de aplicación

Volviendo al ejemplo de la función de Rosenbrock, podemos deducir que su gradiente viene dado por las expresiones siguientes:

$$\frac{\delta f}{\delta x} = -2(a-x) - 4b(y-x^2)x \tag{61}$$

$$\frac{\delta f}{\delta y} = 2b(y-x^2) \tag{62}$$

donde la primera expresión es el gradiente de f respecto de x y la segunda el gradiente de f respecto de y.

En el siguiente fragmento de código puede verse definida la función de Rosenbrock y su función gradiente, definidas de forma que puedan utilizarse en el proceso de optimización.

Código 5.1: función de Rosenbrock y su función gradiente

```
1  import numpy
2
3  # Funcion de Rosenbrock
4  def f(punto, *args):
5
6      # Descomponer el punto y los parametros para manejarlos mejor
7      x, y = punto
8      a, b = args
9
10     return (a - x)**2 + b*(y - x**2)**2
11
12 # Gradiente de la funcion de Rosenbrock
13 def gradf(punto, *args):
14
15     x, y = punto
16     a, b = args
17
18     gfx = -2*(a - x) -4*b*(y - x**2)*x
19     gfy = 2*b*(y - x**2)
20
21     # El gradiente es un vector de 2 componentes, uno por variable
22     return numpy.asarray((gfx, gfy))
```

A continuación, puede verse cómo utilizar la optimización por descenso de gradientes de SciPy para minimizar la función de Rosenbrock y encontrar el mínimo. Hay que destacar que la función *fmin_cg* espera un punto inicial a partir del cual empezar a explorar. Se recomienda probar con diferentes puntos iniciales para explorar diferentes «valles» del dominio de la función.

Código 5.2: función de Rosenbrock y su función gradiente

```
1  # Interesa probar con diferentes puntos iniciales
2  punto0 = numpy.asarray((0, 0))
3  #punto0 = numpy.asarray((1,0))
4
5  # Valores de los parametros a y b
6  args = (1, 100)
7
8  from scipy import optimize
9  res1 = optimize.fmin_cg(f, punto0, fprime=gradf, args=args)
10 print (res1)
```

5.3.3. Cuestiones adicionales

El algoritmo de descenso de gradientes es muy utilizado por su eficiencia y simplicidad, por lo que conviene tener presentes algunas cuestiones adicionales para poder aprovecharlo en todo su potencial.

Como se ha visto en el ejemplo anterior, una de las principales limitaciones del descenso de gradientes es que siempre encuentra el fondo del valle en el que se escoge el punto inicial. Eso obliga a probar diferentes puntos iniciales o bien a utilizar otros métodos de optimización. En otras palabras, el descenso

de gradientes es un método de optimización **local**, lo que debe tenerse presente a la hora de aplicarlo.

Al describir el método se hablado del parámetro γ, que es la **tasa de aprendizaje***. La elección de una tasa de aprendizaje adecuada es un aspecto crítico para la configuración del descenso de gradientes, ya que este parámetro controla el tamaño de cada «paso» o «salto» del algoritmo dentro del dominio de la función. Así pues:

*** En inglés, *learning rate*.**

- Una tasa de aprendizaje muy pequeña permite alcanzar el mínimo con precisión (no se lo «salta»), pero requiere muchas iteraciones del algoritmo y por tanto la optimización es más lenta.

- Por el contrario, aunque una tasa de aprendizaje demasiado grande acelera la optimización, puede ocurrir que el algoritmo se «salte» el mínimo y esté dando vueltas a su alrededor, o agote las iteraciones sin alcanzarlo.

El problema de la elección de la tasa de aprendizaje es objeto de investigación. Las soluciones básicas consisten en empezar con una tasa alta e ir reduciéndola, si bien hay otras soluciones avanzadas que analizan los gradientes obtenidos y van adaptando la tasa en consecuencia.

De hecho, una mejora del descenso de gradientes consiste en añadir una cierta «inercia» (**momento**) al método, sumando una fracción del vector de velocidad de las anteriores actualizaciones, es decir que si las últimas actualizaciones han tomado una cierta dirección, se hace que el método tienda a seguir por esa dirección. Esto consigue que la respuesta del método mejore, sobre todo cuando los gradientes de las variables son muy diferentes en magnitud.

El método que se considera más útil actualmente se llama **Adam** (*ADaptive Moment Estimation*), y lo que hace es almacenar una historia de los últimos pasos y gradientes para calcular una media con decrecimiento exponencial de los últimos pasos y gradientes y así ajustar de forma automática la tasa de aprendizaje y el momento del algoritmo.

Enlace de interés

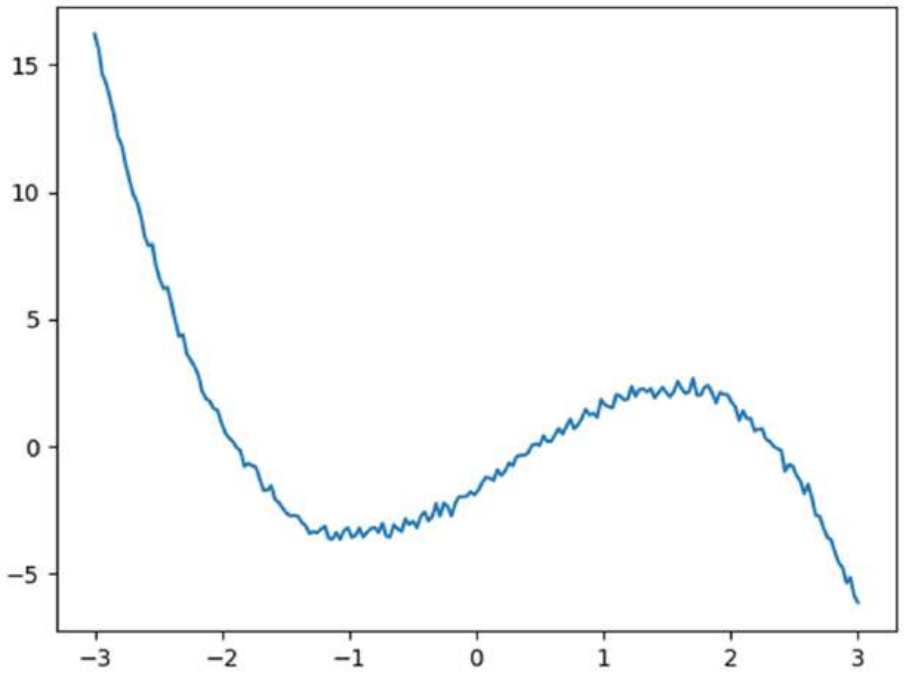
En la página http://ruder.io/optimizing-gradient-descent/ se describen con detalle más aspectos de la configuración del descenso de gradientes.

5.4. Salto de valles

Estamos trabajando en un acelerador de partículas y hemos medido una señal con uno de los detectores, tal y como se muestra en la figura 48.

Necesitamos aproximar la señal mediante una función matemática, y en este caso a la vista de la forma de la señal se decide utilizar un polinomio de grado 3, de la forma $f(x) = ax^3 + bx^2 + cx + d$. El problema que se nos plantea es determinar el valor de los coeficientes a, b, c y d para que se ajusten lo máximo posible a la señal leída.

Figura 48. Señal detectada en el acelerador de partículas



El **salto de valles*** es una metaheurística de optimización **global** cuyo objetivo es encontrar una aproximación razonablemente buena del mínimo global de una función determinada. Hay, por tanto, dos ideas importantes: primero que lo que se pretende encontrar es una aproximación del punto óptimo global, dando por hecho que buscar exhaustivamente el óptimo global no es posible en un plazo de tiempo razonable dado el tamaño del espacio de búsqueda. En segundo lugar, el salto de valles está diseñado para evitar, en la medida de lo posible, quedar atascado en óptimos locales.

*** En inglés, *basin hopping*.**

El salto de valles parte de un estado al azar dentro del espacio de búsqueda y va aplicando cambios en ese estado para ver si su energía (en este caso, la función que se desea minimizar) aumenta o disminuye. En cada salto, realiza una breve exploración del entorno, es decir aplica un método de optimización local como el descenso de gradientes. Con el fin de evitar quedar atascado en mínimos locales, el salto de valles permite cambios a estados de mayor energía (que en principio son peores) para conseguir salir de un mínimo local, en busca de otro valle en el que pueda hallarse el mínimo global. A medida que avanza la ejecución, sin embargo, los empeoramientos que se permiten son cada vez menores, de forma que poco a poco el algoritmo vaya encontrando una aproximación razonablemente buena al mínimo global.

5.4.1. Descripción del método

El objetivo básico del salto de valles es pasar de un estado cualquiera del espacio de búsqueda, con una energía probablemente alta, a un estado de baja energía. El término *energía* se sustituirá, en cada problema, por la medida concreta que se desee minimizar (error, valor de una función, tiempo de respuesta, coste económico, etc.).

Se dice que un estado s' es **vecino** de otro estado s si existe alguna transformación sencilla (mínima) para pasar de s a s'. En el caso de espacios de búsqueda discretos, dicha transformación suele ser evidente; por el contrario, si el espacio de búsqueda es continuo, es necesario definir una distancia máxima de transformación para considerar que dos estados son vecinos.

Para modelar la tolerancia del método a cambios que impliquen un aumento de energía, es decir un empeoramiento de la solución, se considera que el método tiene una cierta **temperatura** inicial T que funciona como una tolerancia a los cambios a estados de mayor energía; la temperatura inicial tiene un valor relativamente alto, que se va reduciendo a medida que avanzan las iteraciones para que finalmente el método acabe estabilizándose. De esta forma, en las iteraciones iniciales es probable que el algoritmo acepte un nuevo estado s' aunque tenga una energía más alta que el estado previo s; sin embargo, a medida que el algoritmo avanza, es menos probable que se acepte un estado con energía más alta. La temperatura, por tanto, decrece a medida que el algoritmo avanza; una formulación habitual es $T = ir * F$, donde ir es el número de iteraciones restantes y F es un factor que permite ajustar la tolerancia con los niveles de energía de los nuevos estados, cuestión que se analiza más adelante.

El primer paso de un algoritmo que utiliza el salto de valles consiste en tomar un estado al azar en el espacio de búsqueda. También es posible tomar un estado conocido que se sepa que es mejor que la media. Denominaremos s al estado actual, y $E(s)$ será su nivel de energía.

En cada iteración del algoritmo se genera aleatoriamente un estado s' vecino del estado actual s; en función del nivel de energía del estado actual $e = E(s)$ y del nuevo estado $e' = E(s')$, el sistema cambia a s' o por el contrario permanece en s. La **probabilidad de aceptación** del nuevo estado viene dada por la expresión:

$$P(e,e',T) = \begin{cases} 1 & \text{si } e' < e \\ \exp((e-e')/T) & \text{si } e' \geq e \end{cases} \quad (63)$$

Como se ve, la temperatura T —y por tanto el número de iteraciones actual— influye en la decisión de aceptar o no un nuevo estado. Recordemos que $T = ir * F$; F se puede utilizar para ajustar la probabilidad de aceptación, dado que según el orden de magnitud de la energía de un estado, un incremento de energía de 1 puede ser muy grande o muy pequeño. Una regla sencilla para ajustar el algoritmo es $i * F \simeq \Delta E_{max}$, siendo i el número total de iteraciones del algoritmo y ΔE_{max} el máximo incremento de energía aceptable, ya que así la probabilidad inicial de aceptación tendrá un valor cercano al 50 %.

En resumen, en las primeras iteraciones es probable que se acepten cambios de estado que impliquen un incremento de la energía (alejándose del objetivo, que es minimizar la energía). De esta forma se favorece que el sistema salga de un mínimo local e intente buscar un mínimo global. Sin embargo, a medida que el algoritmo avanza en su ejecución se va haciendo menos probable que se acepten cambios de estado con incrementos de energía, pues se supone que lo que debe hacer el algoritmo es encontrar el mínimo del entorno en el que se encuentre.

Paralelamente, tras cada iteración se ejecuta un algoritmo de minimización local, que permite localizar el mínimo local de cada valle. Eso permite comparar los valles entre sí mediante sus mínimos, lo que se traduce en un mejor comportamiento del algoritmo.

Una mejora habitual del algoritmo consiste en almacenar el mejor estado encontrado en toda su ejecución, y devolver ese estado como resultado final, aunque no se trate del estado final alcanzado. Es muy sencillo implementar la función y puede mejorar el comportamiento del algoritmo en algunos casos.

5.4.2. Ejemplo de aplicación

Retomando el ejemplo del ajuste de la señal a un polinomio de grado 3, para resolver este problema utilizando salto de valles hay que definir una función objetivo que calcule el error entre la señal leída y el polinomio propuesto por el método. Así, al minimizar ese error, se conseguirá reproducir la señal con la máxima fidelidad. Esta función **objetivo**, a su vez, utiliza una función **polinomio** que a partir de los polinomios y de los valores de x utilizados genera los valores del polinomio correspondiente.

El siguiente código resuelve este problema utilizando la función *basin hopping* de SciPy. Los parámetros que requiere este método son la función objetivo, un punto inicial desde el que empezar a explorar (en el ejemplo generado aleatoriamente) y por último el número de iteraciones máximo que deberá ejecutar.

El resultado de *basin hopping* es un objeto que contiene, entre otros, el valor mínimo encontrado para la función objetivo y el punto en el que se ha

encontrado, es decir, la solución propuesta, en este caso los coeficientes del polinomio.

Código 5.3: aproximación polinómica de una señal con salto de valles

```
import numpy

# Dominio de la funcion
xVal = numpy.linspace(-3, 3, 200)

yVal = numpy.loadtxt('data/signal.data')


# Funcion que aplica un polinomio en un dominio
def polinomio(coefs, dominio):
    a, b, c, d = coefs
    return numpy.array([a*x**3 + b*x**2 + c*x + d for x in dominio])


import math
from sklearn.metrics import mean_squared_error

def objetivo(solution):
    return math.sqrt(mean_squared_error(yVal,
                polinomio(solution, xVal)))/(max(yVal) - min(yVal))


from scipy.optimize import basinhopping

# Initial guess: let it be random (4 parameters in solution)
from random import random
punto0 = [random()*2-1 for i in range(4)]

opti = basinhopping(func=objetivo, x0=punto0, niter=100)

print('Error minimo:', opti.fun)
print('Coeficientes:', opti.x)
```

La figura 49 compara la señal leída originalmente con la aproximación polinómica obtenida por el método. Como se puede observar, el polinomio sigue con la máxima fidelidad posible los valores de la señal.

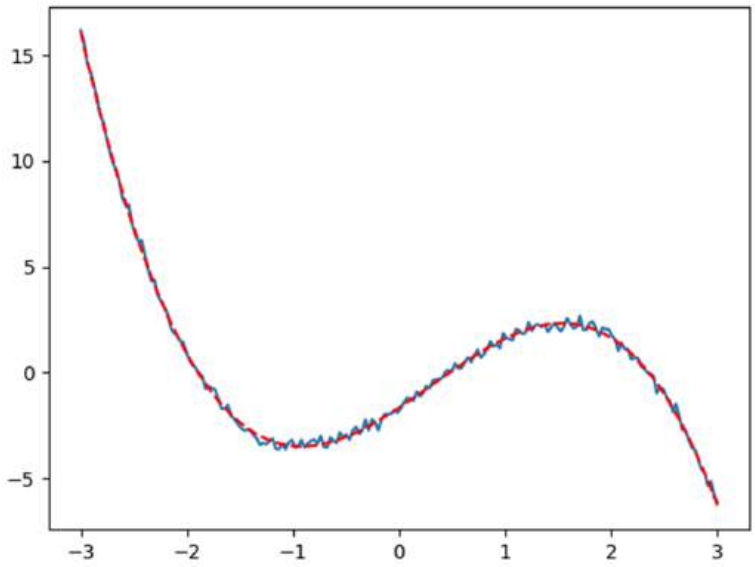
5.4.3. Análisis del método

Función no convexa

Una función objetivo con diferentes mínimos locales se denomina función no convexa.

El salto de valles es aplicable a numerosos problemas de optimización en los que se busque el mínimo. Dado que se trata de un método de optimización **global**, está especialmente indicado para problemas en los que existen mínimos locales. Además, gracias al paso de optimización local en cada iteración, es capaz de encontrar siempre el mínimo punto disponible en cada valle, lo que al menos garantiza la obtención de un mínimo local y la comparación de unos valles con otros por medio de sus mínimos respectivos. Otra de sus principales ventajas es que no requiere elegir muchos parámetros de funcionamiento, a diferencia de otros métodos de optimización.

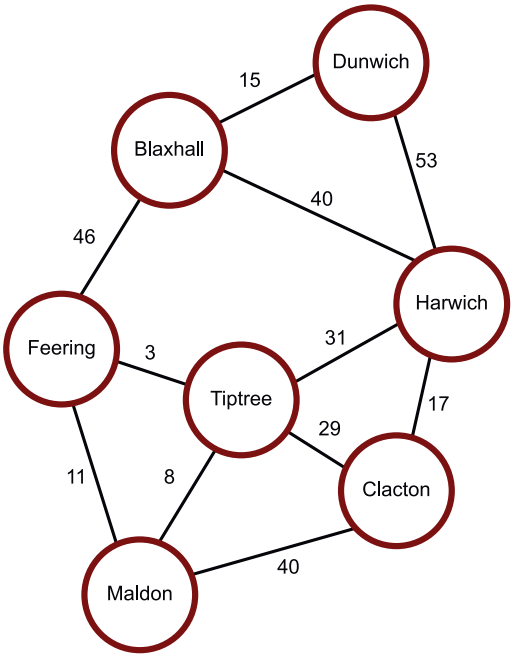
Figura 49. Aproximación de la señal del acelerador de partículas



5.5. Algoritmos genéticos

El problema del **viajante de comercio** (*travelling salesman problem*, o simplemente TSP) es un problema clásico de optimización. En este problema hay un conjunto de ciudades, con una cierta distancia entre cada par de ellas. El viajante debe visitarlas todas una y solo una vez. La solución óptima debería encontrar el orden en el que visitar las ciudades de forma que la distancia recorrida por el viajante sea la mínima posible, para ahorrar tiempo, combustible, etc. La figura 50 muestra un ejemplo de este problema.

Figura 50. Ejemplo del problema del viajante de comercio



Fuente: wikimedia.org

Independientemente de este enunciado «clásico», el problema TSP es un problema tipo que se puede aplicar a numerosas situaciones en las que haya que optimizar el orden de un conjunto de elementos. Expresándolo formalmente, este problema propone encontrar el camino más corto para recorrer todos los nodos de un grafo ponderado. Este tipo de problemas se denominan problemas **combinatorios**. Resolver este tipo de problemas, a pesar de su aparente simplicidad, puede resultar extremadamente costoso computacionalmente. Por ejemplo, en TSP el espacio de soluciones es el espacio de las diferentes ordenaciones posibles de n ciudades, y el número de ordenaciones posibles es $n!$ Para un número medianamente grande de ciudades es imposible computacionalmente explorar exhaustivamente todas las combinaciones.

La mayoría de métodos de optimización no pueden resolver problemas combinatorios adecuadamente. En este subapartado estudiaremos una familia de métodos de optimización, los **algoritmos genéticos**, que son adecuados para resolver problemas de optimización clásicos y combinatorios. Además, debido a su mayor riqueza y complejidad, los aplicaremos a varios problemas para poder ilustrar diferentes aspectos teóricos y prácticos.

Los seres vivos exhiben una enorme diversidad y una asombrosa adaptación a los entornos más diversos. Esto es posible mediante los mecanismos de la **selección natural**, descritos por Darwin en el siglo XIX. La idea central de su **teoría de la evolución** es que los organismos mejor adaptados al medio son los que tienen más posibilidades de sobrevivir y dejar descendencia, y por lo tanto son más abundantes que los peor adaptados, que posiblemente acaben extinguiéndose.

Las características de un organismo están determinadas por su **información genética**, que se hereda de padres a hijos. De esta manera, las características que ayudan a la supervivencia son las que se heredan con mayor probabilidad. En el caso de la reproducción sexual, la información genética del padre y la de la madre se combinan de manera aleatoria para formar la información genética de los hijos. Es lo que se llama el **cruzamiento**. Se trata de un proceso muy complejo, dado el gran volumen de información transferido, por lo que en raras ocasiones se produce un error de copia, dando lugar a una **mutación**, que en adelante heredarán los descendientes del portador.

La información genética de un individuo se divide en **genes**, que son secuencias de información genética que describen una característica del individuo*.

***Téngase en cuenta que se trata de una descripción muy simplificada. La realidad es bastante más compleja.**

Los **algoritmos genéticos** emulan la selección natural sobre un conjunto de individuos para buscar la mejor solución a un problema determinado. La "información genética" de cada individuo es una posible solución al problema; por analogía, hay un "gen" para cada variable o parámetro del problema sobre el que se desea ejecutar el proceso de optimización. Para emular la selección natural, se crea una **población** o conjunto de individuos y se le hace evolu-

cionar de forma que los mejor adaptados, o sea, los que son mejor solución para el problema, se reproduzcan con mayor probabilidad y poco a poco vayan surgiendo individuos mejor adaptados al problema; en otras palabras, mejores soluciones.

5.5.1. Descripción del método

Sea Q el problema que queremos resolver, y S el espacio de las soluciones de Q, en el que cada solución $s \in S$ se compone de n variables de la forma $s = \{x_1,x_2,x_3,...,x_n\}$. Esas variables pueden ser de cualquier tipo de datos; los más habituales en este caso son binarios, enteros y reales, o una combinación de los mismos. Además, cada variable tiene un rango de valores propio. En un algoritmo genético, cada individuo es una solución $s_i \in S$ con n variables (o genes, si seguimos la metáfora biológica).

También existe una función $f : S \to \mathbb{R}$ que mide la idoneidad* de una solución $s \in S$, y que en definitiva es la función que queremos minimizar o maximizar. En el ámbito de la selección natural, la idoneidad de un individuo corresponde a su adaptación al medio; los mejor adaptados sobrevivirán con mayor probabilidad.

*En inglés *fitness*

El punto de partida de un algoritmo genético es una población P formada por k individuos tomados al azar de S. A continuación, se evalúa la idoneidad de cada solución $s \in P$ utilizando f, y los individuos se ordenan de menor a mayor idoneidad (creciendo numéricamente si se desea maximizar f, decreciendo si se desea minimizarla).

Cada iteración de un algoritmo genético recibe el nombre de **generación**, nuevamente por analogía con los organismos vivos. En cada generación se toma la población existente, ordenada por idoneidad, y se genera una nueva población mediante el **cruce** de los individuos existentes dos a dos. Para emular la selección natural, los individuos más idóneos se reproducen con mayor probabilidad que los menos idóneos; de esta forma es más probable que sus genes se transmitan a la siguiente generación de individuos.

Hay diferentes métodos para cruzar los genes de dos individuos A y B: cortar por un gen determinado de A e insertar los genes de B a partir de ese punto (cruce en un punto, ver figura 51); o cortar por dos puntos de A e insertar los genes de B correspondientes (cruce en dos puntos, que es el método que usaremos en nuestro ejemplo, ver figura 52); también es posible cruzar al azar cada uno de los genes (cruce uniforme, ver figura 53), entre otros métodos. Con frecuencia el método de cruce ha de tener en cuenta la naturaleza del problema, pues puede ocurrir que sobre los genes operen algunas restricciones: que deban estar ordenados, que no deban repetirse, que su suma no deba exceder una cierta cantidad, etc. El objetivo final del cruce es, emulando a

la naturaleza, combinar las características ventajosas de dos individuos para conseguir un individuo mejor adaptado incluso a su entorno.

Figura 51. Cruce en un punto

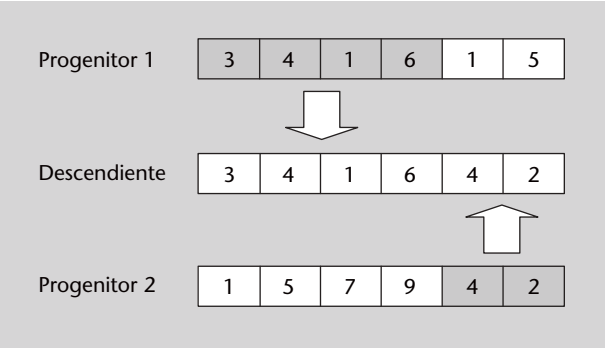


Figura 52. Cruce en dos puntos

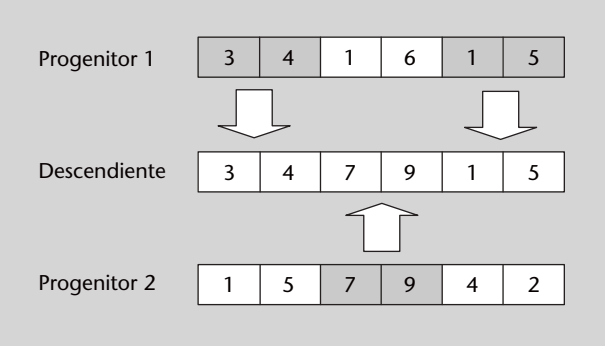
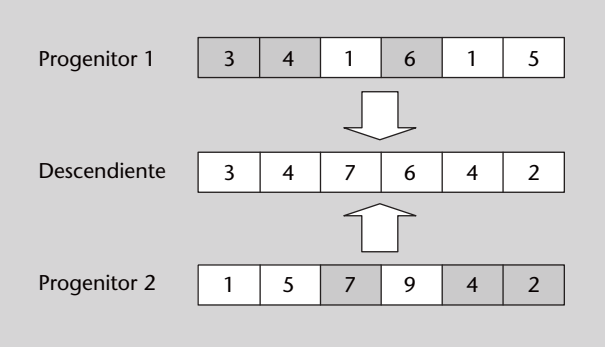


Figura 53. Cruce uniforme



El otro mecanismo que interviene en la evolución biológica, y que los algoritmos genéticos también utilizan, es el de la **mutación**. Los individuos que se han generado por cruce pueden sufrir, con una determinada probabilidad, un cambio aleatorio en alguno(s) de sus genes. El objetivo de las mutaciones es enriquecer la población introduciendo genes que no se encuentren en ella, lo que permite a su vez explorar nuevas zonas del espacio de soluciones *S* que pueden contener soluciones mejores. La probabilidad o tasa de mutación no suele ser muy alta. En el ejemplo presentado a continuación es de un 1 %, a modo de orientación.

Seguidamente se evalúa la idoneidad de esta nueva población, que a su vez dará nacimiento a una nueva generación de individuos, y así sucesivamente. No existe un criterio único para establecer el número de generaciones (iteraciones) de un algoritmo genético; puede ser fijo, o hasta alcanzar una cierta idoneidad, o hasta que los nuevos individuos no aporten ninguna mejora durante algunas generaciones.

Este último paso del algoritmo se denomina **selección**, y a grandes rasgos corresponde a la selección natural de individuos que ocurre en el ámbito biológico. Dado que algunos individuos están mejor adaptados que otros, es de esperar que los mejor adaptados se reproduzcan con mayor probabilidad. Sin embargo, tampoco es conveniente que solo los mejor adaptados se reproduzcan, ya que se corre el riesgo de perder la biodiversidad de la población y acabar teniendo muchas soluciones idénticas entre sí. Por ese motivo se plantean estrategias de selección equilibradas. Una de las más utilizadas es la del **torneo**, en la que se eligen dos o más individuos al azar, y se selecciona al mejor adaptado de ellos. Así los mejor adaptados se seleccionan con mayor probabilidad, pero los menos adaptados tienen oportunidades de reproducirse.

A partir de los individuos seleccionados para dar lugar a la nueva generación se aplican las técnicas de cruce y mutación para generar a los descendientes.

5.5.2. Ampliaciones y mejoras

Existen numerosas variantes y mejoras de los algoritmos genéticos. Una de las más habituales es la del **elitismo**: en cada generación, los E mejores individuos pasan inalterados a la siguiente generación. De esta forma el algoritmo no pierde soluciones que pueden ser mejores que la solución final, y además permite probar numerosos cruces y mutaciones de las mejores soluciones. Frecuentemente se toma $\mid E \mid = 1$, como en el ejemplo desarrollado más adelante.

Otras propuestas para mejorar los algoritmos genéticos incluyen el cruce entre más de dos individuos y los algoritmos **meméticos**, en los que los individuos tienen la capacidad de aprender y transmitir lo aprendido a sus descendientes.

También existe un área denominada **programación genética**, semejante a los algoritmos genéticos pero cuyo objetivo es generar programas (no simplemente encontrar valores) que resuelvan un determinado problema mediante evolución y selección natural.

5.5.3. Ejemplos de aplicación

En este apartado se hará una breve introducción a la biblioteca **Distributed Evolutionary Algorithms in Python (DEAP)**, que actualmente es la librería

más completa, flexible y al día de algoritmos genéticos y programación evolutiva en Python.

Concretamente, se verá cómo:

- instalar DEAP,
- crear, configurar y ejecutar un algoritmo genético que optimiza un problema sencillo,
- recopilar estadísticas de la ejecución del algoritmo y visualizar su evolución,
- usar DEAP para modelar y resolver un problema combinatorio,
- aplicar restricciones para generar soluciones válidas a los problemas planteados.

Cada uno de los subapartados siguientes corresponde a uno de los objetivos anteriores. Para facilitar la comprensión los programas se presentarán fragmento a fragmento junto con la explicación correspondiente.

Instalar DEAP

El paquete DEAP se encuentra en el repositorio oficial de Python (PyPI), así que se puede instalar simplemente usando el comando `pip`. En Linux simplemente hay que abrir una terminal y escribir:

```
1 pip3 install deap
```

o bien

```
1 sudo pip3 install deap
```

En Windows hay que acceder a la carpeta de ejecutables de Python, donde se encontrará el programa **pip.exe**. Hay que abrir una terminal en él (mayúsculas + botón derecho en la carpeta y «Abrir ventana de comandos aquí») y escribir:

```
1 pip.exe install deap
```

Enlace de interés

Para más detalles o información actualizada se recomienda visitar la página oficial de DEAP: http://bit.ly/2kkdHgY.

Primer algoritmo genético con DEAP

DEAP es una biblioteca de programación evolutiva muy flexible, pero la contrapartida a esa flexibilidad es que hay que definir una serie de elementos antes de poder resolver cualquier problema de optimización; no es posible escribir una llamada a una función y ya está.

En este primer ejemplo aprenderemos cómo configurar un algoritmo genético con DEAP aplicándolo a un problema muy sencillo llamado **OneMax**, en el que tenemos una cadena de bits y el algoritmo debe encontrar la combinación de valores que dé la suma más alta. Lógicamente la solución es una cadena en la que todo sean unos. Sin embargo, el algoritmo no sabe eso y simplemente irá explorando soluciones hasta dar con la mejor. Así, OneMax es un buen ejemplo para probar algoritmos de optimización, ya que la solución es conocida. En este ejemplo tomaremos cadenas de 50 bits.

El planteamiento de este problema mediante algoritmos genéticos es sencillo: cada individuo es una cadena de bits.

1) Importaciones. DEAP contiene una serie de submódulos que hay que importar. Además, se importa el módulo *random* para generar la población inicial de individuos (los bits de la cadena). Dentro de DEAP, el submódulo *creator* permite crear individuos y poblaciones, *base* contiene los elementos básicos de DEAP, *tools* contiene herramientas para el cruzamiento y la mutación de individuos, y *algorithms* contiene los propios algoritmos genéticos.

```
1 import random
2 from deap import creator, base, tools, algorithms
```

2) Definición de los individuos y de las poblaciones. El primer paso con DEAP consiste en definir el formato de los individuos (soluciones) y crear una población con ellos. Para definir un individuo, se necesitan dos instrucciones. La primera instrucción especifica el **objetivo** del individuo: generalmente minimizar o maximizar una función dada; se utilizará respectivamente `FitnessMin` o `FitnessMax`. Por último, el argumento *weights* es una tupla de pesos de cada uno de los objetivos (DEAP permite optimizar funciones multiobjetivo). Los pesos utilizados deben ser positivos para objetivos que se quieran maximizar, y negativos para objetivos que se deban minimizar.

La segunda instrucción define la **estructura** de los individuos, en este caso les da un nombre (`Individuo`), especifica que serán una lista de atributos y finalmente los asocia con el objetivo anterior.

> Los pesos de los objetivos siempre deben ser de tipo tupla, de ahí la coma suelta que hay al final cuando hay un único objetivo.

```
1 creator.create('FitnessMax',base.Fitness,weights=(1.0,)) # TUPLA!!!
2 creator.create('Individuo', list, fitness=creator.FitnessMax)
```

A continuación se crea una caja de herramientas (*toolbox*) para poder acceder a otras operaciones y para crear la población inicial. En primer lugar se definen

los atributos (bits, en este ejemplo) indicando cómo se deben crear, en este caso con la función *randint* y parámetros 0 y 1. En segundo lugar, se da un nombre a los individuos y se especifica cómo crear sus atributos, en este caso repitiendo cincuenta veces el atributo binario. Por último, se define la población, dándole un nombre e indicando cómo crear a los individuos, en este caso repitiendo la creación de un individuo y almacenándolos en una lista.

```
toolbox = base.Toolbox()
toolbox.register('attrBool', random.randint, 0, 1)
toolbox.register('individuo', tools.initRepeat, creator.Individuo,
                 toolbox.attrBool, n=50)
toolbox.register('population', tools.initRepeat, list,
                 toolbox.individuo)
```

3) Función objetivo. La estructura de la función objetivo, la que evalúa la calidad de una solución, es la siguiente: la entrada es un individuo y la salida es el valor del objetivo (u objetivos) para esa solución. Es decir, valora la calidad de una solución y, en definitiva, es la función que se quiere minimizar o maximizar. En el problema OneMax se quiere maximizar la suma de los bits, por tanto:

```
def evalOneMax(individuo):
    return sum(individuo),    # COMA AL FINAL, DEBE SER UNA TUPLA!!
```

Es muy importante no olvidar la coma al final para que el resultado sea siempre una tupla.

4) Estrategias. El último paso antes de ejecutar el algoritmo es configurar sus estrategias para el cruzamiento, la mutación y la selección de individuos. También se le indica cuál es la función utilizada para evaluar a un individuo. La función de cruzamiento (*mate*) en este caso es la de cruzamiento en dos puntos; la de mutación (*mutate*) lo que hace es cambiar bits con una probabilidad de 0.05; la selección se realiza mediante un torneo entre tres individuos.

```
toolbox.register('evaluate', evalOneMax)
toolbox.register('mate', tools.cxTwoPoint)
toolbox.register('mutate', tools.mutFlipBit, indpb = 0.05)
toolbox.register('select', tools.selTournament, tournsize=3)
```

5) Ejecución del algoritmo genético. Para ejecutar el algoritmo genético se debe crear la población, en este caso con cien individuos, a continuación se indica el número de generaciones y a continuación se inicia un bucle `for` para iterar.

Se recomienda que el tamaño de la población sea del mismo orden de magnitud que el número de variables del problema.

El bucle principal lleva a cabo cinco tareas:

1) generar una nueva población a partir de la anterior, en este caso con una probabilidad de cruzamiento de cada progenitor de 0.5 (`cxpb`) y una probabilidad de que un individuo entre en el proceso de mutación de 0.1 (`mutpb`);

2) calcular la función objetivo sobre cada individuo (es decir, su adaptación al problema);

3) asociar la adaptación con los individuos;

4) reemplazar la población anterior con la nueva;

5) (opcional) seleccionar el mejor individuo y mostrar su adaptación y atributos para monitorizar el algoritmo.

```
poblacion = toolbox.population(100)

NGEN = 40

for gen in range(NGEN):
    descendientes = algorithms.varAnd(poblacion, toolbox, cxpb=0.5,
                                      mutpb=0.1)
    adaptaciones  = toolbox.map(toolbox.evaluate, descendientes)
    for adap, indiv in zip(adaptaciones, descendientes):
        indiv.fitness.values = adap

    poblacion = toolbox.select(descendientes, k=len(poblacion))

    mejor = tools.selBest(poblacion, k=1)
    print(gen, evalOneMax(top[0]), top[0])
```

Al ejecutar el programa se puede ver el valor de adaptación y los atributos correspondientes del mejor individuo de cada generación. Se puede comprobar fácilmente si la adaptación es igual a 50 (el máximo posible) y todos los bits son 1.

```
...
36 (50,) [1, 1, 1, 1, 1, ..., 1, 1, 1, 1, 1, 1, 1, 1, 1]
37 (50,) [1, 1, 1, 1, 1, ..., 1, 1, 1, 1, 1, 1, 1, 1, 1]
38 (50,) [1, 1, 1, 1, 1, ..., 1, 1, 1, 1, 1, 1, 1, 1, 1]
39 (50,) [1, 1, 1, 1, 1, ..., 1, 1, 1, 1, 1, 1, 1, 1, 1]
>>>
```

5.5.4. Recopilación de estadísticas

El objetivo del siguiente ejemplo es mostrar cómo recopilar estadísticas de la ejecución de un algoritmo genético, como por ejemplo el valor objetivo mínimo, medio y máximo en cada generación. Eso puede resultar de ayuda para ajustar los parámetros del algoritmo.

El problema por resolver es bastante sencillo, de hecho ya se vio en el apartado 5.4, de salto de valles: se ha leído una señal de un acelerador de partículas y se desea aproximar mediante un polinomio de grado 3 de la forma ax^3+bx^2+cx+d.

El algoritmo debe encontrar los coeficientes que minimicen el error entre la señal leída y el polinomio.

Configuración del algoritmo

Los componentes que hay que configurar en este algoritmo son básicamente los mismos que en el ejemplo anterior de OneMax, las principales diferencias son que en este caso los atributos son números reales y el objetivo es minimizar el error, por lo que el peso del objetivo es negativo.

```
import random
from deap import creator, base, tools, algorithms

creator.create('FitnessMin', base.Fitness, weights=(-1.0,))
creator.create('Individual', list, fitness=creator.FitnessMin)

toolbox = base.Toolbox()

toolbox.register('attrFloat',lambda:random.random()*2-1)  #[-1,1]
toolbox.register('individual', tools.initRepeat, creator.Individual,
                 toolbox.attrFloat, n=4)
toolbox.register('population', tools.initRepeat, list,
                 toolbox.individual)
```

La función objetivo mide la diferencia entre la señal y el polinomio propuesto. Como en la solución de salto de valles, se recurre a una función que genera los valores del polinomio.

```

import numpy

xVal = numpy.linspace(-3, 3, 200)
yVal = numpy.loadtxt('data/signal.data')

def polinomio(coefs, dominio):
    a, b, c, d = coefs
    return numpy.array([a*x**3+b*x**2+c*x + d for x in dominio])

import math
from sklearn.metrics import mean_squared_error

def objetivo(solucion):
    return math.sqrt(mean_squared_error(yVal,
            polinomio(solucion)))/(max(yVal) - min(yVal)), #COMA!
```

Respecto a la configuración del algoritmo genético, usaremos cruzamiento en un solo punto, ya que solo hay cuatro atributos. Además, en este caso la mutación es Gaussiana (añade un valor aleatorio al atributo) con media 0 y desviación 0.5. La población contiene cuarenta individuos y el algoritmo se ejecuta durante sesenta generaciones.

```

toolbox.register('evaluate', objetivo)
toolbox.register('mate',     tools.cxOnePoint)
```

```
toolbox.register('mutate',   tools.mutGaussian, mu=0, sigma=0.5,
                 indpb = 0.2)
toolbox.register('select',   tools.selTournament, tournsize=3)


poblacion = toolbox.population(40)

NGEN = 60
```

Usando el módulo de estadística

Para recopilar estadísticas del algoritmo se puede usar el módulo *Statistics* creando un *logbook*, que es una estructura de datos que registra los resultados en cada iteración y permite mostrarlos al terminar.

```

# Statistics
stats = tools.Statistics(key=lambda ind: ind.fitness.values)
stats.register('min', numpy.min)
stats.register('avg', numpy.mean)
#stats.register('std', numpy.std)
#stats.register('max', numpy.max)

logbook = tools.Logbook()
```

Dentro del bucle principal se debe llamar a `stats.compile` y `logbook.record` al generar cada nueva población para recopilar los datos.

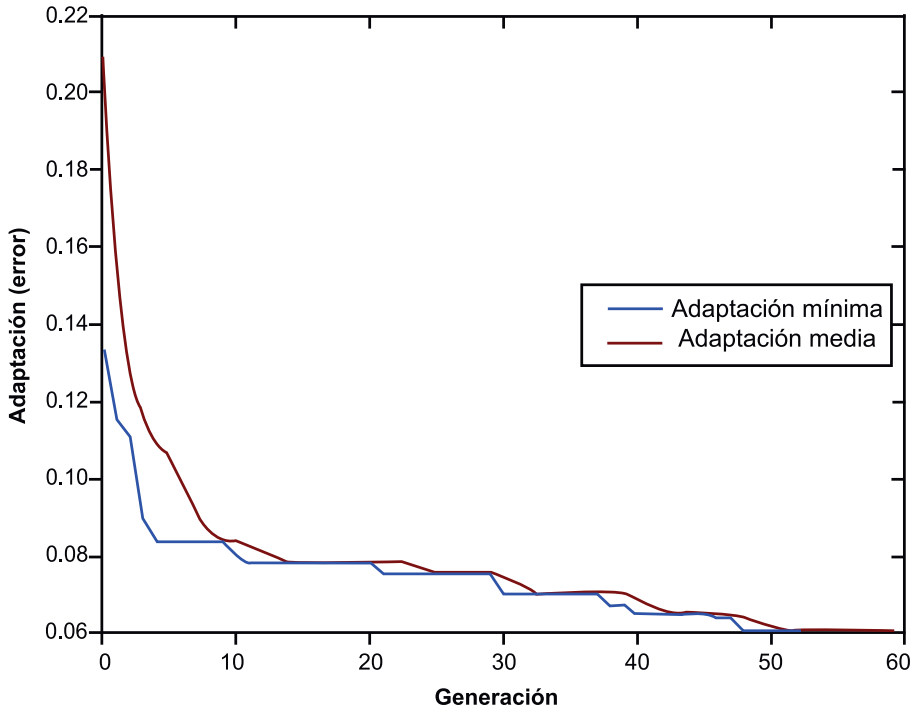
```
for gen in range(NGEN):
    descendientes = algorithms.varAnd(poblacion, toolbox, cxpb=0.5,
                                      mutpb=0.1)
    adaptaciones  = toolbox.map(toolbox.evaluate, descendientes)
    for fit, ind in zip(adaptaciones, descendientes):
        ind.fitness.values = fit

    poblacion = toolbox.select(descendientes, k=len(poblacion))

    # Stats recording
    registro = stats.compile(poblacion)

    logbook.record(gen=gen, **registro)
```

Al terminar el bucle, los datos recopilados se pueden recoger utilizando la expresión `logbook.select`. En el siguiente ejemplo se dibuja una gráfica con la adaptación mínima y la media a lo largo de las generaciones, tal y como se muestra en la figura 54.

```
# Gather the data collected and plot it
generation = logbook.select('gen')
fitnessMin = logbook.select('min')
fitnessAvg = logbook.select('avg')

import matplotlib.pyplot as plt

line1 = plt.plot(generation, fitnessMin, "b-",
                 label="Adaptacion minima")
```

```
plt.xlabel("Generacion")
plt.ylabel("Adaptacion (error)")

line2 = plt.plot(generation, fitnessAvg, "r-",
                 label="Adaptacion media")

lns = line1 + line2
labs = [l.get_label() for l in lns]
plt.legend(lns, labs, loc="center right")

plt.show()
```

Figura 54. Evolución del algoritmo genético al aproximar una señal



5.5.5. Problemas combinatorios

Los algoritmos genéticos son uno de los métodos más indicados para resolver problemas combinatorios como el del viajante de comercio expuesto al inicio. Sin embargo, es necesario configurarlos de una forma muy específica para que realmente se adecúen a la naturaleza combinatoria del problema. Concretamente, para el TSP, hay que tener en cuenta que:

- Una solución es una permutación (ordenación) de todas las ciudades.
- Todas las ciudades deben aparecer una y solo una vez en cada solución.
- Por tanto, las mutaciones deben intercambiar dos ciudades, no modificar una aleatoriamente.
- Y el cruzamiento debe mantener todas las ciudades una y solo una vez en los nuevos individuos.
- Para conseguir soluciones lo más genéricas posible, representaremos las ciudades por sus índices (0, 1, ..., $n-1$).

A continuación, veremos cómo configurar DEAP para resolver este tipo de problemas*.

*** Adaptado de http://bit.ly/2AKSSSD**

En primer lugar, se genera un conjunto de ciudades con coordenadas (*x*,*y*). La distancia entre ciudades es la distancia euclídea.

```
import random, math

# Para generar siempre las mismas ciudades
random.seed(0)

def distance(cityA, cityB):
    return math.sqrt((cityA[0]-cityB[0])**2+(cityA[1]-cityB[1])**2)

# Las coordenadas x e y van de 0 a 500
def generateCities(n):
    return [(random.randint(0,500), random.randint(0,500))
            for c in range(n)]

numCities = 30
cities    = generateCities(numCities)
print(cities)
```

La generación de una solución usa una permutación de los índices de las ciudades. La estrategia de cruzamiento se llama `cxOrdered`, que mantiene una y solo una copia de cada índice. La mutación intercambia dos índices, y es la función *mutShuffleIndexes*.

```
from deap import algorithms, base, creator, tools
import numpy

toolbox = base.Toolbox()
creator.create('FitnessMin', base.Fitness, weights=(-1.0,))
creator.create('Individual', list, fitness=creator.FitnessMin)

toolbox.register('Indices', numpy.random.permutation, len(cities))
toolbox.register('Individual', tools.initIterate,
                 creator.Individual, toolbox.Indices)
toolbox.register('Population', tools.initRepeat, list,
                 toolbox.Individual)

toolbox.register('mate', tools.cxOrdered)
toolbox.register('mutate', tools.mutShuffleIndexes, indpb=0.05)
```

Finalmente, la función objetivo calcula la distancia total recorrida dado el orden de los índices. El algoritmo genético se configura y se ejecuta como es habitual.

```

# Distancia total recorrida
def totalDistance(individual):
    return sum((distance(cities[individual[i]],
                cities[individual[i-1]])
                for i in range(len(individual)))),


# Ejecutar el algoritmo
toolbox.register('evaluate', totalDistance)
toolbox.register('select', tools.selTournament, tournsize=3)

```

```
population = toolbox.Population(100)
NGEN = 100

for gen in range(NGEN):
    offspring = algorithms.varAnd(population, toolbox, cxpb=0.5,
                                  mutpb=0.1)
    fits      = toolbox.map(toolbox.evaluate, offspring)
    for fit, ind in zip(fits, offspring):
        ind.fitness.values = fit

    population = toolbox.select(offspring, k=len(population))

    top = tools.selBest(population, k=1)
    print(gen, totalDistance(top[0]), top[0])

print('='*40)
top10 = tools.selBest(population, k=10)
for ind in top10:
    print(totalDistance(ind), ind)
```

5.5.6. Problemas con restricciones

Los problemas de optimización a menudo imponen ciertas restricciones a las soluciones posibles. Es importante tener en cuenta esas restricciones para no desperdiciar tiempo de computación explorando soluciones que no son aceptables; en otras palabras, las restricciones ayudan a acotar el espacio de soluciones, y por tanto a disminuir su tamaño.

Supongamos el siguiente problema: disponemos de una cierta cantidad de dinero (50.000) que queremos invertir en el mercado de acciones, de manera que maximicemos el beneficio de la inversión. Hay diez tipos de acciones diferentes, cada una con un precio base y un precio de venta, y podemos comprar tantas unidades como queramos de cada una.

```
preciosCompra = [100, 80, 90, 130, 105, 60, 80, 40, 140, 110]
preciosVenta  = [112, 95, 99, 140, 113, 77, 88, 53, 146, 121]
dinero        = 50000
```

La cuestión es la siguiente: aunque el precio de venta es constante, el precio de compra no lo es; concretamente, si compramos n unidades de una acción, su precio de compra efectivo vendrá dado por la expresión $precioCompraReal = precioCompra(1 + n/1000)$. Por ejemplo, si compramos sesenta unidades de la primera acción, su precio de compra efectivo será $100(1 + 60/1.000) = 106$, de forma que habrá que gastarse $60 \cdot 106 = 6330$ para comprarlas.

Así que el objetivo del problema es encontrar la cantidad óptima que se debe comprar de cada acción para que el beneficio sea máximo. Asumimos que una solución será una lista de enteros, es decir, el número de acciones de cada tipo que se compran. Pero como solo disponemos de 50.000 euros, el total gastado no puede superar esa cantidad (aunque puede ser inferior). Esa es la restricción de este problema.

Hay tres estrategias para tratar con restricciones en los problemas de optimización:

1) Codificar las soluciones del problema de una forma tal que sea imposible incumplir las restricciones. En este caso esa codificación implicaría tomar cada número de la solución como el porcentaje de dinero que se debe invertir en cada acción. Sin embargo, sería una codificación poco intuitiva y que daría soluciones subóptimas, ya que daría lugar a cantidades no enteras de acciones que habría que redondear.

2) Añadir una penalización en la función objetivo a las soluciones que incumplan las restricciones, por ejemplo restando al beneficio el dinero gastado en exceso, de forma que naturalmente el método evite esas soluciones.

3) Instruir al algoritmo genético para comprobar que los nuevos individuos creados en cada generación (tras el cruzamiento y la mutación) cumplen las restricciones, y corregir los individuos que no lo hacen. Esta es la aproximación mostrada en este apartado.

La *toolbox* de DEAP permite añadir **decoradores**, que son funciones que envuelven (*wrappers*) las funciones de cruzamiento y de mutación para que todos los individuos generados se hayan verificado. Dentro de esas funciones decoradoras, los individuos que incumplen una restricción son forzados a cumplirla, es decir, se devuelven al espacio de soluciones. En este problema, la estrategia es eliminar aleatoriamente una unidad de compra de una acción de forma repetida hasta que el dinero gastado está dentro de los límites.

La configuración inicial del algoritmo es similar a la de los ejemplos anteriores, con individuos que son una lista de enteros entre 0 y 200, cruzamiento en un punto y mutaciones que asignan un valor aleatorio entre 0 y 200.

```
1  import random
2  from deap import creator, base, tools, algorithms
3
4  creator.create('FitnessMax', base.Fitness, weights=(1.0,))
5  creator.create('Individual', list, fitness = creator.FitnessMax)
6
7  toolbox = base.Toolbox()
8
9  toolbox.register('attrInt', lambda: random.randint(0,200))
10 toolbox.register('individual', tools.initRepeat, creator.Individual,
11                 toolbox.attrInt, n=len(preciosCompra))
12
13 toolbox.register('population', tools.initRepeat, list,
14                 toolbox.individual)
15
16 toolbox.register('evaluate', objetivo)
17 toolbox.register('mate', tools.cxOnePoint)
18 toolbox.register('mutate', tools.mutUniformInt, low=0, up=200,
19                 indpb=0.1)
20 toolbox.register('select', tools.selTournament, tournsize=3)
```

La función objetivo calcula el beneficio de las inversiones, teniendo en cuenta el aumento del precio de compra con la cantidad:

```
def objetivo(solucion):
    return sum([amt*(sp - bp*(1+amt/1000))
                for amt, bp, sp in zip(solucion, preciosCompra,
                                       preciosVenta)]),
```

Si se ejecuta este programa tal cual, se puede obtener un resultado como el siguiente:

```
499 beneficio= (4431.66,) gastado= (59287.34,)
inversiones= [60, 96, 50, 38, 40, 142, 50, 163, 21, 50]
```

Así que la mejor solución no es aceptable porque gasta más dinero del disponible. Para añadir una restricción a la solución, hay que definir la función decoradora como la siguiente. Nótese que en realidad son tres funciones anidadas, y a la más exterior le podemos dar el nombre que convenga:

```
def comprueba(presupuesto):
    def decorator(func):
        def wrapper(*args, **kargs):
            descendientes = func(*args, **kargs)
            for hijo in descendientes:
                while gastado(hijo) > presupuesto:
                    pos = random.randint(0, len(hijo)-1)
                    # Hay que evitar atributos negativos!
                    hijo[pos] = max(hijo[pos]-1, 0)
            return descendientes
        return wrapper
    return decorator
```

Básicamente comprueba todos los individuos (`hijo`) de la nueva población (`descendientes`) y les resta aleatoriamente una unidad de una acción hasta que el dinero gastado está dentro de los límites. Por cierto, la función *gastado* simplemente suma el total de dinero gastado:

```
def gastado(solucion):
    return sum([amt*bp*(1+amt/1000)
                for amt, bp in zip(solucion, preciosCompra)])
```

Finalmente, hay que decirle a DEAP que ejecute la comprobación en la población inicial y tras cada cruzamiento y mutación de la manera siguiente:

```
toolbox.decorate('population', comprueba(dinero))
toolbox.decorate('mate',       comprueba(dinero))
toolbox.decorate('mutate',     comprueba(dinero))
```

Una vez configurado todo lo anterior, el bucle principal del algoritmo es similar al de los ejemplos anteriores:

```
population = toolbox.population(100)
NGEN = 100

for gen in range(NGEN):
    offspring = algorithms.varAnd(population, toolbox, cxpb=0.5,
                                  mutpb=0.1)
    fits      = toolbox.map(toolbox.evaluate, offspring)

    for fit, ind in zip(fits, offspring):
        ind.fitness.values = fit

    population = toolbox.select(offspring, k=len(population))

    # Mostrar la informacion en cada iteracion
    top = tools.selBest(population, k=1)
    print(gen, 'Beneficio=', objective(top[0]), 'gastado=',
          spent(top[0]), 'inversiones=', top[0])
```

De forma que la salida del programa sea algo como:

```
499 beneficio= (4360.52,) gastado= 49998
inversiones= [53, 85, 40, 27, 30, 133, 41, 156, 13, 43]
```

5.5.7. Análisis del método

Loa algoritmos genéticos son aplicables a gran número de problemas de optimización, y por lo general encuentran soluciones razonablemente buenas en un tiempo aceptable. Son, por tanto, una opción a considerar siempre que se plantee un problema de optimización.

No obstante, adolecen de algunos problemas. El principal es que dependen de un gran número de parámetros y operaciones (número de individuos, tasa de mutación, estrategia de cruzamiento) y no hay reglas que indiquen claramente qué valores elegir para cada problema. Por tanto, hay que ajustar esos parámetros siguiendo recomendaciones muy generales.

Otra crítica que reciben los algoritmos genéticos es que, al proponer un gran número de soluciones al problema, se requiere calcular la función de idoneidad muchas veces; a menudo esa función es muy costosa en tiempo de computación, lo que hace que utilizar un algoritmo genético requiera mucho tiempo de ejecución.

5.6. Colonias de hormigas

El **problema del viajante de comercio** es un problema clásico de optimización en el que se dispone de un conjunto de ciudades, con una distancia entre

cada par de ciudades. El viajante al que se refiere el problema debe visitar todas las ciudades una y sólo una vez, intentando por su parte recorrer la mínima distancia (para gastar menos tiempo, combustible, etc.).

El algoritmo de optimización mediante colonias de hormigas* es un algoritmo probabilístico que imita la habilidad de las hormigas para encontrar el camino más corto desde su hormiguero hasta una fuente de alimento. Las hormigas son un excelente ejemplo de sistema emergente, dado que cada hormiga individual carece de la inteligencia suficiente para encontrar el camino más corto hasta una fuente de alimento, y sin embargo su comportamiento coordinado lo consigue.

*** En inglés, *ant colony optimization*.**

La forma en que las hormigas consiguen encontrar el camino más corto es la siguiente: en principio las hormigas vagabundean al azar alrededor de su hormiguero, y cuando encuentran alimento toman un poco y vuelven a su hormiguero dejando un rastro de feromonas. Si otras hormigas encuentran ese rastro, es probable que dejen de vagabundear al azar y lo sigan, ya que supuestamente conduce a una fuente de alimento. A su vez, las hormigas que vuelven con alimento dejan su propio rastro de feromona, reforzando así ese camino. Si hay varios caminos hacia una misma fuente de alimento el más corto acabará siendo el preferido por las hormigas, por la sencilla razón de que al ser más corto lo recorrerán más hormigas por unidad de tiempo, y por tanto la intensidad de su rastro de feromonas será mayor. Por otra parte, las feromonas se evaporan gradualmente, con lo que los caminos que no se utilizan van perdiendo atractivo. De esta forma, partiendo de una exploración aleatoria se consigue encontrar un camino óptimo o cercano al óptimo.

Sistemas emergentes

Los sistemas emergentes son aquellos que resuelven problemas complejos y exhiben un elevado grado de inteligencia combinando subsistemas sencillos y poco inteligentes. Las neuronas son quizá el ejemplo más claro.

5.6.1. Descripción del método

La adaptación de este comportamiento a un algoritmo de optimización es relativamente directa*: se genera una "hormiga" que en principio explora el espacio del problema de forma aleatoria, y cuando encuentra una solución al problema, la marca positivamente con "feromonas", de manera que las siguientes hormigas seguirán ese camino con una probabilidad más alta, si bien manteniendo un cierto componente aleatorio.

*** Fue propuesta por primera vez por M. Dorigo en su tesis doctoral: Optimization, Learning and Natural Algorithms, PhD thesis, Politecnico di Milano, Italia, 1992.**

La estructura general del algoritmo es muy sencilla: en cada iteración se genera una "hormiga", que propone una solución al problema. El criterio de finalización puede ser tan sencillo como alcanzar un número de iteraciones predeterminado, o puede ser más elaborado, como alcanzar una solución de cierta calidad o comprobar que sucesivas iteraciones no mejoran la solución obtenida.

Las soluciones parciales del problema se denominan **estados de la solución**. Para generar una solución completa hay que partir del estado inicial e ir moviéndose a un nuevo estado de la solución hasta completar la solución; la naturaleza específica de cada problema determina cuál es una solución completa en cada caso. En cada momento, la probabilidad P_{xy} de que una hormiga pase de un estado x a un estado y, depende de dos factores: la facilidad ϕ_{xy} para pasar del estado x al y, y el rastro de feromonas ρ_{xy} que en ese momento está asociado a esa transición. Si E es el conjunto de estados de la solución, entonces la probabilidad de elegir un nuevo estado viene determinada por

$$P_{xy} = \frac{(1+\rho_{xy}^{\alpha})\phi_{xy}^{\beta}}{\sum_{i\in E-\{x\}}(1+\rho_{xi}^{\alpha})\phi_{xi}^{\beta}} \tag{64}$$

donde el parámetro $\alpha \geq 0$ controla la influencia del rastro de feromonas ρ_{xy} y el parámetro $0 \leq \beta \leq 1$ controla la influencia de la facilidad de transición ϕ_{xy}. En otras palabras, la probabilidad de cambiar a un estado y se calcula comparando la facilidad y nivel de feromonas del camino xy con las del resto de caminos disponibles desde x.

Cada vez que una hormiga completa una solución es necesario actualizar el rastro de feromonas entre estados de la solución. Se aplican dos operaciones: por una parte, las feromonas tienden a evaporarse con el tiempo; por otra, la hormiga deja un rastro de feromonas por el camino que ha seguido. Así, la cantidad de feromona asociada a la transición entre los estados x e y viene dada por:

$$\rho_{xy} = (1-\gamma)\rho_{xy} + \Delta\rho_{xy} \tag{65}$$

donde $0 \leq \gamma \leq 1$ es el coeficiente de evaporación de las feromonas y $\Delta\rho_{xy}$ es el rastro dejado por la hormiga en la transición de x a y, típicamente cero cuando la hormiga no ha utilizado esa transición en su solución, y un valor constante cuando la transición sí que forma parte de la solución.

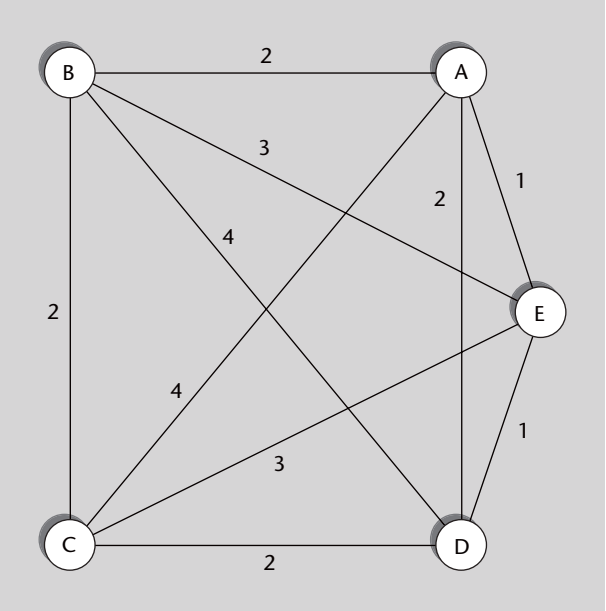
Una mejora habitual de este algoritmo consiste en permitir que las hormigas dejen un rastro de feromonas sólo cuando han encontrado un camino mejor que el mejor camino hasta ese momento, de manera que se premie el hallazgo de caminos mejores y se reduzca el número de iteraciones necesario para encontrarlos.

5.6.2. Ejemplo de aplicación

Las aplicaciones más evidentes de este algoritmo son aquellas en las que se busca el camino más corto entre dos puntos, si bien se han aplicado con éxito a tareas tan diversas como planificación, clasificación, partición de conjuntos, procesado de imágenes y plegado de proteínas.

Como ejemplo de aplicación de la optimización con colonias de hormigas resolveremos el problema del viajante de comercio descrito anteriormente. Podemos formalizar el problema representándolo como un grafo en el que cada ciudad es un nodo y los arcos están etiquetados con la distancia de una ciudad a otra. En este ejemplo vamos a suponer que la ciudad de origen es la *A*, y que el viajante debe regresar a ella al final de su trayecto. En la figura 55 podemos ver un ejemplo de este problema, en el que los valores de los arcos representan las distancias entre ciudades.

Figura 55. Problema ejemplo del viajante de comercio



Para resolver este problema mediante el algoritmo de la colonia de hormigas generaremos sucesivas hormigas. La primera de ellas recorrerá las ciudades en un orden aleatorio, si bien será más probable que viaje a las ciudades más cercanas, una restricción lógica si intentamos simular el comportamiento de una hormiga real. Así, la facilidad de ir de un nodo x a un nodo y será $\phi_{xy} = 1/dist(x,y)$. Una solución de este problema será una secuencia de ciudades en la que cada ciudad aparezca una y sólo una vez. La solución óptima será aquella que minimice la distancia total recorrida por la hormiga, incluyendo la vuelta a la ciudad de partida tras recorrer las restantes ciudades.

El criterio de finalización empleado en este ejemplo es muy sencillo: un número de iteraciones (hormigas) predeterminado, exactamente 3 por no alargar excesivamente el ejemplo. Tal y como se ha explicado anteriormente, es posible utilizar un criterio más sofisticado. El ejemplo de ejecución no pretende mostrar las ventajas del algoritmo, pues hacerlo requeriría resolver un problema con gran número de ciudades y de iteraciones, sino simplemente ilustrar el funcionamiento básico del algoritmo; su verdadera potencia se aprecia en problemas mayores.

Ejecución del algoritmo

La tabla 33 muestra el peso inicial de las transiciones entre los estados (ciudades) del problema propuesto en la figura 55. Se puede observar que, inicialmente, no hay rastros de feromonas; por tanto, la probabilidad de que una hormiga tome una transición determinada es inversamente proporcional a la distancia a esa ciudad, tal y como se ha propuesto antes. Nótese también que en este ejemplo no se distinguen dos sentidos en las transiciones entre una ciudad y otra, esa es la razón por la que sólo se muestra la mitad superior de la tabla.

Tabla 33. Estado inicial del problema del viajante

	B	C	D	E
A	$(1+0)\frac{1}{2} = 0{,}5$	$(1+0)\frac{1}{4} = 0{,}25$	$(1+0)\frac{1}{2} = 0{,}5$	$(1+0)1 = 1$
B		$(1+0)\frac{1}{2} = 0{,}5$	$(1+0)\frac{1}{4} = 0{,}25$	$(1+0)\frac{1}{3} = 0{,}33$
C			$(1+0)\frac{1}{2} = 0{,}5$	$(1+0)\frac{1}{3} = 0{,}33$
D				$(1+0)1 = 1$

Valor $(1+\rho_{xy})\phi_{xy}$ para ir de la ciudad x a la ciudad y

En la primera iteración del algoritmo, una primera hormiga recorre las ciudades en un orden aleatorio, si bien condicionado por las probabilidades de cada transición.

> Por tanto, esta primera hormiga ejecuta un algoritmo ávido-aleatorio, en el que elige el camino aleatoriamente, si bien las transiciones más atractivas en cada momento se eligen con mayor probabilidad, aunque a largo plazo se trate de malas decisiones.

La ruta seguida por la hormiga se marcará con feromonas, haciendo que las siguientes hormigas sigan esas transiciones con mayor probabilidad. En este caso la ruta seguida por la hormiga ha sido $A \to E \to D \to B \to C \to A$, con una longitud total de $1 + 1 + 4 + 2 + 4 = 12$.

Por otra parte, al terminar la primera iteración las feromonas sufren una cierta evaporación de acuerdo con la ecuación 65. En este ejemplo tomaremos $\gamma = 0{,}3$, un valor relativamente alto dado que en este ejemplo el número de iteraciones es muy reducido; por regla general conviene tomar un valor menor, por ejemplo 0,1. Toda esa información se muestra en la tabla 34.

Tabla 34. Problema del viajante tras la primera iteración

	B	C	D	E
A	$(1+0)\frac{1}{2} = 0{,}5$	$(1+0{,}7)\frac{1}{4} = 0{,}425$	$(1+0)\frac{1}{2} = 0{,}5$	$(1+0{,}7)1 = 1{,}7$
B		$(1+0{,}7)\frac{1}{2} = 0{,}85$	$(1+0{,}7)\frac{1}{4} = 0{,}425$	$(1+0)\frac{1}{3} = 0{,}33$
C			$(1+0)\frac{1}{2} = 0{,}5$	$(1+0)\frac{1}{3} = 0{,}33$
D				$(1+0{,}7)1 = 1{,}7$

De esta forma, la siguiente hormiga se encuentra unas probabilidades diferentes de tomar cada transición, guiada por el recorrido de la hormiga anterior. La segunda hormiga sigue el camino $A \to E \to C \to D \to B \to A$, con una longitud

de 1 + 3 + 2 + 4 + 2 = 12; al no mejorar la solución anterior, esta hormiga no deja feromonas, aunque las feromonas existentes sufren una cierta evaporación. En la tabla 35 se puede ver el estado de las transiciones tras la segunda iteración.

Tabla 35. Problema del viajante tras la segunda iteración

	B	C	D	E
A	$(1+0)\frac{1}{2} = 0{,}5$	$(1+0{,}49)\frac{1}{4} = 0{,}373$	$(1+0)\frac{1}{2} = 0{,}5$	$(1+0{,}49)1 = 1{,}49$
B		$(1+0{,}49)\frac{1}{2} = 0{,}745$	$(1+0{,}49)\frac{1}{4} = 0{,}373$	$(1+0)\frac{1}{3} = 0{,}33$
C			$(1+0)\frac{1}{2} = 0{,}5$	$(1+0)\frac{1}{3} = 0{,}33$
D				$(1+0{,}49)1 = 1{,}49$

La última hormiga sigue el camino $A \to E \to D \to C \to B \to A$, con una longitud de 1 + 1 + 2 + 2 + 2 = 8, más corto que el mejor camino encontrado hasta ahora; de hecho, es fácil comprobar que es el camino óptimo para este problema (como el camino inverso, obviamente). En este caso la hormiga sí que deja un rastro de feromonas, que sería utilizado por las hormigas siguientes si las hubiera. En la tabla 36 puede verse el estado final de las transiciones.

Tabla 36. Problema del viajante tras la tercera iteración

	B	C	D	E
A	$(1+0{,}7)\frac{1}{2} = 0{,}85$	$(1+0{,}49)\frac{1}{4} = 0{,}373$	$(1+0)\frac{1}{2} = 0{,}5$	$(1+1{,}043)1 = 2{,}043$
B		$(1+1{,}043)\frac{1}{2} = 1{,}022$	$(1+0{,}49)\frac{1}{4} = 0{,}373$	$(1+0)\frac{1}{3} = 0{,}33$
C			$(1+0{,}7)\frac{1}{2} = 0{,}85$	$(1+0)\frac{1}{3} = 0{,}33$
D				$(1+1{,}043)1 = 2{,}043$

5.6.3. Análisis del método

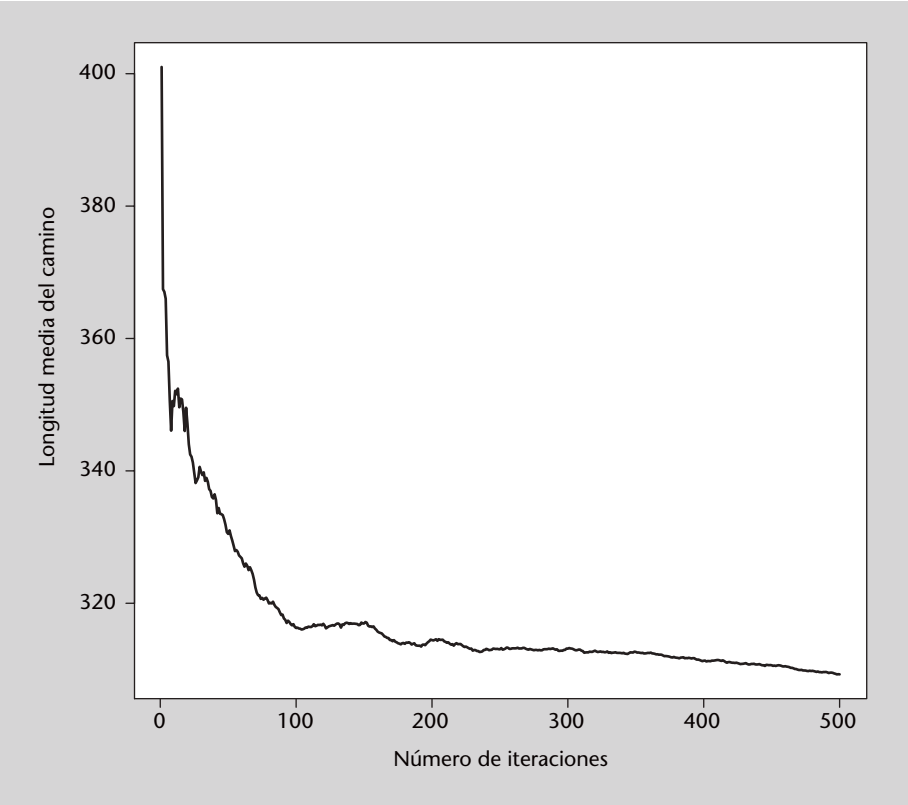
Este método de optimización tiene numerosas ventajas, pues permite encontrar soluciones razonablemente buenas a problemas muy complejos mejorando progresivamente soluciones aleatorias. Además por lo general evita los mínimos locales. Su aplicabilidad es bastante general, pues se puede aplicar a cualquier problema que pueda traducirse a la búsqueda de un camino óptimo en un grafo. Sin embargo, en algunos casos esa traducción puede ser difícil o muy costosa.

Quizá el principal problema de este algoritmo radique en la gran cantidad de parámetros que hay que elegir: pesos α y β en el cálculo de probabilidades, coeficiente de evaporación γ, etc.

Para ilustrar el efecto del número de iteraciones en la longitud del camino obtenido, en la figura 56 se muestra la evolución de la longitud media de los caminos seguidos por las hormigas frente al número de iteraciones (hormigas). En este ejemplo se han tomado 100 ciudades con una distancia máxima entre ellas de 10. Se puede observar que la longitud de los caminos baja rápidamente y se estabiliza a medida que se incrementan las iteraciones. Para valorar la eficiencia del algoritmo, hay que tener en cuenta que el número de recorridos

posibles en este problema es $n!$, siendo n el número de ciudades; así pues, para 100 ciudades el número de recorridos posibles es 100!, o sea más de 10^{157} recorridos posibles. Evidentemente, una exploración exhaustiva de todos ellos es impracticable.

Figura 56. Efecto del número de iteraciones en la longitud media del camino



5.6.4. Código fuente en Python

Código 5.4: algoritmo de la colonia de hormigas

```
1  # -*- coding: utf-8 -*-
2  import random, sys, math
3
4  # Nota: en lugar de matrices se usan listas de listas
5
6  # Genera una matriz de distancias de nCiudades x nCiudades
7  def matrizDistancias(nCiud, distanciaMaxima):
8      matriz = [[0 for i in range(nCiud)] for j in range(nCiud)]
9
10     for i in range(nCiud):
11         for j in range(i):
12             matriz[i][j] = distanciaMaxima*random.random()
13             matriz[j][i] = matriz[i][j]
14
15     return matriz
16
17 # Elige un paso de una hormiga, teniendo en cuenta las distancias
18 # y las feromonas y descartando las ciudades ya visitadas.
19 def eligeCiudad(dists, ferom, visitadas):
20     # Se calcula la tabla de pesos de cada ciudad
21     listaPesos  = []
22     disponibles = []
23     actual      = visitadas[-1]
24
25     # Influencia de cada valor (alfa: feromonas; beta: distancias)
```

```
    alfa = 1.0
    beta = 0.5

    # El parametro beta (peso de las distancias) es 0.5, alfa=1.0
    for i in range(len(dists)):
        if i not in visitadas:
            fer  = math.pow((1.0 + ferom[actual][i]), alfa)
            peso = math.pow(1.0/dists[actual][i], beta) * fer
            disponibles.append(i)
            listaPesos.append(peso)

    # Se elige aleatoriamente una de las ciudades disponibles,
    # teniendo en cuenta su peso relativo.
    valor      = random.random() * sum(listaPesos)
    acumulado = 0.0
    i          = -1
    while valor > acumulado:
        i          += 1
        acumulado += listaPesos[i]

    return disponibles[i]


# Genera una "hormiga", que elegira un camino teniendo en cuenta
# las distancias y los rastros de feromonas. Devuelve una tupla
# con el camino y su longitud.
def eligeCamino(distancias, feromonas):
    # La ciudad inicial siempre es la 0
    camino     = [0]
    longCamino = 0

    # Elegir cada paso segun la distancia y las feromonas
    while len(camino) < len(distancias):
        ciudad       = eligeCiudad(distancias, feromonas, camino)
        longCamino += distancias[camino[-1]][ciudad]
        camino.append(ciudad)

    # Para terminar hay que volver a la ciudad de origen (0)
    longCamino += distancias[camino[-1]][0]
    camino.append(0)

    return (camino, longCamino)

# Actualiza la matriz de feromonas siguiendo el camino recibido
def rastroFeromonas(feromonas, camino, dosis):
    for i in range(len(camino) - 1):
        feromonas[camino[i]][camino[i+1]] += dosis

# Evapora todas las feromonas multiplicandolas por una constante
# = 0.9 (en otras palabras, el coeficiente de evaporacion es 0.1)
def evaporaFeromonas(feromonas):
    for lista in feromonas:
        for i in range(len(lista)):
            lista[i] *= 0.9

# Resuelve el problema del viajante de comercio mediante el
# algoritmo de la colonia de hormigas. Recibe una matriz de
# distancias y devuelve una tupla con el mejor camino que ha
# obtenido (lista de indices) y su longitud
def hormigas(distancias, iteraciones, distMedia):
    # Primero se crea una matriz de feromonas vacia
    n = len(distancias)
    feromonas = [[0 for i in range(n)] for j in range(n)]

    # El mejor camino y su longitud (inicialmente "infinita")
    mejorCamino     = []
    longMejorCamino = sys.maxsize

    # En cada iteracion se genera una hormiga, que elige un camino,
    # y si es mejor que el mejor que teniamos, deja su rastro de
    # feromonas (mayor cuanto mas corto sea el camino)
    for iter in range(iteraciones):
```

```
98          (camino,longCamino) = eligeCamino(distancias, feromonas)
99
100         if longCamino <= longMejorCamino:
101             mejorCamino     = camino
102             longMejorCamino = longCamino
103
104         rastroFeromonas(feromonas, camino, distMedia/longCamino)
105
106         # En cualquier caso, las feromonas se van evaporando
107         evaporaFeromonas(feromonas)
108
109
110     # Se devuelve el mejor camino que se haya encontrado
111     return (mejorCamino, longMejorCamino)
112
113
114 # Generacion de una matriz de prueba
115 numCiudades     = 10
116 distanciaMaxima = 10
117 ciudades        = matrizDistancias(numCiudades, distanciaMaxima)
118
119 # Obtencion del mejor camino
120 iteraciones = 1000
121 distMedia   = numCiudades*distanciaMaxima/2
122 (camino, longCamino) = hormigas(ciudades, iteraciones, distMedia)
123 print("Camino: ", camino)
124 print("Longitud del camino: ", longCamino)
```

5.7. Optimización con enjambres de partículas

Supongamos que se nos plantea buscar el mínimo (o máximo) de una función matemática cualquiera, como por ejemplo

$$f(x,y) = x^2(4 - 2{,}1x^2 + \frac{x^4}{3}) + xy + y^2(-4 + 4y^2) \quad (66)$$

dentro de un intervalo determinado, en este caso [(–10, – 10),(10,10)]. Se trata de un ejemplo puramente ilustrativo, pues f es una función que se puede derivar fácilmente y por tanto, es posible encontrar los mínimos de forma analítica. Concretamente, hay dos mínimos globales, en (0,09, – 0,71) y en (–0,09,0,71), en los que la función toma el valor de –1,0316.

El método de optimización con enjambres de partículas* debe su nombre a que utiliza un conjunto de soluciones candidatas que se comportan como partículas moviéndose por un espacio: el espacio de soluciones. El movimiento de las partículas se determina por su velocidad, que depende de la posición del mejor punto que ha encontrado cada una de ellas en su recorrido y del mejor punto encontrado globalmente por el enjambre.

*** En inglés, *particle swarm optimization*.**

Vale la pena mencionar el origen de este método, pues curiosamente surgió a partir de un trabajo que simulaba el comportamiento de entidades sociales

como rebaños de animales. Analizando el comportamiento de esa simulación, y simplificando un poco las reglas que la gobernaban, se observó que las entidades tendían a buscar el punto mínimo del espacio en el que se encontraban.

Lectura complementaria

El método de optimización con enjambres de partículas se describió en J. Kennedy; R. Eberhart (1995)."Particle Swarm Optimization". IEEE International Conference on Neural Networks

5.7.1. Descripción del método

Suponed que queremos encontrar el punto mínimo de una función $f : \mathbb{R}^n \to \mathbb{R}$, donde el espacio de soluciones $S \subset \mathbb{R}^n$ está acotado por los puntos $inf, sup \in \mathbb{R}^n$. El intervalo del espacio de soluciones S no tiene por qué ser el mismo en todas las dimensiones.

Primer paso

El primer paso del método de optimización con enjambres de partículas consiste en crear un conjunto de partículas (el enjambre) al que llamaremos E. A cada partícula $p_i \in E$ se le asigna una **posición inicial** x_i en S según la fórmula:

$$x_i \leftarrow U(inf, sup) \tag{67}$$

Donde $U : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^n$ es una función que devuelve un número aleatorio con distribución uniforme dentro del subespacio de $\mathbb{R}^n$ delimitado por los dos puntos que recibe como parámetro.

Además cada partícula tiene una **velocidad inicial** v_i que viene dada por la fórmula:

$$v_i \propto U(-|sup - inf|, |sup - inf|) \tag{68}$$

Es decir, la velocidad inicial en cada dimensión es proporcional al intervalo del espacio de soluciones en esa dimensión. Habitualmente, la velocidad inicial se multiplica por un factor $0 < k < 1$ para evitar que una velocidad inicial excesiva haga que las partículas se salgan de S en pocas iteraciones.

Por último, cada partícula recuerda el **mejor punto** m_i que ha visitado; inicialmente $m_i = x_i$, pues es el único punto que ha visitado cada partícula. También es necesario almacenar el mejor punto visitado por el enjambre, g, es decir el **mínimo global** obtenido hasta un momento dado:

$$g \leftarrow \underset{m_i}{\operatorname{argmin}} f(m_i) \tag{69}$$

Iteraciones del algoritmo

Como es habitual en este tipo de métodos, no hay un número predefinido de iteraciones; el usuario del método deberá elegir. Las opciones más habituales son: número fijo de iteraciones, repetir hasta alcanzar un valor de f suficientemente bueno, hasta que no se observen mejoras en g o hasta que transcurra un tiempo determinado.

En cada iteración las partículas cambian su velocidad influidas por su mejor punto m_i y por el mejor punto global en ese momento, g. En primer lugar se calcula la nueva velocidad de cada partícula del enjambre $i = 1,...,|E|$ según la expresión:

$$v_i \leftarrow \alpha v_i + \beta_c a_c (m_i - x_i) + \beta_s a_s (g - x_i) \tag{70}$$

Los parámetros y valores que aparecen en la expresión anterior son:

- α es un coeficiente que regula la **inercia** de la partícula, esto es, su tendencia a seguir con su trayectoria e ignorar la influencia de otros elementos.
- β_c (c por **cognitiva**) regula la influencia del aspecto cognitivo de la propia partícula, es decir, la influencia del mejor punto que ella recuerda en el cálculo de su nueva velocidad.
- β_s (s por **social**) regula la influencia del aspecto social sobre la partícula, es decir, la influencia del mejor punto encontrado por el enjambre. Cuanto mayor sea, más tenderán las partículas a moverse hacia el mejor punto, g, y por tanto la influencia de lo social, del enjambre, será mayor.
- a_c y a_s son dos números aleatorios calculados mediante $U(0,1)$ que asignan una influencia aleatoria a los aspectos cognitivos y sociales respectivamente.

La nueva posición de cada partícula depende de su posición actual y de su nueva velocidad:

$$x_i \leftarrow x_i + v_i \tag{71}$$

Si la nueva posición de la partícula es una solución mejor que la mejor solución que recordaba, es decir, si $f(x_i) < f(m_i)$ entonces x_i pasa a ser la nueva mejor posición de esa partícula: $m_i \leftarrow x_i$.

Tras actualizar todas las partículas, se comparan sus mejores posiciones m_i con la mejor posición global g; si alguna $m_i < g$, entonces $g \leftarrow m_i$.

Al acabar las iteraciones, g contiene la mejor solución encontrada por el algoritmo.

Selección de los parámetros α, β_c y β_s

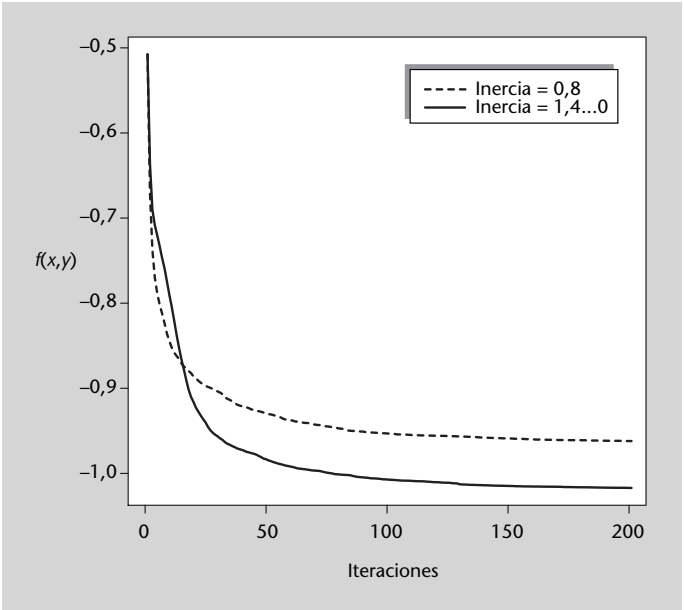
En la expresión 70 aparecen tres parámetros que se utilizan para calcular la nueva velocidad de una partícula. Estos parámetros permiten ajustar la influencia de la inercia de la partícula (α), del mejor punto que ella conoce (β_c) y del mejor punto encontrado por el enjambre (β_s). Ahora bien, ¿qué valor deben tomar? Diferentes trabajos empíricos* recomiendan que tanto β_c como β_s tengan un valor de 2,0, mientras que la inercia debe influir menos ($\alpha = 0{,}8$).

***Destacamos Y. Shi; R. C. Eberhart (1998). "Parameter selection in particle swarm optimization". Proceedings of Evolutionary Programming VII.**

Una mejora del método consiste en empezar con una inercia relativamente alta ($\alpha = 1{,}4$) que se va reduciendo en cada iteración, por ejemplo, multiplicándola por un factor $r < 1$. El sentido de esta mejora es el siguiente. En las primeras iteraciones las partículas se mueven a mayor velocidad debido a su elevada inercia; con ello se consigue explorar más exhaustivamente el espacio de soluciones *S* y evitar así quedar atrapado en mínimos locales. Sin embargo, a medida que la ejecución del algoritmo avanza, interesa pulir las soluciones encontradas, llevando a cabo una exploración más fina de su entorno. Para conseguirlo las partículas se deben mover lentamente y explorar los alrededores de los mejores puntos encontrados, lo que se consigue haciendo que su inercia sea menor.

Esta mejora es bastante habitual y suele mejorar los resultados de los algoritmos de optimización con enjambres de partículas. En la figura 57 se compara la ejecución del algoritmo básico (inercia constante) y del algoritmo mejorado (inercia decreciente). En este caso, se muestra el valor medio de $f(x,y)$ de 10 partículas durante 200 iteraciones. Se ha elegido un número relativamente pequeño de partículas para que la obtención de puntos mínimos no se deba al azar inicial.

Figura 57. Comparación de las estrategias de inercia constante e inercia decreciente



Como se puede observar, con inercia constante el algoritmo empieza mejorando antes los resultados (minimizando $f(x,y)$), si bien llega un punto en el que se estanca porque no es capaz de hacer una exploración más fina para seguir mejorando la posición de las partículas. Por su parte, con inercia decreciente el algoritmo empieza a optimizar más lentamente que con inercia constante, pero a medida que la ejecución avanza sigue mejorando los resultados ya que las partículas, ahora más lentas, pueden explorar con mayor precisión el área alrededor del mínimo global. Por esta razón, el código de ejemplo del subapartado 5.7.4 hace uso de inercia decreciente.

El valor del factor r depende del número de iteraciones previstas, y conviene ajustarlo de manera que hacia las últimas iteraciones la inercia tienda a cero, pero no antes. Por ejemplo, con 100 iteraciones se tiene que $0{,}9^{100} = 0{,}0000265$, luego $r = 0{,}9$ es adecuado si el número de iteraciones es 100, pero no si es por ejemplo 1000, ya que en ese caso a partir de la iteración 100 la inercia queda prácticamente desestimada. En ese caso sería mejor tomar $r = 0{,}99$, por ejemplo, ya que en la iteración 100 la inercia sigue teniendo un peso razonable, $0{,}99^{100} = 0{,}366$.

5.7.2. Ejemplo de aplicación

A modo de ejemplo se va a utilizar la optimización con enjambres de partículas para buscar el mínimo de la función 66. Se trata de un problema muy sencillo, con un espacio de soluciones muy "pequeño" (sólo dos dimensiones y unos intervalos bastante reducidos), por lo que una simple exploración al azar puede dar resultados aceptables. Sin embargo, se ha elegido así expresamente, pues permite mostrar la traza de ejecución completa, en este caso con cuatro partículas y tres iteraciones.

En la tabla 37 se puede ver, para cada iteración, cómo las partículas se van moviendo hacia la solución y van mejorando sus posiciones. Así, con tan solo cuatro partículas y tres iteraciones, el mínimo obtenido mejora claramente, pasando de –0,0372 a –0,927, bastante cercano al mínimo global que ya conocemos.

5.7.3. Análisis del método

El método de optimización con enjambres de partículas es un método adecuado a gran cantidad de problemas. Si bien el ejemplo planteado en el subapartado 5.7.2 permite comprobar paso a paso la ejecución del algoritmo en un caso sencillo, la potencia de este método se hace patente en problemas con mayor número de dimensiones, en los que la búsqueda al azar resulta infructuosa. El método ofrece las siguientes ventajas:

Tabla 37. Ejecución del enjambre de partículas

Partícula	Posición	Velocidad	$f(x,y)$	Mejor posición	Mejor $f(x,y)$
			Enjambre inicial		
1	(-1,463, 0,695)	(-0,236, -0,745)	0,195	(-1,463, 0,695)	0,195
2	(-0,0183, -0,101)	(-0,348, -0,211)	-0,0372	(-0,0183, -0,101)	-0,0372
3	(-1,625, -0,943)	(-0,164, -0,567)	3,197	(-1,625, -0,943)	3,197
4	(1,0491, -0,996)	(-0,555, -0,278)	1,225	(1,0491, -0,996)	1,225
g	(-0,0183, -0,101)		-0,0372		
			Iteración 1		
1	(1,079, 0,182)	(2,542, -0,513)	2,405	(-1,463, 0,695)	0,195
2	(-0,297, -0,333)	(-0,279, -0,232)	-0,101	(-0,297, -0,333)	**-0,101**
3	(-0,400, -0,722)	(1,225, 0,222)	-0,123	(-0,400, -0,722)	**-0,123**
4	(-0,453, -1)	(-1,502, -0,754)	1,188	(-0,453, -1)	**1,188**
g	(-0,400, -0,722)		**-0,123**		
			Iteración 2		
1	(0,641, -0,00817)	(-0,438, -0,190)	1,306	(-1,463, 0,695)	0,195
2	(-0,168, -0,379)	(0,129, -0,0454)	-0,317	(-0,168, -0,379)	**-0,317**
3	(0,580, -0,758)	(0,980, -0,0367)	-0,297	(0,580, -0,758)	**-0,297**
4	(-1,579, -1)	(-1,126, -0,830)	3,664	(-0,453, -1)	1,188
g	(-0,168, -0,379)		**-0,317**		
			Iteración 3		
1	(-2, -0,932)	(-4,926, -0,924)	5,139	(-1,463, 0,695)	0,195
2	(-0,0640, -0,663)	(0,104, -0,285)	-0,927	(-0,0640, -0,663)	**-0,927**
3	(1,000847, -1)	(0,421, -0,824)	1,234	(0,580, -0,758)	-0,297
4	(-0,541, -0,0443)	(1,0381, 0,956)	1,0153	(-0,541, -0,0443)	**1,0153**
g	(-0,0640, -0,663)		**-0,927**		

Las coordenadas expresan (x,y). g es la mejor posición global.

- Número relativamente reducido de parámetros. En muchos casos es suficiente con elegir un número de partículas, pues el método suele funcionar bien asignando a los parámetros los valores $\alpha = 1{,}4..,0$, $\beta_c = 2{,}0$ y $\beta_s = 2{,}0$, tal y como se ha explicado más arriba.
- Adecuado para espacios de soluciones continuos.
- Al haber muchas partículas no se queda atascado en mínimos locales.

Por otra parte hay que tener en cuenta algunas limitaciones o inconvenientes del método:

- Puede no ser aplicable o resultar ineficiente en espacios de soluciones discretos.
- Si el coste computacional del la función objetivo es alto, este método incurrirá en elevados tiempos de ejecución, ya que al haber muchas partículas es necesario evaluar la función objetivo muchas veces.

Para ilustrar el funcionamiento del método la figura 58 muestra el recorrido de una partícula. Nótese cómo los primeros cambios de posición son muy grandes, y a medida que se suceden las iteraciones la partícula se mueve con saltos cada vez más pequeños alrededor del punto mínimo que está buscando. Este comportamiento está potenciado por el uso de inercia decreciente, si bien también ocurre —aunque en menor grado— cuando se usa inercia constante debido a la mayor cercanía de la partícula a su propio mínimo local y al mínimo global del enjambre.

Figura 58. Recorrido de una partícula

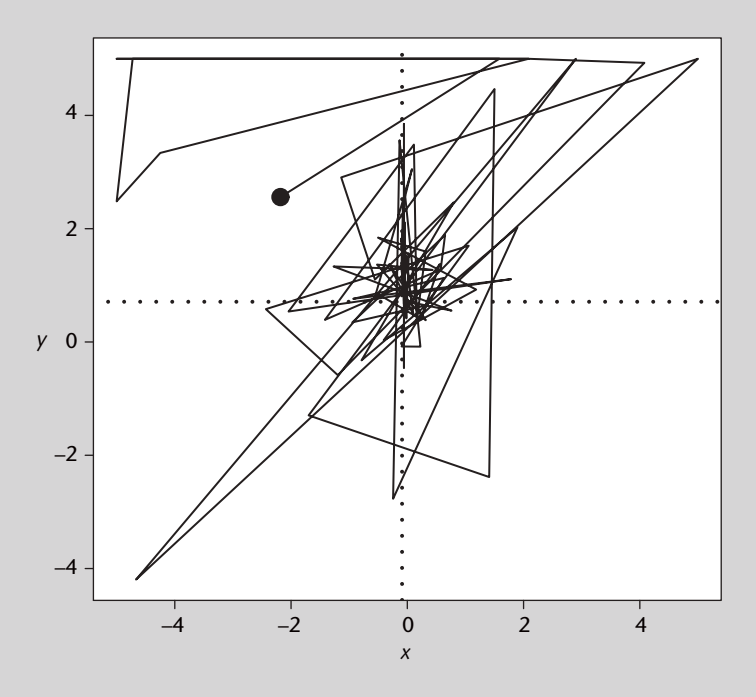


Figura 58

Recorrido de una partícula desde el punto inicial (señalado por el punto negro) hacia el punto óptimo (situado en la intersección de las líneas de puntos). Se muestra el recorrido de la mejor partícula de una simulación con 4 partículas y 100 iteraciones.

5.7.4. Código fuente en Python

Código 5.5: algoritmo del enjambre de partículas

```
1  # -*- coding: utf-8 -*-
2  from random import random
3
4  # Funcion que se quiere minimizar
5  def funcion(x, y):
6      sum1 = x**2 * (4-2.1*x**2 + x**4/3.0)
7      sum2 = x*y
8      sum3 = y**2 * (-4+4*y**2)
9      return sum1 + sum2 + sum3
10
11
12 # Devuelve un numero aleatorio dentro de un rango con
13 # distribucion uniforme (proporcionada por random)
14 def aleatorio(inf, sup):
15     return random()*(sup-inf) + inf
16
17
18 # Clase que representa una particula individual y que facilita
19 # las operaciones necesarias
20 class Particula:
```

```
21      # Algunos atributos de clase (comunes a todas las particulas)
22      # Parametros para actualizar la velocidad
23      inercia     = 1.4
24      cognitiva   = 2.0
25      social      = 2.0
26      # Limites del espacio de soluciones
27      infx = -2.0
28      supx =  2.0
29      infy = -1.0
30      supy =  1.0
31      # Factor de ajuste de la velocidad inicial
32      ajusteV = 100.0
33
34      # Crea una particula dentro de los limites indicados
35      def __init__(self):
36          self.x  = aleatorio(Particula.infx, Particula.supx)
37          self.y  = aleatorio(Particula.infy, Particula.supy)
38          self.vx = aleatorio(Particula.infx/Particula.ajusteV,
39                              Particula.supx/Particula.ajusteV)
40          self.vy = aleatorio(Particula.infy/Particula.ajusteV,
41                              Particula.supy/Particula.ajusteV)
42          self.xLoc     = self.x
43          self.yLoc     = self.y
44          self.valorLoc = funcion(self.x, self.y)
45
46      # Actualiza la velocidad de la particula
47      def actualizaVelocidad(self, xGlob, yGlob):
48          cogX    = Particula.cognitiva*random()*(self.xLoc-self.x)
49          socX    = Particula.social*random()*(xGlob-self.x)
50          self.vx = Particula.inercia*self.vx + cogX + socX
51          cogY    = Particula.cognitiva*random()*(self.yLoc-self.y)
52          socY    = Particula.social*random()*(yGlob-self.x)
53          self.vy = Particula.inercia*self.vy + cogY + socY
54
55      # Actualiza la posicion de la particula
56      def actualizaPosicion(self):
57          self.x = self.x + self.vx
58          self.y = self.y + self.vy
59
60          # Debe mantenerse dentro del espacio de soluciones
61          self.x = max(self.x, Particula.infx)
62          self.x = min(self.x, Particula.supx)
63          self.y = max(self.y, Particula.infy)
64          self.y = min(self.y, Particula.supy)
65
66          # Si es inferior a la mejor, la adopta como mejor
67          valor = funcion(self.x, self.y)
68          if valor < self.valorLoc:
69              self.xLoc     = self.x
70              self.yLoc     = self.y
71              self.valorLoc = valor
72
73
74  # Mueve un enjambre de particulas durante las iteraciones indicadas.
75  # Devuelve las coordenadas y el valor del minimo obtenido.
76  def enjambreParticulas(particulas, iteraciones, reduccionInercia):
77
78      # Registra la mejor posicion global y su valor
79      mejorParticula = min(particulas, key=lambda p:p.valorLoc)
80      xGlob     = mejorParticula.xLoc
81      yGlob     = mejorParticula.yLoc
82      valorGlob = mejorParticula.valorLoc
83
84      # Bucle principal de simulacion
85      for iter in range(iteraciones):
86          # Actualiza la velocidad y posicion de cada particula
87          for p in particulas:
88              p.actualizaVelocidad(xGlob, yGlob)
89              p.actualizaPosicion()
90
91          # Hasta que no se han movido todas las particulas no se
92          # actualiza el minimo global, para simular que todas se
```

```
93          # mueven a la vez
94          mejorParticula = min(particulas, key=lambda p:p.valorLoc)
95          if mejorParticula.valorLoc < valorGlob:
96              xGlob     = mejorParticula.xLoc
97              yGlob     = mejorParticula.yLoc
98              valorGlob = mejorParticula.valorLoc
99
100         # Finalmente se reduce la inercia de las particulas
101         Particula.inercia*=reduccionInercia
102
103     return (xGlob, yGlob, valorGlob)
104
105
106 # Parametros del problema
107 nParticulas = 10
108 iteraciones = 100
109 redInercia  = 0.9
110
111 # Genera un conjunto inicial de particulas
112 particulas=[Particula() for i in range(nParticulas)]
113
114 # Ejecuta el algoritmo del enjambre de particulas
115 print(enjambreParticulas(particulas, iteraciones, redInercia))
```

5.8. Búsqueda tabú

Una editorial está elaborando un atlas del mundo y desea colorear un mapa político utilizando un número determinado de colores y evitando que países contiguos tengan el mismo color, lo que dificultaría la lectura del mapa. ¿Cómo asignar un color a cada país?

El método de la **búsqueda tabú** es una mejora de la exploración aleatoria del espacio de soluciones S, cuya ventaja radica en que se recuerdan las soluciones probadas con anterioridad. De esa forma, no se vuelven a explorar regiones que ya se han visitado previamente, y así la eficiencia de búsqueda mejora. Su nombre se debe a que las soluciones ya analizadas se convierten en tabú, ya que se impide al algoritmo acceder a ellas.

5.8.1. Descripción del método

Supongamos que queremos buscar el mínimo de una función $f : S \to \mathbb{R}$, donde S es el espacio de soluciones posibles. Expresándolo de otra manera, buscamos el punto $x^* \in S$ que cumple $x^* = \text{argmin}\, f(x) \forall x \in S$, o bien una aproximación aceptable.

Búsqueda aleatoria global

La búsqueda aleatoria global es diferente de la búsqueda aleatoria local, pues consiste en ir probando puntos de S al azar (sin ninguna relación entre ellos) hasta que se cumpla la condición de finalización.

Antes de describir el método de la búsqueda tabú nos detendremos brevemente en el método de **búsqueda aleatoria local**, pues ambas están estrechamente relacionadas.

Cada punto $x \in S$ tiene un **vecindario** $V(x) \subset S$, que es el conjunto de puntos que están separados de x por un único cambio. La definición de V depende

del problema concreto de que se trate, pero en general dos puntos son vecinos si sólo se diferencian en una de sus variables (posiblemente en una cantidad acotada si se trata de variables numéricas).

El método de la búsqueda aleatoria local empieza evaluando un punto al azar $x_0 \in S$. A continuación toma un punto vecino al azar $x_1 \in V(x_0)$, lo evalúa y toma un vecino de x_1. El proceso se repite hasta que se cumple la condición de finalización que se haya establecido.

Es evidente que el método de búsqueda aleatoria volverá a menudo sobre puntos explorados previamente, con la consiguiente pérdida de tiempo. Aquí entra la mejora de la búsqueda tabú: se almacenan los últimos puntos explorados, que quedan marcados como tabú, de forma que el método los evitará y se verá forzado a explorar áreas nuevas. Visto de otra forma, se define una nueva función de vecindario $V'(x,i)$ que depende de la iteración i del algoritmo, pues excluye los vecinos que se han visitado con anterioridad. En cada iteración se elige al azar un punto $x_{i+1} \in V'(x_i,i)$, se evalúa y se marca como tabú para evitar visitarlo en futuras iteraciones.

Dado que los espacios de soluciones suelen tener un número elevado de dimensiones, la probabilidad de que un punto vuelva a ser visitado es muy pequeña. Por esa razón, marcar puntos de S como tabú tendría un efecto prácticamente nulo respecto a la búsqueda aleatoria local. Para que la búsqueda tabú tenga un efecto práctico las marcas tabú no se aplican a los puntos completos, sino a cada una de las componentes de los puntos. Así, si por ejemplo el espacio de soluciones es $\mathbb{Z}^3$, las marcas tabú se aplican por separado a los valores que toman x, y y z. De esa forma el efecto tabú es más potente y la mejora respecto a la búsqueda aleatoria local es apreciable.

Con la anterior estrategia se corre el riesgo de bloquear muchos puntos, lo que puede provocar que se pierdan soluciones valiosas. La forma de evitarlo es relajar la calidad de tabú de los valores, haciendo que las marcas tabú duren sólo unas iteraciones; seguidamente, los valores vuelven a estar disponibles. En el subapartado 5.8.3 se estudia la influencia de la duración de las marcas tabú en la eficiencia del método.

Dado que se trata de un método que parte de un punto al azar y explora el espacio de soluciones moviéndose entre puntos vecinos, el método de búsqueda tabú se clasifica como método de búsqueda local.

5.8.2. Ejemplo de aplicación

El ejemplo planteado inicialmente, consistente en colorear los países de un mapa sin que dos países del mismo color se toquen, es un caso particular del

problema más general del **coloreado de grafos**. Como es habitual, el mapa que queremos colorear (figura 59) puede verse como un grafo en el que cada país se representa mediante un nodo y los nodos correspondientes a países linderos están unidos mediante un arco.

Figura 59. Problema del coloreado de un mapa

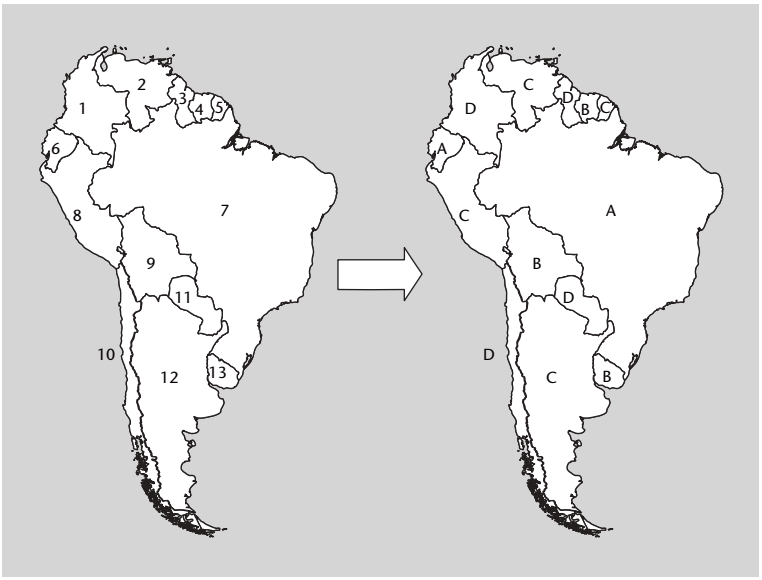


Figura 59

Ejemplo de coloreado de un mapa: tenemos un mapa con trece países (mapa izquierdo) y queremos colorearlo con cuatro colores (A, B, C y D). En el mapa derecho se muestra un posible coloreado del mapa obtenido mediante búsqueda tabú.

El problema genérico del coloreado de grafos tiene muchas más aplicaciones prácticas, como la asignación de registros del procesador al compilar, o la gestión de recursos excluyentes como pistas de aeropuertos, líneas de producción, etc. Al tratarse de un problema NP-completo, resulta interesante para probar métodos heurísticos, dado que el espacio de soluciones S tiene un tamaño $|S| = n^k$, siendo n el número de nodos y k el número de colores que se quiere utilizar; por ese motivo, una exploración exhaustiva es impracticable salvo para grafos muy pequeños.

Aunque la función objetivo más evidente para resolver este problema es el número de colores empleado en el mapa, esa función no cumple las propiedades enunciadas en el subapartado 5.1; en concreto, su principal problema es que su resolución es muy gruesa y por tanto, es incapaz de medir pequeñas mejoras en el coloreado, con lo que los algoritmos de búsqueda no reciben ninguna realimentación positiva respecto a las mejoras que van encontrando. Es necesario utilizar una función objetivo de mayor resolución, y en este caso la candidata idónea es la función que cuenta el número de colisiones, esto es de países linderos que tienen el mismo color. Como es evidente, esta función permite soluciones ilegales, pero a cambio proporciona una orientación adecuada para la búsqueda de su mínimo.

Las marcas tabú se aplican a cada una de las componentes de la solución. En este caso una solución es una correspondencia de un color para cada país, por lo que las marcas tabú serán colores prohibidos para un país determinado, debidas a que recientemente se le ha asignado ese color.

Si se deseara obtener el número mínimo de colores con los que se puede colorear un mapa dado, no habría más que iterar el algoritmo de búsqueda con un número de colores decreciente, hasta que el algoritmo sea incapaz de encontrar una asignación de colores válida (sin países linderos del mismo color).

5.8.3. Análisis del método

La búsqueda tabú es un método conceptualmente sencillo y relativamente eficiente de mejorar la búsqueda aleatoria local. No obstante, se trata de un método de búsqueda local, con las limitaciones que ello conlleva; a menudo conviene ejecutarlo repetidas veces para hacerlo más global y permitirle encontrar el mínimo global.

Como todo método que utiliza una función de vecindario, está mejor adaptado a espacios de soluciones discretos; en otro caso, es necesario discretizar el espacio o definir algún tipo de distancia de vecindad.

No hay un valor óptimo para la duración de las marcas tabú, sino que depende del número de dimensiones del espacio de soluciones y del número de valores posible de cada componente. En la figura 60 se estudia la influencia de la duración de las marcas tabú en la eficiencia del método en el ejemplo del coloreado del mapa. Una duración igual a cero equivale a la búsqueda aleatoria local, que es claramente peor que la búsqueda tabú. Una duración muy corta (<10) provoca que el tabú tenga poco efecto. Por otra parte, una duración muy larga vuelve el método muy rígido, pues excluye numerosas soluciones, y dificulta la exploración. Por tanto, tal y como se ve en la figura, es mejor elegir un número intermedio de iteraciones, en este ejemplo concreto 15.

Figura 60. Influencia de la duración del tabú

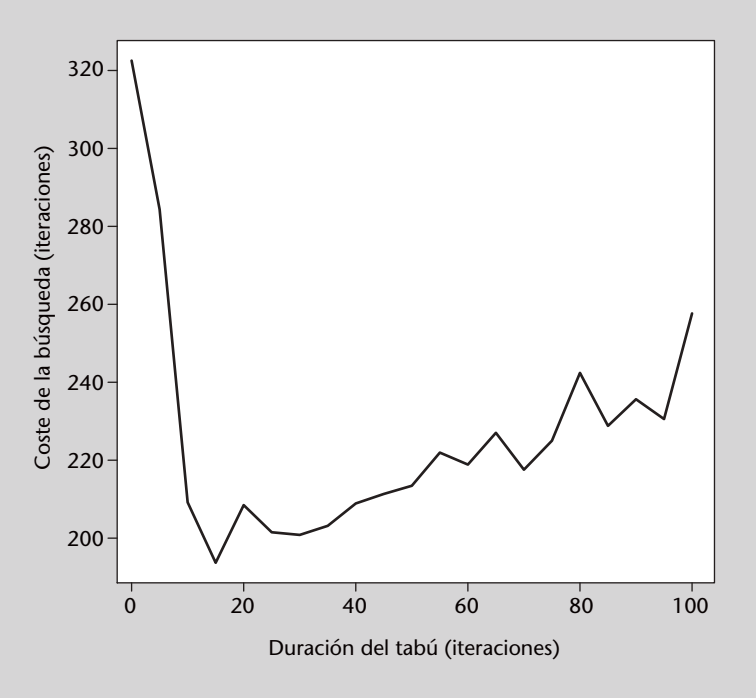


Figura 60

La duración del tabú influye claramente en la eficiencia de los algoritmos de búsqueda tabú. En la figura se muestra el número de iteraciones necesario para obtener la solución en un mapa con 200 países, una probabilidad de contacto entre ellos de 0,04 y cinco colores disponibles. Se muestran los valores medios de la búsqueda en 100 mapas diferentes.

5.8.4. Código fuente en Python

Código 5.6: coloreado de mapas con búsqueda tabú

```
# -*- coding: utf-8 -*-
# Nota: los grafos se representan como listas de listas
from random import randint, random, sample

# Dados dos nodos, devuelve 1 si estan conectados, 0 si no
def conectados(grafo, nodo1, nodo2):
    if nodo1 < nodo2:
        nodo1, nodo2 = nodo2, nodo1
    if nodo1 == nodo2:
        return 0
    else:
        return grafo[nodo1][nodo2]


# Funcion objetivo, que cuenta los vertices conectados que tienen
# el mismo color
def objetivo(grafo, estado):
    suma = 0
    for i in range(len(estado)-1):
        for j in range(i+1,len(estado)):
            if estado[i] == estado[j]:
                suma += conectados(grafo, i, j)
    return suma

# Devuelve una lista con los vertices que tienen algun problema
# (estan conectados a otro del mismo color)
def verticesProblema(grafo, estado):
    nVertices = len(estado)
    vertices  = []
    for i in range(nVertices):
        encontrado = False
        j = 0
        while j<nVertices and not encontrado:
            if (estado[i]==estado[j]) and conectados(grafo, i, j):
                encontrado = True
            j += 1
        if encontrado:
            vertices.append(i)
    return vertices

# Genera la lista de vecinos de un estado y devuelve el
# mejor de ellos, teniendo en cuenta la informacion tabu y
# actualizandola para anotar el cambio de estado producido.
def vecino(grafo, estado, k, tabu, iterTabu):
    # En primer lugar se obtienen todos los vertices
    # conectados a algun otro vertice del mismo color
    vertices = verticesProblema(grafo, estado)

    # Para cada uno de esos pares, se propone una
    # nueva solucion consistente en cambiar el color de uno de
    # los vertices (comprobando que no tome un color tabu).
    candidatos = []
    for v in vertices:
        # Si quedan colores disponibles para ese vertice
        if len(tabu[v]) < k:
            nuevoColor = randint(1,k)
            while nuevoColor in tabu[v]:
                nuevoColor = randint(1,k)
            nuevoEstado    = estado[:]
            nuevoEstado[v] = nuevoColor
            candidatos.append(nuevoEstado)

    # Se devuelve el mejor estado obtenido (funcion objetivo menor).
    # Si no hay soluciones posibles (los vertices problematicos
    # tienen todos los colores en tabu) se devuelve un cambio al
    # azar para desbloquear la situacion.
    if len(candidatos) > 0:
        elegido  = min(candidatos, key=lambda e:objetivo(grafo,e))
```

```
69          posicion = candidatos.index(elegido)
70          cambiado = vertices[posicion]
71      else:
72          elegido  = estado[:]
73          cambiado = sample(vertices, 1)[0]
74          elegido[cambiado] = randint(1, k)
75
76      # The change is added to the tabu list.
77      tabu[cambiado][estado[cambiado]] = iterTabu
78      return elegido
79
80
81  # Actualiza la informacion tabu, descontando una iteracion de
82  # todas las entradas y eliminando las que llegan a 0
83  def actualizaTabu(tabu):
84      for tabuVertice in tabu:
85          for color in tabuVertice.keys():
86              tabuVertice[color] -= 1
87          claves = [c for c,v in tabuVertice.items() if v<=0]
88          for c in claves:
89              del tabuVertice[c]
90
91
92  # Dados un grafo, una asignacion inicial y un numero de colores,
93  # busca una asignacion optima utilizando busqueda tabu.
94  # Tambien hay que indicar el numero maximo de iteraciones, asi
95  # como el numero de iteraciones que se mantiene un color en tabu.
96  def busquedaTabu(grafo, asignacion, k, iteraciones, iterTabu):
97
98      estado    = asignacion[:]
99      mejor     = estado[:]
100     nVertices = len(estado)
101
102     # La estructura "tabu" almacena, para cada vertice (posiciones
103     # en la lista), los colores  que tiene prohibidos (claves de
104     # cada diccionario) y durante cuantas iteraciones lo estaran
105     # (valores de cada diccionario)
106     tabu = [{} for n in range(nVertices)]
107     i = 0
108     while i < iteraciones and objetivo(grafo, estado) > 0:
109
110         # Selecciona el mejor vecino del estado actual
111         estado = vecino(grafo, estado, k, tabu, iterTabu)
112
113         if objetivo(grafo,estado)<objetivo(grafo,mejor):
114             mejor = estado[:]
115         print(i,objetivo(grafo,estado), objetivo(grafo,mejor))
116
117         # Actualiza los estados tabu (descuenta una iteracion)
118         actualizaTabu(tabu)
119
120         # Avanza a la siguiente iteracion
121         i += 1
122
123     return mejor
124
125
126 # Genera un grafo en forma de matriz de conectividad, con un numero
127 # de vertices y una probabilidad de contacto entre dos vertices.
128 # Devuelve la mitad inferior de la matriz como lista de listas.
129 def generaGrafo(nVertices, probContacto):
130     resultado = []
131     for i in range(nVertices):
132         # 1 si el numero aleatorio es menor o igual que
133         # la probabilidad de contacto, 0 si no
134         f = lambda : random()<=probContacto and 1 or 0
135         resultado.append([f() for j in range(i)])
136
137     return resultado
138
139
140 # Programa principal
```

```
141
142 # Ejemplo: grafo que representa Sudamerica (no se resuelve)
143 sudamerica = [[],
144               [1],
145               [0,1],
146               [0,0,1],
147               [0,0,0,1],
148               [1,0,0,0,0],
149               [1,1,1,1,1,0],
150               [1,0,0,0,0,1,1],
151               [0,0,0,0,0,0,1,1],
152               [0,0,0,0,0,0,0,1,1],
153               [0,0,0,0,0,0,1,0,1,0],
154               [0,0,0,0,0,0,1,0,1,1,1],
155               [0,0,0,0,0,0,1,0,0,0,0,1]]
156
157 # Generacion de un grafo aleatorio.
158 # Parametros del problema: numero de paises y probabilidad
159 # de que dos paises cualesquiera sean linderos.
160 # Logicamente, cuantos mas paises haya, mayor numero de
161 # contactos tendra un pais determinado.
162 nPaises      = 13
163 probContacto = 0.05
164 mapa         = generaGrafo(nPaises,probContacto)
165
166 # Ejecucion del algoritmo: numero maximo de iteraciones del
167 # algoritmo y numero que iteraciones que permanece un color
168 # como tabu para un pais dado.
169 iteraciones = 200
170 iterTabu    = 15
171
172 # Numero de colores
173 k=4
174
175 # Se crea un estado inicial aleatoriamente
176 estado = [randint(1,k) for i in range(nPaises)]
177
178 # Se ejecuta la busqueda tabu con k colores.
179 solucion = busquedaTabu(sudamerica, estado, k, iteraciones, iterTabu)
180 print('Solucion:' + str(solucion))
181 print('Colisiones=' + str(objetivo(sudamerica, solucion)))
```

6. Aprendizaje profundo

6.1. Introducción

En los últimos años, la inteligencia artificial ha ganado un gran protagonismo en los medios de información generalistas gracias a una serie de impresionantes logros en diferentes áreas. La mayor parte de estos logros se deben a los avances en los métodos denominados de **aprendizaje profundo***. En este capítulo veremos en qué consisten los métodos de aprendizaje profundo más importantes, qué tipos hay y cómo se pueden programar y aplicar.

*** En inglés, *deep learning* (DL).**

6.1.1. Logros recientes

Los métodos de aprendizaje profundo han protagonizado numerosas noticias en los últimos años debido a diferentes logros en muchos tipos de aplicación de la inteligencia artificial (IA). Por citar algunos ejemplos destacables:

- En visión artificial han convertido en una tarea habitual reconocer caras, caracteres, tipos de objetos en una foto, vehículos y peatones, etc. Todo esto con una alta precisión. También han conseguido superar a los humanos en tareas tan complejas como distinguir entre varias especies de aves muy parecidas entre sí.

- En medicina se ha aplicado en diferentes tareas, como buscar proteínas que encajen con una determinada sustancia, identificar melanomas a partir de fotografías, detectar diferentes patologías en imágenes de resonancia magnética, etc.

- Se están desarrollando sistemas de conducción automática que son capaces de conducir un coche por una autopista real, utilizando información sencilla (cámaras). Según los desarrolladores, se han evitado numerosos accidentes gracias a esta tecnología. También se están desarrollando robots capaces de llevar a cabo tareas complejas, por ejemplo reaccionar aunque se encuentren piezas desordenadas.

- En procesamiento del lenguaje se ha conseguido una gran calidad en reconocimiento de voz, como el los famosos asistentes por voz que incluyen los móviles y otros dispositivos. También ha mejorado notablemente la traducción automática, que actualmente empieza a ser útil para diferentes propósitos.

- En diferentes ciencias se están aplicando para acelerar el procesamiento de datos, como por ejemplo para calcular la distorsión que un objeto masivo produce en la luz que llega de objetos astronómicos lejanos.

- Un sistema de aprendizaje profundo (AlphaGo) venció al campeón del mundo de Go, también conocido como damas chinas, un juego mucho más complejo que el ajedrez. También se ha desarrollado el sistema DQN, un sistema que es capaz de aprender a jugar a juegos de consola (concretamente una antigua vídeoconsola Atari) por sí mismo, sin utilizar ningún conocimiento previo sobre el juego, simplemente a partir de lo que ve en pantalla.

6.1.2. Causas

Los sistemas de aprendizaje profundo son una evolución de las **redes neuronales**, que son sistemas inspirados en las neuronas del cerebro para conseguir respuestas «inteligentes». Las redes neuronales empezaron a desarrollarse en la década de 1950, pero no han conseguido un interés tan generalizado hasta hace unos pocos años (a partir de 2010). ¿Por qué ahora sí han tenido un gran éxito en tantas tareas de IA? Básicamente se proponen tres razones para este resurgimiento de las redes neuronales:

- La disponibilidad de grandes conjuntos de datos (lo que se denomina *big data*). Los sistemas de aprendizaje profundo necesitan miles o millones de datos de ejemplo para conseguir resolver sus tareas con tan alto grado de precisión.

- Diferentes mejoras teóricas que permiten entrenar redes neuronales más complejas (concretamente, más *profundas*, como se verá más adelante). Anteriormente los métodos de entrenamiento fracasaban con redes complejas, lo que impedía conseguir el aprendizaje por niveles de complejidad, característico de los sistemas de aprendizaje profundo.

- Las mejoras en el hardware que permiten entrenar sistemas con millones de datos de ejemplo. Especialmente destacan los procesadores gráficos*, que en principio se desarrollaron para visualizar videojuegos, pero que la comunidad de IA ha aprovechado para entrenar sus métodos. La gran ventaja de usar GPU frente a CPU se encuentra en que las GPU tienen numerosas unidades de cálculo en paralelo, lo que acelera las operaciones sobre vectores en varios órdenes de magnitud.

*** En inglés, *Graphics Processing Units* (GPU).**

6.1.3. Arquitecturas

Una ventaja adicional de los sistemas de aprendizaje profundo es su diseño modular, lo que permite combinar diferentes componentes de la forma más

adecuada a cada tarea. Esa modularidad da lugar a diferentes arquitecturas, de las que las más importantes son las siguientes:

- Redes de **propagación hacia delante** (*feed-forward networks*), también denominadas **perceptrón multicapa**: son sistemas generalmente utilizados para tareas de clasificación y de regresión, en las que el flujo de información va desde la entrada hacia la salida, sin bucles ni retrocesos.
- Redes **convolucionales**, especializadas en procesamiento de imagen y otros datos estructurados, que incluyen elementos especializados en capturar patrones repetidos (por ejemplo, líneas dentro de una imagen).
- Redes **recurrentes**: en este tipo de redes existe algún tipo de recurrencia o vuelta atrás del flujo de datos, lo que les permite tener memoria. Se utilizan en procesamiento de secuencias: señales, texto, tareas que requieren múltiples pasos.
- **Autocodificadores**: sistemas no supervisados especializados en generar representaciones simplificadas de los datos de entrenamiento. Su objetivo es similar a los de los métodos de reducción de la dimensionalidad clásicos vistos en la asignatura, como PCA.
- Sistemas de **aprendizaje por refuerzo** (*reinforcement learning*): estrictamente no se trata de un tipo de arquitectura, sino de un nuevo tipo de tarea en el que el sistema se entrena por interacción con su entorno. Los sistemas de aprendizaje profundo se utilizan a menudo en estas tareas.
- Sistemas **generativos**, en los que hay dos sistemas opuestos y se hace que uno de ellos aprenda a «inventarse» nuevos datos, tan parecidos a los datos de entrenamiento que parezcan reales.

En apartados posteriores se estudiarán con mayor detalle estas arquitecturas, su aplicación y su programación.

6.1.4. Bibliotecas

El uso de métodos de aprendizaje profundo requiere la utilización de diferentes métodos matemáticos relativamente complejos, y que además deben ejecutarse con la máxima velocidad y aprovechando el hardware disponible para que su uso sea viable. Esto hace que la opción más habitual para utilizar estos métodos pase por el uso de una o más bibliotecas de programación que resuelvan buena parte de los problemas planteados y permitan centrarse en el desarrollo del sistema que la aplicación necesite.

Hoy en día existen numerosas bibliotecas de aprendizaje profundo, de las que destacan las siguientes (el lenguaje de programación es Python, salvo que se indique lo contrario):

- Theano: una de las primeras bibliotecas, desarrollada por los pioneros del aprendizaje profundo. Se trata de una biblioteca de bajo nivel, en el sentido de que ofrece diferentes operaciones matemáticas pero requiere que el usuario programe bastante para tener un sistema en marcha.

- Tensorflow: el competidor de Theano, desarrollado por Google. También es una biblioteca de bajo nivel.

- Lasagne: biblioteca de alto nivel que, utilizando Theano, ofrece a los usuarios instrucciones sencillas para crear sistemas de aprendizaje profundo.

- Keras: equivalente a Lasagne, con la ventaja de que puede operar sobre Theano y sobre Tensorflow.

- Caffe: biblioteca especializada en sistemas de visión artificial.

- nolearn: biblioteca de muy alto nivel, muy fácil de usar pero que a cambio permite pocos cambios.

- Torch: biblioteca en lenguaje Lua, utilizada por Facebook y Twitter entre otros.

- dl4j: biblioteca en Java, más orientada al entorno empresarial.

Como se ve, el lenguaje principal en esta área es Python. Los ejemplos que se presentarán en los apartados siguientes utilizan **Keras** sobre **Tensorflow**, ya que es quizá la combinación más popular actualmente, resulta sencilla pero a la vez permite diseñar sistemas especializados.

Keras y Tensorflow

También se puede utilizar Keras sobre Theano, aunque se recomienda Tensorflow porque en general resulta más sencillo de configurar y resulta más sencillo configurarlo para usar la GPU. Como son bibliotecas en actualización constante, se recomienda consultar las instrucciones de instalación en las páginas de los proyectos:
https://www.tensorflow.org/
https://keras.io/

6.2. Redes neuronales

El cerebro está formado por diferentes tipos de células. Las **neuronas** son las células que, estableciendo conexiones entre ellas y enviando señales por esas conexiones, dan lugar al comportamiento emergente que denominamos inteligencia.

El objetivo de las **redes neuronales** es crear un análogo computacional a las neuronas biológicas para intentar crear inteligencia en un ordenador. En ese sentido, se definen unos elementos llamados neuronas o simplemente **unidades** con conexiones con otras unidades por las que se propagan señales. Sin embargo la analogía no va mucho más allá y la experiencia en el área ha demostrado que lo que funciona en biología no tiene por qué funcionar en informática.

En este apartado veremos qué son y cómo funcionan las redes neuronales, que son la base de los sistemas de aprendizaje profundo.

6.2.1. Componentes de una red neuronal

Una red neuronal básica se compone de los elementos siguientes:

- Una capa de unidades de **entrada**, que reciben las variables del exterior disponibles para tratar el problema: los píxeles de una imagen, la posición de un objeto, los valores de ciertos productos, etc.
- Una capa de unidades de **salida**, compuesta por una o más unidades que producen una salida al exterior, que es el «resultado» de la red: la clase de imagen, el valor de tensión eléctrica necesario para accionar un motor, si se debe comprar o vender un producto, etc.
- Una capa de unidades **ocultas**, que reciben conexiones de la capa de entrada y se conectan a las unidades de salida.
- Por último, el conjunto de **conexiones** entre capas. En general las conexiones entre unidades son unidireccionales.

Como se ve, las unidades se organizan en capas. En el tipo de red neuronal más sencilla, la red neuronal prealimentada o perceptrón, las conexiones siempre van de las unidades de la capa de entrada a las de la capa oculta y de ahí a la capa de salida; las conexiones no vuelven atrás.

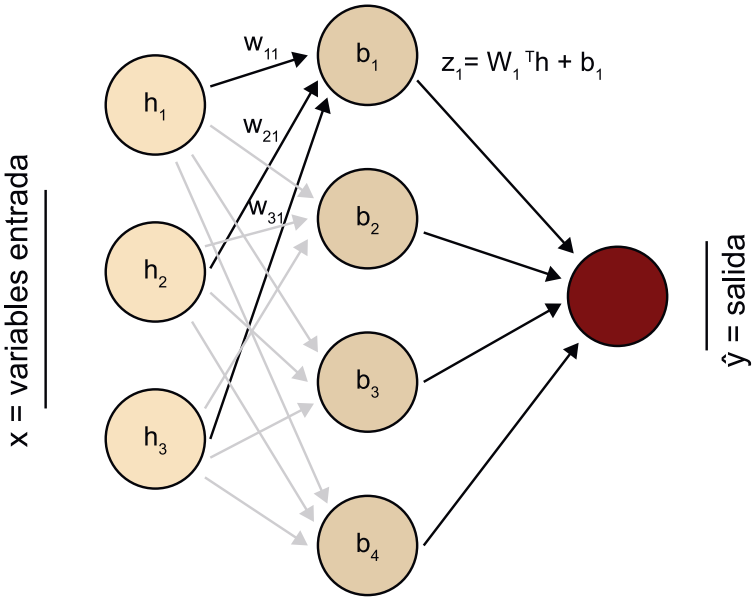
Así, el funcionamiento del perceptrón consiste básicamente en la **propagación** hacia delante de las señales de entrada, lógicamente modificadas a medida que atraviesan capas.

Cuando todas las unidades de una capa están conectadas a todas las unidades de la siguiente capa, se dice que esta segunda capa está **completamente conectada**. En el perceptrón las capas son así, pero en otras arquitecturas no tiene por qué ocurrir eso.

De forma intuitiva, y antes de entrar en más detalle, el funcionamiento del perceptrón es el siguiente:

1) Las unidades de entrada reciben valores del exterior, y se **activarán** —es decir, producirán un determinado valor a su salida— o no en función de la entrada recibida.

Figura 61. Red neuronal básica



2) Las salidas de la capa de entrada son, a su vez, las entradas de la capa oculta, de forma que estas unidades reciben un conjunto de entradas frente a las que reaccionarán. Para ello, cada unidad de la capa oculta tiene un **vector de pesos**, un valor por cada conexión entrante, que combina con las señales correspondientes y como resultado producen la activación o no de la salida de cada unidad oculta.

3) Finalmente, las unidades de la capa de salida también reciben las señales de la capa oculta, realizan una operación con esas señales y sus propios vectores de pesos y como resultado calculan su propia salida, que será el resultado de la red.

6.2.2. Funciones de activación

La forma en que una unidad combina sus entradas X con su vector de pesos W para calcular su salida z viene dada generalmente por la expresión:

$$z = W^T X + b \tag{72}$$

donde b es un valor escalar denominado sesgo (*bias*), necesario para garantizar un valor de base. El valor $z \in \mathbb{R}$ se utiliza a su vez como entrada para la **función de activación**, que es la que decide cuál es la salida final.

La función de activación más utilizada en las unidades de entrada y ocultas es la llamada **ReLU** (de *Rectified Linear Unit*), que es tan sencilla como:

$$g(z) = \text{máx}(0,z) \tag{73}$$

Es decir, la salida es 0 si la entrada es negativa, y es igual a z si esta es positiva.

Existen variaciones de la función ReLU que mejoran su comportamiento en algunas situaciones, como las funciones ReLU con pérdida, en la que la función tiene un pequeño gradiente negativo cuando $z < 0$, lo que facilita el entrenamiento de la red. También resulta interesante la familia de las **ELU** (*exponential linear units*), definidas como:

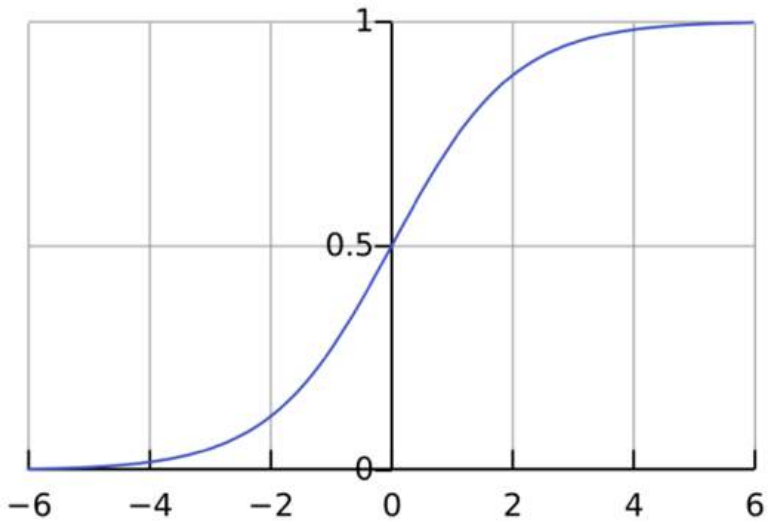
$$f(x) = \begin{cases} x & \text{si} \quad x \geq 0 \\ a(e^x - 1) & \text{si} \quad x < 0 \end{cases} \tag{74}$$

Experimentos recientes muestran que las ELU dan mejores resultados, pues su entrenamiento es más eficiente gracias a que su valor medio está más cercano a cero.

En el caso de unidades de salida, se suelen utilizar otras funciones de activación, que principalmente dependen del tipo de salida deseada:

- Si se trata de un valor **real**, típico en problemas de regresión, la función de activación es lineal, es decir $g(z) = z$.
- En el caso de valores **binarios**, para clasificación en dos clases o detección de anomalías, se suele utilizar la función logística sigmoide, mostrada a continuación.
- En valores **multiclase**, como en un clasificador, se utiliza la función **softmax**: hay una unidad de salida para cada clase, de forma que la salida activada tendrá un valor cercano a 1 y las demás tendrán valores cercanos a 0.

Figura 62. Función logística sigmoide



6.2.3. Entrenamiento de una red neuronal

Como se ha explicado, en una red neuronal de tipo perceptrón las señales se propagan hacia delante, es decir, de las entradas hacia las salidas, sin vueltas atrás. También se ha expuesto que cada unidad tiene un vector de pesos W, con un valor para cada una de sus entradas, así como un sesgo b. La pregunta que surge es: ¿de dónde surgen W y b? Al fin y al cabo estos valores son los que controlan el comportamiento de la red. Pues bien, esos valores no aparecen solos ni se ponen a mano, sino que es la propia red la que debe aprender los valores óptimos para conseguir realizar la tarea encomendada de la mejor forma posible.

Por tanto, al trabajar con redes neuronales se distinguen dos fases: una primera fase de **entrenamiento** en la que la red recibe datos de los que aprender y en la que va ajustando los pesos y sesgos, y otra fase de **ejecución**, en la que se utiliza la red para llevar a cabo la tarea tal y como la ha aprendido.

Los perceptrones son sistemas de aprendizaje **supervisado**, por lo que el entrenamiento de un perceptrón requiere un conjunto de datos etiquetados. Por otra parte, los pesos y sesgos se inicializan a valores aleatorios pequeños (en torno a 0,1). A continuación, se van inyectando los ejemplos de entrenamiento a la red y se analiza la diferencia entre la salida obtenida y la salida esperada (según la etiqueta asociada a cada ejemplo).

Esa diferencia entre la salida obtenida y la salida esperada se expresa mediante una **función de coste** C. En redes con una sola salida se puede utilizar como C la función de error cuadrático, mientras que en problemas con varias salidas suelen utilizarse la entropía cruzada o la divergencia de Kullback-Leibler.

El objetivo del entrenamiento es reducir los valores de C, es decir, la diferencia entre la salida obtenida y la esperada. Para ello hay que ajustar gradualmente los pesos y sesgos de cada capa utilizando el método de **descenso de gradientes**, visto en el subapartado 5.3, concretamente calculando el gradiente de la función de coste respecto a los pesos y sesgos de la unidad de salida:

$$\frac{\partial C}{\partial \hat{W}} \tag{75}$$

donde $\hat{W} = \{W,b\}$, es decir, se integran pesos y sesgo en un único vector para facilitar las operaciones.

Así pues, al determinar los $\hat{W}$ óptimos para reducir la función de coste C se está ajustando la capa de salida; sin embargo, las capas anteriores no han reci-

bido ningún ajuste. Aquí entra en juego el proceso denominado **propagación hacia atrás** (*backpropagation*): una vez ajustada la capa de salida, se procede a ajustar la capa oculta mediante el mismo procedimiento, y así se van ajustado las capas desde la salida hacia la entrada, de ahí que sea «hacia atrás».

De esta forma se acaban ajustando todos los parámetros de la red, haciendo que el conjunto de parámetros produzca el menor error posible con los datos de entrenamiento. Este **error de entrenamiento** puede ser distinto de cero en función de los datos y la complejidad del modelo; un modelo más complejo será capaz de aprender mejor los datos de entrenamiento, pero como se verá más abajo no es conveniente llevar esa idea al extremo para conseguir un error de entrenamiento igual a cero.

Otra cuestión importante es la forma en que se produce la optimización por descenso de gradientes, atendiendo a cómo se utilizan los ejemplos de entrenamiento. Hay tres formas de proceder:

- **Descenso de gradientes por lotes** (*batches*). Consiste en utilizar todos los ejemplos de entrenamiento a la vez. Aunque es el método que da mejores resultados, computacionalmente resulta muy costoso, no solo en tiempo sino también en memoria, que es un factor crítico si se usan GPU.

- **Descenso de gradientes estocástico**. Utiliza los ejemplos de entrenamiento de uno en uno. Aunque es un método rápido, provoca una gran varianza en el entrenamiento, ya que las características específicas de cada ejemplo condicionan enormemente los pesos elegidos.

- **Descenso de gradientes por minilotes** (*mini-batches*). Es una solución intermedia, en la que los ejemplos se agrupan en bloques y se ejecuta el descenso de gradientes sobre cada bloque. Así se reduce la varianza pero con un coste computacional moderado. Conviene elegir un tamaño de bloque tal que los ejemplos quepan en la memoria de trabajo. Es el método más utilizado.

Una ventaja adicional de la aproximación por minilotes es que permite múltiples iteraciones de entrenamiento sobre el mismo conjunto de datos, bien tomando los mismos lotes, bien particionando los datos de entrenamiento de forma distinta cada vez. A estas iteraciones de entrenamiento se les da el nombre de **épocas** (*epochs*).

6.2.4. Problemas de aprendizaje

En las redes neuronales, igual que en otros métodos de aprendizaje automático, pueden aparecer diferentes problemas de aprendizaje que es fundamental conocer, identificar y saber resolver adecuadamente, ya que en algunos casos

las soluciones son opuestas y hay que evitar cambiar parámetros al azar hasta que el sistema mejore.

En primer lugar, recordemos el protocolo de entrenamiento de un sistema: para entrenar correctamente un sistema de forma que posteriormente se pueda aplicar a nuevos datos con éxito, es necesario dividir los datos en tres bloques:

- **Entrenamiento** (*train*): los datos que el método usa para aprender, ajustando sus parámetros internos.

- **Validación** (*validation*): datos que se utilizan para probar el sistema una vez entrenado. Al ser datos diferentes se pueden usar para comprobar si el sistema es capaz de procesar datos nuevos. Estos datos ayudan al desarrollador a ajustar los hiperparámetros del sistema, es decir a cambiar su configuración, tras lo cual hay que volver a entrenarlo.

- **Prueba** (*test*): datos que se reservan hasta que el sistema está completamente entrenado y ajustado, sirven para valorar las prestaciones del sistema final.

También es fundamental que los tres bloques anteriores sigan la misma distribución de datos, es decir, que los ejemplos de diferentes clases y características estén repartidos por igual en ellos.

Figura 63. Entrenamiento, validación y test

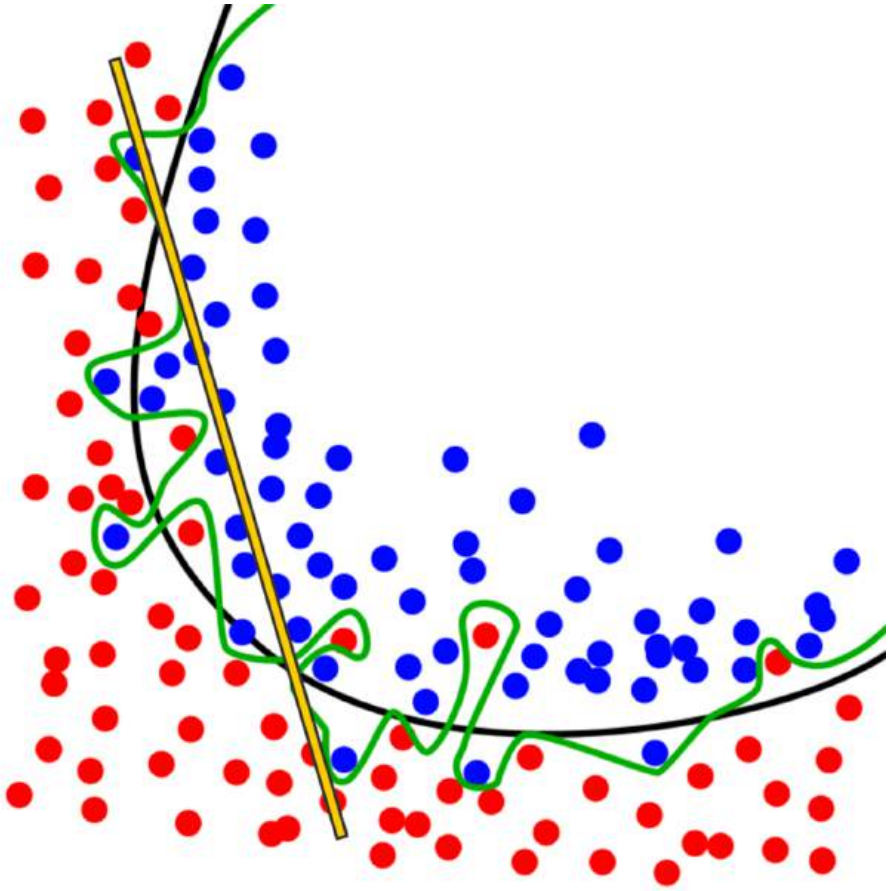
A grandes rasgos, podemos encontrarnos con dos tipos de errores, ilustrados en la figura 64.

- **Error de infraajuste** (*underfitting*): indica que el sistema no es suficientemente complejo para aprender los datos de entrenamiento. Para corregirlo hay que incrementar la complejidad del modelo (por ejemplo, más unidades en el caso de una red neuronal).

- **Error de sobreajuste** (*overfitting*): indica que el sistema no es capaz de generalizar lo aprendido con los datos de entrenamiento a otros datos. Conviene aplicar técnicas de **regularización**, explicadas más adelante.

El comportamiento recomendable, siguiendo con la figura, es el indicado por la línea negra: un modelo suficientemente complejo como para adaptarse a la estructura general de los datos, pero no tanto como para perderse intentando modelar los detalles de cada uno de los ejemplos, lo que le impide tener capacidad de **generalización**.

Figura 64. Tipos de errores: infraajuste (naranja) y sobreajuste (verde)



6.2.5. Algunas soluciones

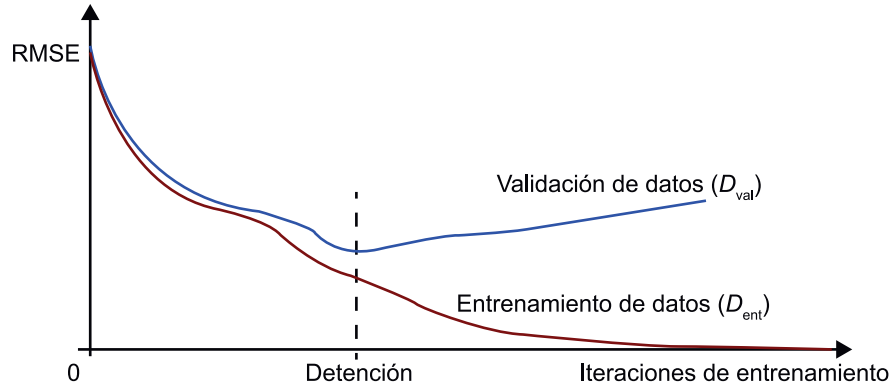
Muchas de estas técnicas pueden aplicarse también a otros métodos de aprendizaje automático.

De los problemas vistos anteriormente, sin duda el más complicado de subsanar es el del sobreajuste. En general, las técnicas cuyo objetivo es conseguir un modelo con mayor capacidad de generalización, y por tanto sin sobreajuste, reciben el nombre de técnicas de **regularización**. Veamos algunas de las técnicas de regularización más importantes:

- **Detención temprana** (*early stopping*): es una idea tan sencilla como detener el entrenamiento cuando el sistema está empezando a sobreajustarse a los datos de entrenamiento. Se puede detectar que eso ocurre cuando el error de **validación** empieza a crecer, aunque el error de entrenamiento siga disminuyendo, como se puede ver en la figura 65. Este método suele aplicarse a las épocas de entrenamiento, en especial cuando se utilizan los datos de entrenamiento varias veces.

- **Normalización de pesos**: una técnica sencilla que consiste en limitar los valores de los pesos de las unidades de la red neuronal, evitando valores muy grandes o muy pequeños. De esta manera se consigue que la red esté más equilibrada y no dé excesiva importancia a algunos ejemplos o combinaciones de variables, lo que al final evita el sobreajuste.

Figura 65. Evolución de los errores de entrenamiento y validación



Fuente: http://bit.ly/2ApsZar

- **Abandono** (*dropout*): una técnica sorprendente que se aplica a las redes neuronales, y que consiste en desactivar unidades aleatoriamente durante el entrenamiento. Esto provoca que el resto de unidades se hagan cargo de las tareas de las unidades desactivadas, lo que al final redunda en una red más robusta y más generalista. Esta técnica funciona muy bien y tiene poco coste computacional, por lo que se usa con mucha frecuencia, como se verá en los ejemplos de código.

- **Preparación de datos**: cuando los datos disponibles no son suficientes para conseguir el nivel de entrenamiento deseado, a veces conviene reducir la complejidad del problema para que la cantidad de ejemplos sea suficiente. Por ejemplo, en un sistema de vigilancia puede ser imposible entrenar con fotos completas de la entrada a un edificio, así que quizá convenga tomar solo las fotos de caras y entrenar con ellas, simplificando la tarea del sistema.

- **Aumento de datos**: una técnica complementaria de la anterior, en este caso se opta por aumentar el número de datos de entrenamiento generando datos falsos a partir de los ejemplos disponibles. Por ejemplo, añadiendo ruido a una grabación de audio, girando o recortando fotografías, etc.

6.2.6. Aprendizaje profundo

En este apartado hemos estudiado una red neuronal sencilla, el perceptrón, en la que solo hay tres capas. Una red con una sola capa oculta como esta es capaz de aprender cualquier conjunto de datos finito, siempre que tenga un número suficientemente grande de unidades en la capa oculta.

El problema de ese planteamiento, aumentar la anchura de la capa oculta, es doble: primero, es una solución computacionalmente costosa, ya que puede requerir una capa con muchos miles o incluso millones de unidades; segundo, un sistema así solo se aprenden las combinaciones de variables presentes en

los datos de entrenamiento, con lo que su poder de generalización es muy rudimentario.

Una estrategia alternativa para aumentar la complejidad de una red neuronal es incrementar el número de capas ocultas. Como resultado, se obtiene una red que es capaz de generalizar mejor a nuevos datos, ya que en cada capa se modela un nivel de abstracción superior que en la anterior, con lo que al final la red es capaz de detectar características más generales.

Ejemplo

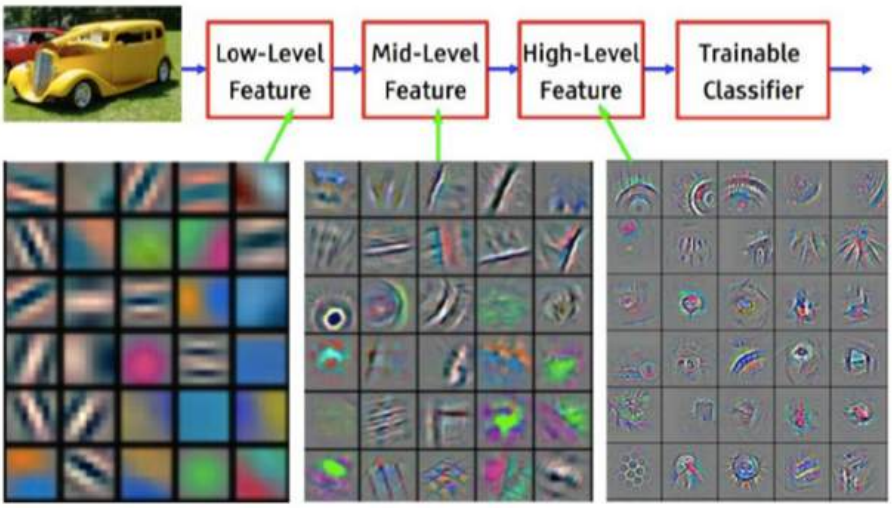
En un sistema de visión por ordenador con muchas capas ocultas, la primera capa detectará características sencillas como bordes; la segunda, elementos compuestos con los objetos de la anterior, como esquinas, cruces; la tercera, combinaciones de los anteriores, como triángulos, rectángulos, etc. Así sucesivamente hasta que las capas superiores son capaces de detectar características complejas como ojos, ruedas, matrículas, señales de tráfico, y así hasta producir un resultado global de gran calidad por su capacidad de abstracción.

Esta es la idea del aprendizaje profundo: crear redes neuronales con varias capas ocultas, de ahí lo de «profundo». Depende del autor, pero generalmente se considera profunda una red con diez o más capas.

Hasta hace unos años (alrededor del 2010) no se habían utilizado estos sistemas de forma generalizada por la dificultad y coste computacional de entrenarlos; sin embargo, diferentes avances teóricos y técnicos han abierto las técnicas de aprendizaje profundo y han revolucionado la IA.

En los subapartados siguientes estudiaremos diferentes tipos de redes neuronales profundas y sus aplicaciones.

Figura 66. Características de bajo, medio y alto nivel en análisis de imágenes



Fuente: http://bit.ly/2nreM7N

6.3. Perceptrón multicapa

6.3.1. Idea

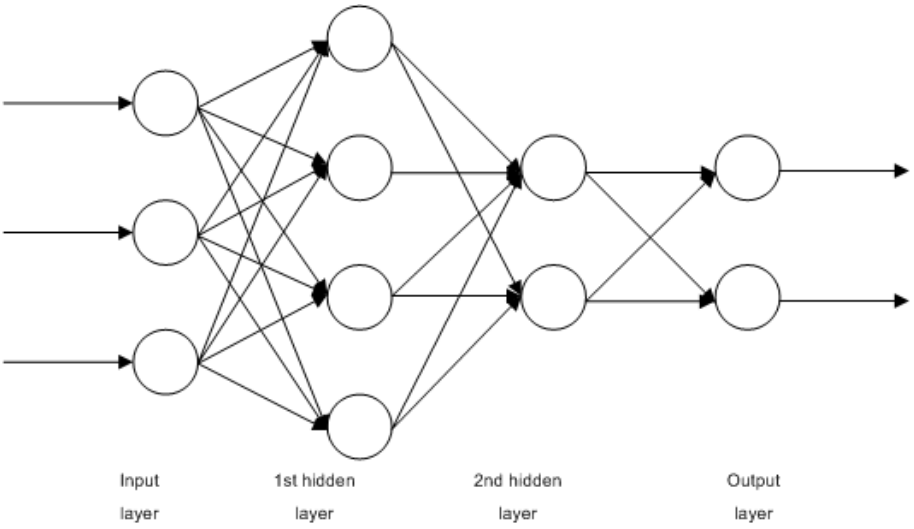
El tipo de red neuronal profunda más sencillo es el denominado **perceptrón multicapa*** o también red neuronal **prealimentada**.

*** En inglés, *multilayer perceptron* (MLP) o bien *feed-forward neural network* (FNN).**

Como su nombre sugiere, simplemente se trata de un perceptrón con varias capas ocultas. La ventaja de esa configuración es que las sucesivas capas ocultas van aprendiendo características cada vez más complejas, ya que parten de las características aprendidas por las capas anteriores.

Su estructura es muy sencilla: una capa de entrada, varias capas ocultas y una capa de salida. Todas las capas están completamente conectadas, y solo hay conexiones hacia delante.

Figura 67. Ejemplo de perceptrón multicapa



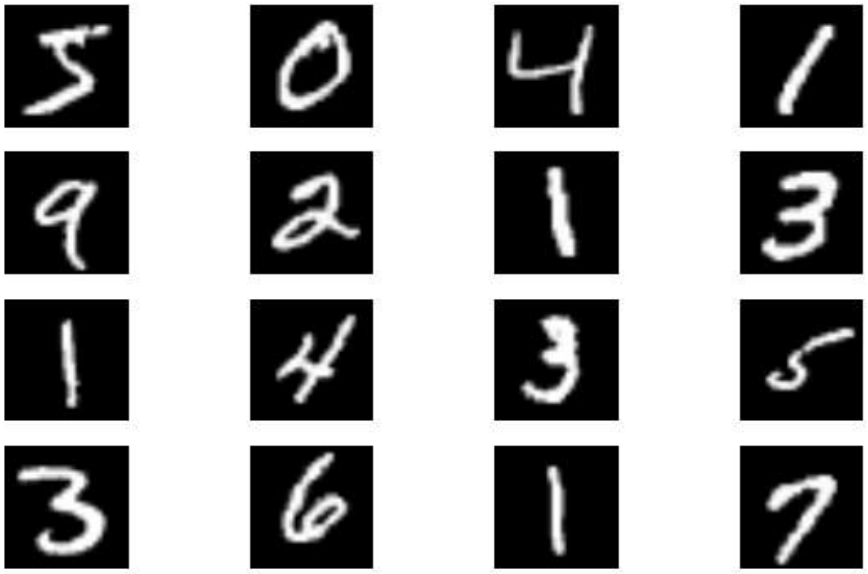
Fuente: Wikimedia.org

6.3.2. Ejemplo de MLP

Uno de los conjuntos de datos estándar para probar redes neuronales es la base de datos MNIST, que contiene 70.000 imágenes en escala de grises de dígitos manuscritos. Las imágenes tienen un tamaño de 28×28 píxeles. Cada imagen tiene una etiqueta de clase de 0-9 igual al dígito representado en la imagen. Los datos están repartidos en sesenta mil imágenes de entrenamiento y diez mil imágenes de test. El conjunto de datos se incluye en la distribución de las librerías Keras, lo que permite trabajar con él de forma muy sencilla*.

*** Se puede acceder a la información detallada en https://keras.io/datasets/.**

Figura 68. Dieciséis primeras imágenes de la base de datos MNIST



El siguiente programa muestra cómo crear un MLP que clasifique los datos de MNIST en cada una de las diez clases. La red se entrenará por minilotes de ciento veintiocho ejemplos, y el entrenamiento se repetirá veinte veces (épocas). Tras importar los paquetes necesarios, normalizar los datos y organizarlos en entrenamiento y test, se define el modelo de la red. En este caso se utiliza `keras.Sequential`, lo que indica que se trata de un MLP porque las capas se organizan secuencialmente, sin saltos ni vueltas atrás. Concretamente, esta red tiene una primera capa de entrada. A continuación, alterna una capa de abandono (*dropout*) con una capa densa. Finalmente, añade otra capa de abandono previa a la capa de salida, de tipo *softmax*, ya que se trata de un problema de clasificación multiclase.

La función de coste o pérdida (*loss*) es la entropía cruzada, en la versión diseñada para problemas de clasificación, y el optimizador es `RMSprop`, que es una versión mejorada del descenso de gradientes.

En Keras es necesario compilar el modelo, lo que lo prepara para su ejecución de forma eficiente, así como para definir cuál será la métrica que se usará para evaluar la calidad del modelo, en este caso la precisión en la clasificación.

En ese punto ya se puede entrenar el modelo con los datos de entrenamiento, y por último se evalúa con los datos de prueba.

Capa de abandono

Aunque se generen como una capa más, las capas de abandono no se cuentan como una capa más, sino como un modificador de la capa siguiente.

Código 6.1: perceptrón multicapa como clasificador de dígitos de MNIST

```
1  import keras
2  from keras.datasets import mnist
3  from keras.models import Sequential
4  from keras.layers import Dense, Dropout
5  from keras.optimizers import RMSprop
6
7  # Configuracion general del programa
8  batch_size  = 128
9  num_classes = 10
10 epochs      = 20
11
12
13 # Organizar datos en entrenamiento y prueba
```

```
(x_train, y_train), (x_test, y_test) = mnist.load_data()

x_train  = x_train.reshape(60000, 784)
x_test   = x_test.reshape(10000, 784)

# Los datos de entrada (pixeles) son enteros,
# convertirlos a reales para trabajar con ellos
x_train  = x_train.astype('float32')
x_test   = x_test.astype('float32')

# Normalizar los valores a 0..1
x_train /= 255
x_test  /= 255
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')

# Convertir cada numero de clase en un vector de 0s y 1s
# para poderlo comparar con la salida softmax
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)


# Definir el modelo de la red, con dos capas ocultas
model = Sequential()
model.add(Dense(512, activation='relu', input_shape=(784,)))
model.add(Dropout(0.2))
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(10, activation='softmax'))

model.summary()


# Compilar el modelo y entrenarlo
model.compile(loss='categorical_crossentropy',
              optimizer=RMSprop(),
              metrics=['accuracy'])

history = model.fit(x_train, y_train,
                    batch_size=batch_size,
                    epochs=epochs,
                    verbose=1,
                    validation_data=(x_test, y_test))

# Finalmente, evaluarlo con los datos de prueba
score = model.evaluate(x_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

Vale la pena destacar algunas cuestiones prácticas de este programa que hay que tener en cuenta al trabajar con redes neuronales:

- Es necesario **normalizar** los datos de entrada a la red, preferiblemente a un intervalo como 0..1, ya que si no la optimización puede tardar mucho más en converger.

- En problemas de clasificación múltiple hay que convertir los valores de clase en vectores de ceros y unos.

- La función de coste al entrenar la red no tiene por qué ser la misma que se use para evaluar la calidad del modelo. La función de coste tiene que ser fácilmente derivable, pero quizá no sea significativa para comparar la ejecu-

ción de varios sistemas. Por ese motivo, en Keras se especifica la función de coste (parámetro `loss`) y la métrica de evaluación (parámetro `metrics`).

6.4. Clasificación de imágenes con redes neuronales convolucionales (CNN)

Las redes neuronales convolucionales* son un tipo particular de redes neuronales que se utilizan para la clasificación automática de imágenes. Se trata entonces de una técnica de clasificación supervisada, por lo que es necesario disponer de una base de datos con imágenes etiquetadas según un conjunto predefinido de clases.

*** En inglés, *Convolutional Neural Networks* (CNN).**

De la misma forma que en las redes neuronales convencionales, las CNN presentan una arquitectura en red con capas de neuronas interconectadas entre sí. En una red neuronal, las interconexiones se realizan mediante unos parámetros (pesos y *offsets*) que pueden ser aprendidos durante la fase de entrenamiento. Una vez entrenada, la red neuronal puede ser utilizada como clasificador de patrones aplicando datos a la entrada y obteniendo una salida que predice la clase a la que pertenece. En nuestro caso, aplicaremos una imagen a la entrada y obtendremos una etiqueta de la clase a la que corresponde. Por supuesto, durante la fase de entrenamiento deberemos facilitar una cantidad suficiente de imágenes representativas de cada clase que deseemos clasificar.

A modo de ejemplo de aplicación, imaginad un clasificador automático con dos clases que distinga imágenes de playas y de bosques. Para ello necesitaremos una base de datos con un cierto número de imágenes de playas y de imágenes de bosques. Durante la fase de entrenamiento estableceremos los pesos de la red a partir de un conjunto de imágenes etiquetadas según su clase. Una vez definida la red, la podemos utilizar para clasificar automáticamente una nueva imagen.

La diferencia fundamental entre las redes neuronales convencionales y las redes neuronales convolucionales es que estas últimas están específicamente diseñadas para que los datos de entrada sean imágenes. Las CNN suponen un cambio de paradigma en el campo del reconocimiento de patrones en imágenes porque la propia arquitectura de la red se encarga de realizar la extracción de atributos y de su posterior clasificación automática. Las técnicas tradicionales de reconocimiento de patrones en imágenes requieren una extracción de atributos significativos de las imágenes antes de utilizar un clasificador supervisado para clasificarlas. Estos atributos pueden estar basados en aspectos como formas, color o texturas y suelen requerir un notable esfuerzo por parte de un experto que indique qué características son las más relevantes en un problema determinado.

Una imagen de tamaño $NX \times NY$ píxeles y NC canales de color se representa generalmente mediante una matriz de datos de dimensión $NX \times NY \times NC$. Por ejemplo, una imagen de 32×32 píxeles en un espacio de color RGB ($NC = 3$, tres canales de color Red, Green y Blue) se representa mediante una matriz de

tamaño $32 \times 32 \times 3$. En cada canal, los píxeles de la imagen pueden tomar valores discretos entre un conjunto de 2^N niveles, donde N es la profundidad en bits de la imagen (para una profundidad de $N = 8$ bits, por ejemplo, cada píxel puede tomar valores entre 0 y 255, esto son 256 niveles). Resulta claro entonces que son necesarios un total de $NX \cdot NY \cdot NC \cdot N$ bits para almacenar una imagen de tamaño $NX \times NY \times NC$ con una profundidad de bits N (por ejemplo, para almacenar una imagen RGB 16 bits de tamaño 32×32 se requieren un total de $32 \cdot 32 \cdot 3 \cdot 16 = 49.152$ bits = 6,144 kB).

En una red neuronal convencional con una imagen como dato de entrada, cada neurona de la capa de entrada podría llegar a tener un total de $NX \times NY \times NC$ conexiones, lo que en una imagen de $256 \times 256 \times 3$ supondría un total de 196.608 conexiones. Resulta evidente que en este caso la red tendría un excesivo número de parámetros que estimar, lo que supondrá un enorme coste computacional de la fase de entrenamiento y un alto riesgo de sobreajuste de los datos. Para evitar este problema, las redes convolucionales presentan una arquitectura en capas en la que cada capa consiste en un bloque de neuronas organizadas en un volumen regular.

El volumen de entrada, por ejemplo, es igual al tamaño de la imagen de entrada $NX \times NY \times NC$.

Aparte de la capa de entrada, existen tres tipos diferentes de capas: capas convolucionales (*convolutional layers*), capas de agrupación (*pooling layers*) y capas completamente conectadas (*fully-connected layers*), cuyo funcionamiento se describe brevemente a continuación:

1) Una **capa de convolución** aplica un conjunto de NF filtros al volumen de la imagen de entrada. Cada filtro consiste en una matriz de pesos de un cierto tamaño $M \times M \times NC$. A la extensión espacial del filtro $M \times M$ se le conoce como el campo receptivo de una determinada neurona. Los pesos de los filtros serán determinados durante la fase de entrenamiento de la red. Cada filtro se aplica a la imagen de entrada mediante una operación matemática conocida como convolución, en la que cada píxel de la imagen filtrada se obtiene calculando el promedio local ponderado utilizando los coeficientes del filtro como matriz de pesos. Repitiendo esta operación para cada píxel de la imagen de entrada y cada filtro de la capa convolucional, acabamos teniendo un resultado en forma de volumen de dimensiones $(NX - M + 1) \times (NY - M + 1) \times NF$. Al final de las capas convolucionales se aplica una función no lineal de activación que está inspirada en el funcionamiento de las neuronas en sistemas de percepción sensorial. Esta función tiene principalmente dos objetivos: por una parte, evitar que los valores crezcan indefinidamente a medida que los datos se propagan por la red. Por otra, permitir una mayor flexibilidad de la arquitectura para que la red pueda aprender relaciones no lineales y, por tanto, de mayor complejidad entre los atributos extraídos en cada capa. Una de las funciones de activación más utilizadas es la unidad de rectificación lineal

(*rectifier linear unit*, comúnmente conocida como ReLU), que realiza una operación $f(x) = \max(0,x)$.

2) Las **capas de agrupamiento** realizan una operación de submuestreo espacial de los datos, y por tanto agrupan características en las capas ocultas de la red. El resultado es entonces de dimensionalidad inferior a los datos de entrada de la capa. Una de las más utilizadas es la función *max-pooling*, que agrupa regiones de un cierto tamaño y las sustituye por el valor máximo de píxel en la región.

3) La **capa completamente conectada** consiste en una capa que conecta todas las neuronas de entrada con las de la salida. Aunque se puede utilizar en el diseño de las capas interiores de la red, habitualmente se utilizan en la capa de salida, proporcionando un índice de clase que codifica las NL clases del problema (proporcionando un volumen de salida de dimensiones $1 \times 1 \times NL$).

6.4.1. Implementación de las CNN en Python utilizando las librerías Keras

En este ejemplo de implementación de las CNN en Python volvemos a utilizar la base de datos MNIST, que contiene setenta mil imágenes en escala de grises ($NC = 1$) de un tamaño de 28×28. Cada imagen tiene una etiqueta de clase de 0-9 igual al dígito representado en la imagen. Los datos están repartidos en sesenta mil imágenes de entrenamiento y diez mil imágenes de test. El *dataset* se incluye en la distribución de las librerías Keras, lo que permite trabajar de forma muy sencilla*.

*** Se puede acceden a la información detallada en https://keras.io/datasets/.**

El código del ejemplo empieza cargando las librerías Keras y la base de datos MNIST. A continuación, se utiliza la instrucción para cargar las imágenes de entrenamiento y test en dos variables *X_train* y *X_test* con un tamaño de $60.000 \times 1 \times 28 \times 28$ y $10.000 \times 1 \times 28 \times 28$ respectivamente. El siguiente paso consiste en redimensionar cada conjunto de imágenes en vectores de tamaño $Nim \times NC \times NX \times NY$ píxeles utilizando la función *reshape** de las librerías NumPy. Posteriormente se normalizan los datos para que cada imagen se transforme de escala de grises entre 0-255 a valores reales entre 0-1.

*** Descrita en http://bit.ly/2jbFMnx**

Las etiquetas de clase deben ser codificadas en formato categórico en lugar de como valor entero. Para ello se debe utilizar la función *to_categorical* de las librerías NumPy. La variable categórica resultante consiste en un vector columna con tantas posiciones como número de clases en el que todas las posiciones son cero excepto la que indica la clase que deseemos codificar.

La función *baseline_model* es la que permite definir la arquitectura del modelo CNN*. El modelo CNN que vamos a utilizar se compone de seis capas que se describen a continuación:

*** Las funciones de diseño de capas convolucionales que ofrecen las librerías Keras están detalladas en http://bit.ly/2AMuMqq.**

Capa 0: la capa de entrada consiste en el volumen de datos de entrada, que en este caso es $Nim \times 28 \times 28 \times 1$.

Capa 1: capa convolucional 2D con $NF = 32$ filtros de tamaño 5×5 ($M = 5$) y una función de activación no lineal del tipo ReLU.

Capa 2: capa de agrupamiento *max-pooling* con un tamaño 2×2 (función MaxPooling2D).

Capa 3: capa de regularización que excluye aleatoriamente el 20 % de las neuronas para evitar fenómenos de sobreajuste de los datos.

Capa 4: capa de redimensionamiento de matrices 2D en vectores. El objetivo es proporcionar el formato adecuado a la capa completamente conectada que viene a continuación.

Capa 5: capa completamente conectada con ciento veintiocho neuronas y una función de rectificación ReLU como función no lineal de activación.

Capa 6: capa de salida con diez neuronas que codifican las diez clases del problema. Se aplica una función no lineal de activación del tipo exponencial normalizada (*soft-max*) para proporcionar una predicción probabilística de la etiqueta de clase.

En el código del ejemplo, la definición del modelo CNN se realiza en la función definida en la línea 35. Una vez definidas las capas descritas anteriormente, se especifican las características del algoritmo de optimización para el ajuste de los parámetros del modelo durante la fase de entrenamiento: función objetivo, algoritmo de optimización y métrica para evaluar el error. En este caso del ejemplo, se utiliza una función de coste de entropía cruzada, un algoritmo de descenso por gradiente. La actualización del gradiente se realiza aplicando la regla de actualización Adam, que utiliza un gradiente suavizado y aplica una corrección durante el régimen transitorio inicial. Por último, la exactitud (*accuracy*) como métrica de error de predicción. Esta métrica se utiliza también durante la fase de validación del modelo.

A continuación, se realiza la instanciación del modelo (línea 55) y por último se procede al ajuste del modelo (fase de entrenamiento, línea 58). En esta instrucción el parámetro *epochs* indica el número de veces que se itera el procedimiento de entrenamiento a partir de los datos de entrenamiento. El parámetro *batch_size* indica el número de observaciones que se utilizan en cada actualización del gradiente en el algoritmo de optimización. En la misma instrucción se valida los resultados del modelo utilizando los datos de validación y aplicando estos mismos parámetros.

```
1 # Cargar la base de datos MNIST y librerias Keras:
2 import numpy
3 from keras.datasets import mnist
4 from keras.models import Sequential
5 from keras.layers import Dense
6 from keras.layers import Dropout
7 from keras.layers import Flatten
```

```
8  from keras.layers.convolutional import Conv2D
9  from keras.layers.convolutional import MaxPooling2D
10 from keras.utils import np_utils
11 from keras import backend as K
12 K.set_image_dim_ordering('th')
13
14 # Establecer la semilla del generador de numeros aleatorios para
15 #garantizar reproducibilidad de los resultados
16 seed = 7
17 numpy.random.seed(seed)
18
19 # Acceder a las imagenes de entrenamiento y test
20 (X_train, y_train), (X_test, y_test) = mnist.load_data()
21
22 # Dimensionar las imagenes en vectores
23 X_train=X_train.reshape(X_train.shape[0],1,28,28).astype('float32')
24 X_test=X_test.reshape(X_test.shape[0],1,28,28).astype('float32')
25
26 # Normalizar las imagenes de escala de grises 0-255 (entre 0-1):
27 X_train = X_train / 255
28 X_test = X_test / 255
29
30 # Codificar las etiquetas de clase en formato de vectores
31 # categoricos con diez posiciones:
32 y_train = np_utils.to_categorical(y_train)
33 y_test = np_utils.to_categorical(y_test)
34 num_classes = y_test.shape[1]
35
36 # Funcion en la que se define la arquitectura del modelo:
37 def baseline_model():
38 # Capa de entrada:
39         model = Sequential()
40 # Primera capa (Convolucional):
41         model.add(Conv2D(32, (5, 5), input_shape=(1, 28, 28),
42                   activation='relu'))
43 # Tercera capa (agrupamiento):
44         model.add(MaxPooling2D(pool_size=(2, 2)))
45 # Cuarta capa (regularizacion):
46         model.add(Dropout(0.2))
47 # Quinta capa (redimensionamiento):
48         model.add(Flatten()):
49 # Sexta capa (completamente conectada)
50         model.add(Dense(128, activation='relu'))
51 # Capa de salida:
52         model.add(Dense(num_classes, activation='softmax'))
53 # Compilar el modelo y especificar metodo y metrica de optimizacion:
54         model.compile(loss='categorical_crossentropy',
55                       optimizer='adam', metrics=['accuracy'])
56         return model
57
58 # Llamada al modelo:
59 model = baseline_model()
60
61 # Ajuste del modelo:
62 model.fit(X_train, y_train, validation_data=(X_test, y_test),
63           epochs=10, batch_size=200, verbose=2)
64
65 # Evaluacion del modelo utilizando los datos de test:
66 scores = model.evaluate(X_test, y_test, verbose=0)
67 print("Exactitud del modelo: %.2f%%" %(100*scores[1]))
```

Durante la ejecución de la fase de entrenamiento aparecen en pantalla los resultados obtenidos en cada una de las diez iteraciones. En el código del ejemplo los resultados son los siguientes, que deberían ser reproducibles si se aplica la misma semilla del generador de números aleatorios que aparece en las líneas 15 y 16:

```
>Train on 60000 samples, validate on 10000 samples
Epoch 1/10
175s - loss: 0.2310 - acc: 0.9345 - val_loss: 0.0826 - val_acc: 0.9742
Epoch 2/10
180s - loss: 0.0737 - acc: 0.9780 - val_loss: 0.0471 - val_acc: 0.9839
Epoch 3/10
173s - loss: 0.0534 - acc: 0.9837 - val_loss: 0.0429 - val_acc: 0.9860
Epoch 4/10
173s - loss: 0.0403 - acc: 0.9878 - val_loss: 0.0405 - val_acc: 0.9866
Epoch 5/10
172s - loss: 0.0336 - acc: 0.9894 - val_loss: 0.0344 - val_acc: 0.9880
Epoch 6/10
172s - loss: 0.0274 - acc: 0.9916 - val_loss: 0.0308 - val_acc: 0.9898
Epoch 7/10
175s - loss: 0.0233 - acc: 0.9927 - val_loss: 0.0353 - val_acc: 0.9881
Epoch 8/10
173s - loss: 0.0203 - acc: 0.9936 - val_loss: 0.0326 - val_acc: 0.9887
Epoch 9/10
172s - loss: 0.0170 - acc: 0.9943 - val_loss: 0.0304 - val_acc: 0.9901
Epoch 10/10
172s - loss: 0.0144 - acc: 0.9956 - val_loss: 0.0317 - val_acc: 0.9902
```

En las dos últimas líneas de código se evalúa la exactitud (*accuracy*) global del modelo, que en este caso es del 99,2 %.

6.5. Redes recurrentes

6.5.1. Idea

Las redes neuronales vistas hasta ahora solo pueden procesar datos de longitud conocida, definida por la anchura de la capa de entrada. Sin embargo, no pueden llevar a cabo tareas sobre tipos de datos secuenciales, como texto, sonido, señales, etc., Aunque sería posible introducir cada elemento (por ejemplo, cada palabra) en una red MLP o CNN secuencialmente, estas redes tratan cada ejemplo independientemente de los anteriores, por lo que no tendrían en cuenta para nada el orden de las palabras en un texto, por ejemplo. Por esa razón, MLP y CNN no son adecuadas para trabajar con datos secuenciales y estructurados, ya que el flujo de señales en ellas es solo hacia delante.

Para que una red neuronal tenga en cuenta la historia de datos leídos y, por tanto, sea capaz de procesar secuencias de forma adecuada, es necesario que tenga algún tipo de **memoria** de los datos leídos anteriormente. En este contexto, ello implica algún tipo de vuelta atrás de las señales hacia capas anteriores. Este tipo de redes se denominan **recurrentes** (*recurrent neural networks*, RNN).

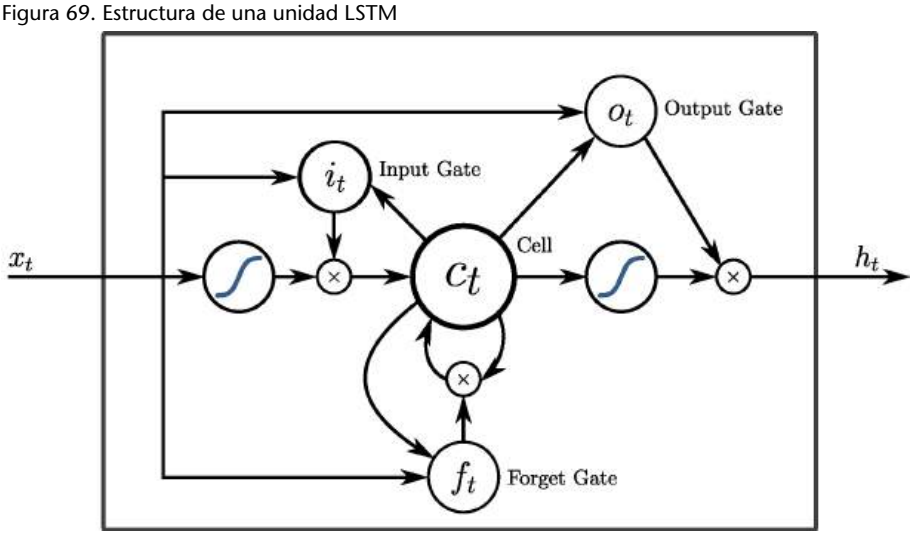
Existen muchas formas de crear una RNN, de las cuales la más sencilla es añadir una o más conexiones hacia atrás en una FNN. Sin embargo, esta aproximación general es muy inestable y extremadamente difícil de entrenar por efecto de la realimentación de señales, que provocan que los valores altos crezcan sin control y los bajos se apaguen, dando lugar a lo que se denomina el problema de los gradientes evanescentes.

Por ese motivo, en la práctica la forma de crear una RNN es utilizar unidades especiales que tienen memoria y que se ha comprobado que no tienen problemas de entrenamiento.

El tipo de unidad más habitual en RNN es la llamada unidad con memoria a corto y largo plazo (*long short-term memory* o LSTM), diseñada para converger rápidamente en su entrenamiento a la vez que es capaz de aprender dependencias a largo plazo en los datos. Su estructura viene dada en la figura siguiente, de la que básicamente se puede decir sin entrar en mucho detalle que la clave de las LSTM para conseguir estabilidad en el entrenamiento y en los patrones detectados consiste en que no aplica la función de activación a sus componentes recurrentes; al evitar esa realimentación, los valores numéricos son estables.

El comportamiento de la unidad LSTM está controlado por tres «puertas» (*gates*): la puerta de entrada (i_t), que controla si un nuevo valor entra en la memoria; la puerta de olvido (f_t), que controla si un valor existente se debe reemplazar con un valor nuevo; y la puerta de salida (o_t), que controla si el valor almacenado se usa para calcular la función de activación y por tanto la salida de la unidad. Los valores de las puertas dependen a su vez de los pesos aprendidos por la unidad.

Figura 69. Estructura de una unidad LSTM



Fuente: Wikimedia.org.

Es muy sencillo utilizar LSTM para crear RNN, en Keras simplemente se puede crear una capa de tipo LSTM. De hecho, lo habitual es combinar capas LSTM con capas normales (FNN), de forma que haya un cierto preproceso con capas FNN y así la(s) capa(s) LSTM reciban características más elaboradas y significativas.

En 2014 se propuso una simplificación de las LSTM denominada **unidad recurrente con puertas** (*gated recurrent unit*, GRU) que consigue un comportamiento equivalente con menor complejidad computacional.

Aunque se utilicen unidades fiables como LSTM o GRU, las RNN siguen siendo más difíciles de ajustar y requieren mucho cuidado en sus diferentes hiperparámetros.

6.5.2. Programación

El sitio web IMDB* (*Internet Movie DataBase*) es uno de los sitios de cine y películas más populares en internet. Las miles de opiniones escritas por los usuarios se han aprovechado para crear un corpus de opiniones escritas junto a una valoración que indica si son positivas o negativas. Utilizaremos este conjunto de datos para entrenar una RNN como clasificador binario.

* http://www.imdb.com/

En el caso de procesamiento de secuencias, son necesarios algunos pasos especiales para preparar los datos:

- En primer lugar, puede haber muchas palabras en el corpus, pero en el ejemplo se usaran solo veinte mil palabras distintas. Igualmente, solo se tomarán las primeras ochenta palabras de cada opinión.

- Como algunas opiniones pueden tener menos de ochenta palabras, se hace necesario rellenar (*pad_sequence*) las opiniones cortas para que todos los ejemplos quepan en una matriz cuadrada.

- En este ejemplo de procesamiento de texto hay muchas palabras distintas (veinte mil). Si usamos una codificación simple, tendremos que representar cada palabra como un vector de 20.000 bits, en el que todos valdrán cero menos el correspondiente a la palabra que entre en cada momento. Esto es muy ineficiente computacionalmente y pobre desde el punto de vista de la representación de la información. Por ese motivo, lo que se suele hacer en estos casos es generar una **proyección** (*embedding*) de los datos a un espacio vectorial de muchas menos dimensiones, del orden del centenar. Cada palabra del corpus se convertirá en un punto en ese espacio vectorial, lo que además de condensar la información ayuda a asociar conceptos cer-

canos. Para conseguirlo se crea una **capa de proyección**, cuya salida son vectores en el espacio vectorial proyectado. Esto, además, es mucho más eficiente computacionalmente hablando y simplifica el diseño de la red, ya que la salida de la capa de proyección tiene un tamaño razonable como entrada para las siguientes capas. Es importante destacar que es la propia capa la que aprende la proyección más adecuada durante el entrenamiento de la red.

Por otra parte, las capas de una RNN que usa LSTM tienen una disposición secuencial, como en el caso de una FNN, y la única diferencia es que una de las capas es de tipo LSTM. De hecho, se podrían añadir capas densas o de convolución si se considerara útil.

En la red que se construye en el ejemplo siguiente hay tres capas: la de proyección, que reduce el vocabulario a un espacio de ciento veintiocho dimensiones; la LSTM, con factor de abandono incluido; y la capa de salida, en este caso con una sola unidad porque la respuesta en este caso es binaria.

La función de coste es la entropía cruzada versión binaria, y el optimizador es Adam, una de las mejores variantes del descenso de gradientes actualmente.

Código 6.2: clasificador de opiniones de películas mediante RNN con LSTM

```
from keras.preprocessing import sequence
from keras.models import Sequential
from keras.layers import Dense, Embedding
from keras.layers import LSTM
from keras.datasets import imdb

# Numero de palabras distintas usadas como maximo
max_features = 20000
# De cada opinion se toman las 80 primeras palabras
maxlen = 80
# Y las opiniones se agrupan en lotes de 32
batch_size = 32

# Cargar los datos
print('Loading data...')
(x_train, y_train), (x_test, y_test) =
          imdb.load_data(num_words=max_features)
print(len(x_train), 'train sequences')
print(len(x_test), 'test sequences')

# Empaquetar los ejemplos en matrices cuadradas (rellenar)
print('Pad sequences (samples x time)')
x_train = sequence.pad_sequences(x_train, maxlen=maxlen)
x_test = sequence.pad_sequences(x_test, maxlen=maxlen)
print('x_train shape:', x_train.shape)
print('x_test shape:', x_test.shape)

# Crear el modelo con tres capas
print('Build model...')
model = Sequential()
model.add(Embedding(max_features, 128))
model.add(LSTM(128, dropout=0.2, recurrent_dropout=0.2))
model.add(Dense(1, activation='sigmoid'))

# Compilar y entrenar
model.compile(loss='binary_crossentropy',
              optimizer='adam',
```

```
38                 metrics=['accuracy'])
39
40 print('Train...')
41 model.fit(x_train, y_train,
42           batch_size=batch_size,
43           epochs=15,
44           validation_data=(x_test, y_test))
45
46 # Evaluar con los datos de test
47 score, acc = model.evaluate(x_test, y_test,
48                             batch_size=batch_size)
49 print('Test score:', score)
50 print('Test accuracy:', acc)
```

6.6. Otras arquitecturas

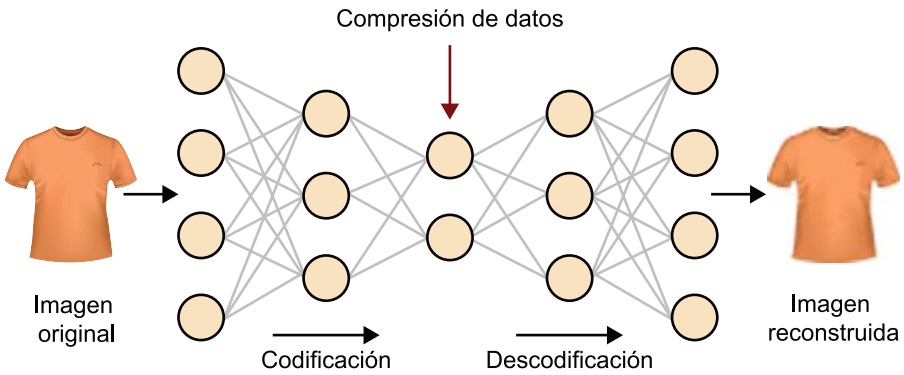
Existen muchas otras formas de diseñar una red neuronal para resolver diferentes tareas. En este subapartado veremos algunas de las más relevantes.

6.6.1. Autocodificadores

Los autocodificadores (*autoencoders*) son redes neuronales de propagación hacia adelante no supervisadas, en las que el objetivo de la red es que la salida reproduzca una copia idéntica a la entrada. ¿Qué sentido tiene eso? El detalle importante es que aunque lógicamente la capa de salida tiene la misma anchura que la de entrada, las capas internas tienen menos unidades. Eso obliga a la red a aprender una representación más compacta de los datos de entrenamiento, es decir, una representación con menos dimensiones. De esta forma, su objetivo es similar al de los métodos de reducción de dimensionalidad como PCA.

En la figura 70 puede verse un diagrama sencillo de un autocodificador. En la práctica pueden tener muchas más capas; es importante que la anchura de las capas se reduzca progresivamente.

Figura 70. Estructura de un autocodificador



La figura 71 compara la calidad de la representación comprimida de una serie de fotografías de caras. La primera fila son las fotografías originales; la segunda, la representación generada por un autocodificador; la tercera, la representación generada por PCA. La dimensión de la capa interna del autocodificador es la misma que el número de componentes principales utilizado. Como se ve el autocodificador es capaz de reproducir las imágenes originales con mayor fidelidad.

Figura 71. Comparación de autocodificadores y PCA

Fuente: Wikimedia.org

6.6.2. Aprendizaje por refuerzo

Si bien el aprendizaje por refuerzo (*reinforcement learning* o RL) ha experimentado un enorme crecimiento con los sistemas de aprendizaje profundo, hay que aclarar que son dos conceptos distintos. Veamos en primer lugar en qué consiste el aprendizaje por refuerzo, para después explicar cómo se le puede aplicar soluciones con redes neuronales profundas.

A diferencia de las situaciones de aprendizaje automático «clásicas», en las que existe un conjunto de datos, etiquetados o no, con los que entrenar un sistema, en el aprendizaje por refuerzo un sistema aprende por su interacción con el entorno, concretamente por la situación que percibe, las acciones que ejecuta y las consecuencias de sus acciones.

Imaginemos un robot que tiene que recoger las piezas de una habitación: supongamos que el robot toma una imagen de la habitación, decide moverse hacia un punto determinado y coger una pieza; como consecuencia, verá que hay una pieza menos en la habitación y que esa operación ha tenido éxito. Por el contrario, si va hacia un punto en el que no hay ninguna pieza e intenta coger algo, verá que no ha avanzado hacia su objetivo final.

De la misma forma, en el aprendizaje por refuerzo el **agente**, es decir, el sistema que aprende y actúa, guía su aprendizaje por las **recompensas** que recibe a consecuencia de sus acciones. Las recompensas pueden ser de diferente naturaleza, aunque siempre orientadas a premiar el avance hacia el objetivo del agente: puntos conseguidos en un juego, kilómetros conducidos hacia el destino sin accidentes, piezas ensambladas por un robot, etc.

De manera más formal, un sistema de aprendizaje por refuerzo se compone de los elementos siguientes:

- Un conjunto de **estados** del problema, S. En el juego de tres en raya, por ejemplo, S será todos los estados posibles de cruz, círculo o casilla vacía, un total de 3^9 estados posibles, aun siendo un juego tan sencillo. Sin embargo, el número de estados posibles se dispara rápidamente, a unas 10^{40} posiciones válidas para el ajedrez o unas 10^{100} para el Go. Fuera del ámbito de los juegos, el número de estados posibles es imposible de calcular y, en la práctica, infinito: ¿cuántos estados posibles hay para el problema de la conducción automática? ¿Cuántas combinaciones de vehículos, condiciones ambientales, carretera, estado del propio vehículo, peatones, etc.?

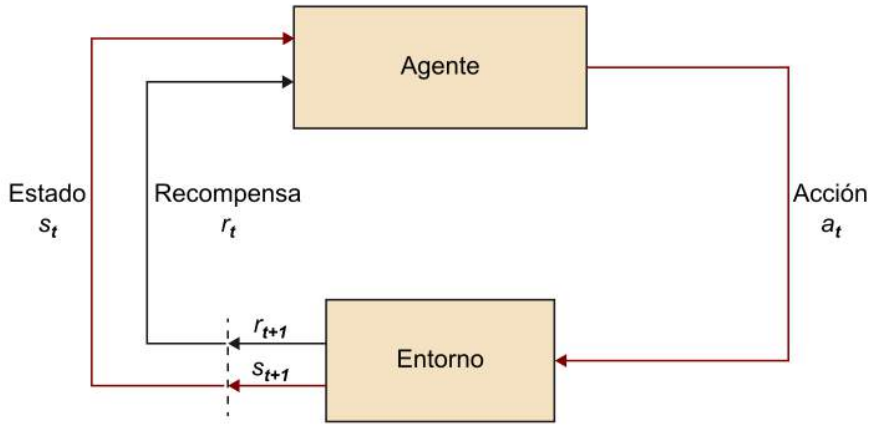
- Un conjunto de posibles **acciones**, A. En un juego serían los posibles movimientos. Manejando un robot A son los movimientos que puede ejecutar, en un sistema de conducción automática las diferentes acciones que puede ordenar al coche: acelerar, frenar, girar, etc.

- Una función de **recompensa**, R, que devuelve un número real en recompensa por la acción tomada por el agente en un estado determinado, es decir, $R : S \times A \rightarrow \mathbb{R}$. Es muy importante destacar que la recompensa depende de la acción tomada y del estado actual, ya que a veces será bueno que el coche acelere pero otras veces no.

- En teoría, un agente de un sistema de RL debería aprenderse una **tabla de recompensas** $Q = S \times A$ para así saber perfectamente cuál es la acción que se debe aplicar en cada estado del sistema. Sin embargo, como para casi cualquier problema S tiene un tamaño casi infinito, es imposible almacenar ese conocimiento en una tabla. De ahí que se hayan propuesto diferentes métodos para aprender aproximaciones razonablemente buenas a Q, métodos que reciben el nombre de Q-Learning.

La figura 72 resume los elementos de un sistema RL que se acaban de explicar.

Figura 72. Diagrama de control de un sistema de aprendizaje por refuerzo



Fuente: tomado de los materiales de IA para videojuegos (UOC)

Enlace de interés

La página https://gym.openai.com/ contiene gran cantidad de juegos preparados para entrenar sistemas de aprendizaje por refuerzo.

Aunque hay varias estrategias para Q-Learning, los sistemas que utilizan redes neuronales profundas están obteniendo grandes éxitos en diferentes ámbitos, como el sistema DQN, que aprende a jugar autónomamente a juegos de consola; el sistema AlphaGo, que venció al campeón mundial de Go, o numerosos sistemas de conducción automática y robótica, entre otros.

Los tipos de redes neuronales empleados para RL dependen de la tarea en cuestión; si por ejemplo la entrada sensorial es de imagen o vídeo, la primera fase del sistema suele ser una red CNN, tras la que se conecta una red FNN. También pueden utilizarse redes RNN si al tener en cuenta los estados anteriores se puede mejorar el rendimiento del sistema.

6.6.3. Sistemas generadores

Las **redes generativas antagónicas** (*generative adversarial networks*, GAN) son sistemas no supervisados cuyo objetivo es generar falsos ejemplos, pero que resulten creíbles y engañen a expertos. Por ejemplo, entrenada con fotos de caras, una GAN es capaz de crear caras nuevas que parezcan reales. También puede utilizarse para modificar contenidos, por ejemplo redibujar una fotografía normal al estilo de algún pintor famoso, colorear una fotografía en blanco y negro, o convertir un dibujo esquemático en una fotografía con materiales.

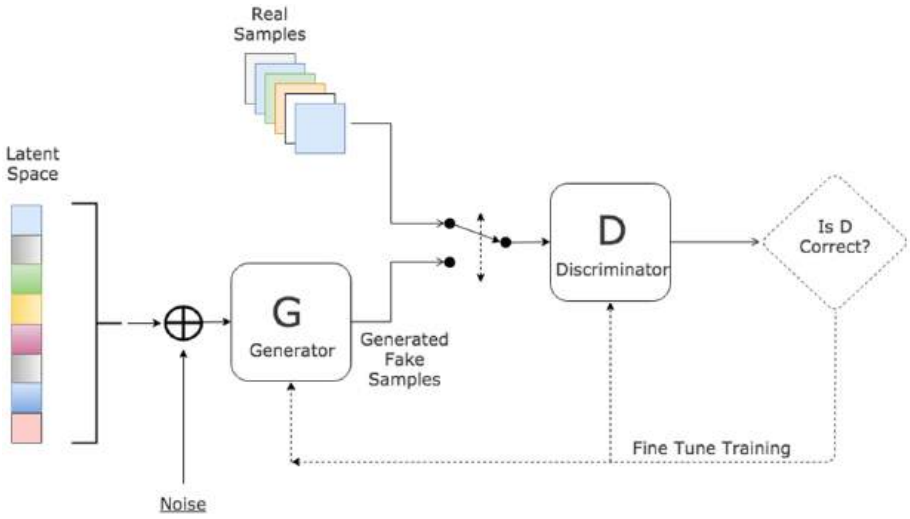
Figura 73. Cuadros pintados por un sistema GAN a partir de una fotografía

Una GAN se compone de dos redes independientes: una llamada **discriminador**, que se especializa en distinguir ejemplos verdaderos de falsos; y otra llamada **generador**, que se especializa en generar ejemplos falsos a partir de un espacio de ejemplos posibles. La clave de una GAN es que las redes se ponen en competencia, de forma que el discriminador aprende a distinguir mejor los

ejemplos falsos de los verdaderos y el generador, a cambio, tiene que generar cada vez ejemplos más creíbles y por tanto de mayor calidad.

Respecto al tipo de red, las GAN se utilizan principalmente en procesamiento y generación de imagen; en ese caso, el discriminador suele ser una CNN, mientras que el generador se construye mediante una red **deconvolucional**, que a grandes rasgos es como una CNN pero al revés, que pasa de menos unidades a más en la salida (similar a la mitad posterior de una red autocodificadora).

Figura 74. Estructura de una red generativa antagónica



Fuente: http://www.kdnuggets.com/2017/01/generative-adversarial-networks-hot-topic-machine-learning.html

7. Anexo: conceptos básicos de estadística

En el apartado 3 definimos la densidad de probabilidad $\rho_x(k)$ en un intervalo b_k, que se define entonces como el número de valores que se han producido en dicho intervalo h_k normalizado con el área total del histograma

$$\rho_x(k) = \frac{h_k}{n\Delta x}, \tag{76}$$

y la función densidad de probabilidad continua tomando el límite en el que el tamaño del intervalo Δx se hace infinitesimalmente pequeño

$$\rho_x(x) = \lim_{\Delta x \to 0, m \to \infty} \rho_x(k). \tag{77}$$

Esta función permite a su vez definir la función de probabilidad acumulada

$$p(x \leq x_0) = \int_{-\infty}^{x_0} \rho_x(x) dx, \tag{78}$$

que indica la probabilidad de que x tome un valor inferior o igual a x_0. También permite definir los momentos estadísticos de la distribución de probabilidad de $x(t)$. El primer momento o media viene dado por

$$E[x] = \bar{x} = \int_{-\infty}^{\infty} x \rho_x(x) dx, \tag{79}$$

que corresponde al valor medio de la señal x, que denotaremos $\bar{x}$. Desde un punto de vista geométrico, el valor medio $\bar{x}$ puede ser interpretado como el centro de masas de la distribución de probabilidad ρ_x.

De forma equivalente, el segundo momento de la distribución de $x(t)$ viene dado por

$$E[x^2] = \int_{-\infty}^{\infty} x^2 \rho_x(x) dx, \tag{80}$$

y puede ser expresado utilizando la definición del primer momento como

$$E[x^2] = E[x]^2 + E[(x - E[x])^2] = \bar{x}^2 + E[(x - \bar{x})^2]. \tag{81}$$

El segundo término de la última expresión se denomina varianza de la distribución

$$Var[x] = E[(x - \bar{x})^2], \tag{82}$$

y su raíz cuadrada es la desviación típica

$$\sigma = \sqrt{Var[x]}, \tag{83}$$

que constituye una medida de la variabilidad de los valores de $x(t)$ en torno a su valor medio $\bar{x}$. En general, el momento de orden p de la distribución ρ_x vendrá dado por

$$E[x^2] = \int_{-\infty}^{\infty} x^p \rho_x(x) dx. \tag{84}$$

Aparte de la media y la varianza, dos de los momentos más relevantes son la asimetría, definida como el momento central (momento de la variable $x - \bar{x}$) de tercer orden

$$E[(x - \bar{x})^3] = \int_{-\infty}^{\infty} (x - \bar{x})^3 \rho_x(x) dx, \tag{85}$$

y la curtosis, definida como el cociente entre los momentos de tercer y cuarto orden con una corrección para que una distribución gaussiana tenga curtosis nula

$$K = \frac{E[x^3]}{E[x^4]} - 3. \tag{86}$$

La curtosis permite cuantificar la medida en que una distribución de probabilidad se desvía de un patrón gaussiano, y es uno de los estadísticos que se utilizan en el cálculo de las componentes independientes del método ICA. El código 7.1 explica cómo calcular los momentos estadísticos de un conjunto de datos. En este ejemplo se utilizan datos gaussianos de media $\mu = 10$ y desviación típica $\sigma = 5{,}5$ (la varianza es $\sigma^2 = 30{,}25$). Al final del código se muestran los valores muestrales estimados, que coinciden con los valores poblacionales correspondientes.

Código 7.1: Cálculo de los momentos estadísticos de un conjunto de datos

```
1 from scipy import stats
2 import numpy as np
3
4 # datos gaussianos:
5 mu, sigma = 10, 5.5
6 y = mu + sigma*np.random.randn(10000)
7
8 # Estimación de momentos:
9 # momento de primer orden: media
```

```
10 media = y.mean()
11
12 # momento segundo orden: varianza
13 var = y.var()
14
15 # momento tercer orden: asimetría (skweness)
16 skew = stats.skew(y)
17
18 # momento cuarto orden: Curtosis
19 kurt = stats.kurtosis(y)
20
21 >>> print media, var, skew, kurt
22 10.0575611617 30.9623694469 0.0126290768091 0.025294666173
```

En caso de tener que calcular los momentos de la distribución a partir de un conjunto discreto de valores de la señal $\{x_1,x_2,\ldots,x_n\}$, procederemos a realizar una estimación muestral. Dos posibles estimadores muestrales (aunque no los únicos) de la media $\bar{x} = E[x]$ y la varianza $\sigma = \sqrt{Var[x]}$ vienen dados, respectivamente, por

$$\bar{x} \approx \frac{1}{n}\sum_{i=1}^{n} x_i \tag{87}$$

$$\sigma \approx \frac{1}{n}\sum_{i=1}^{n}(x_i-\bar{x})^2. \tag{88}$$

Los estimadores muestrales presentan un error respecto a los valores reales de la distribución (valores poblacionales), puesto que para cada conjunto de valores $\{x_1,x_2,\ldots,x_n\}$ se obtiene un valor estimado diferente. El error de estimación consta típicamente de dos partes, una debida a la desviación en media del estimador respecto al valor real conocida como sesgo, y otra parte que da cuenta de la dispersión de los valores estimados a partir de diferentes conjuntos de valores conocida como varianza. Dependiendo del estimador muestral escogido, tendremos una cierta ponderación de error debido a sesgo y debido a varianza. Por ejemplo, es fácil demostrar que mientras que $\frac{1}{n}\sum_{i=1}^{n}(x_i-\bar{x})^2$ estima la desviación típica σ de forma sesgada, el estimador

$$\sigma \approx \frac{1}{n-1}\sum_{i=1}^{n}(x_i-\bar{x})^2 \tag{89}$$

es un estimador no sesgado de la desviación típica de una distribución.

Actividades

1. Visualizad un diagrama de dispersión de las puntuaciones de películas de usuarios con correlaciones muy bajas (cercana a –1), nulas y muy altas (cercanas a 1). Determinad aproximadamente cuál sería la línea de ajuste en cada caso.

2. Relacionad las puntuaciones de películas en MovieLens con la edad: ¿se parecen más las puntuaciones de edades cercanas que las puntuaciones generales entre sí?

3. Completad la aplicación de recomendación de películas. La aplicación deberá pedir opinión sobre cinco películas al usuario y a partir de esos datos sugerirle películas que podrían gustarle.

4. Tomad una imagen con manchas de puntos —dibujadas con cualquier programa de dibujo— y obtened los grupos correspondientes utilizando *k*-medios y *c*-medios difuso.

5. Tomad una fotografía e intentad delimitar zonas similares utilizando *c*-medios difuso.

6. Agrupad las películas de la base de datos MovieLens por sus valoraciones. Analizad si hay correlación con el género de la película (fichero *u.genre*).

7. Utilizad el núcleo gaussiano en el programa de agrupamiento espectral al calcular la matriz de distancia. Analizad la influencia del parámetro en el número de grupos obtenido.

8. Probad con los diferentes valores del parámetro *affinity* en la llamada a *SpectralClustering* del programa 2.12 y comparad los resultados obtenidos.

9. Realizad una descomposición de PCA de una matriz de datos de tamaño $m \times n$ con $m = 10^4$ observaciones y $n = 4$ atributos en la que los dos primeros atributos siguen distribuciones normales $x_1 = N(0{,}5,\ 0{,}1)$, $x_2 = N(0{,}75,\ 0{,}2)$ y los dos últimos se definen como $x_3 = 0{,}3 \cdot x_1 + 1{,}2 \cdot x_2$ y $x_4 = \sqrt{x_1} + x_2$. Representad los valores propios resultantes en orden decreciente y seleccionad cuántos de ellos son necesarios para representar un 95 % de la varianza en los datos.

10. Apliquad un agrupamiento jerárquico a los resultados del análisis MDS con métrica euclídea (ejemplo 3.18) y representad los resultados obtenidos.

11. Modificad la implementación (programa 4.1) del Naïve Bayes (subapartado 4.2.1) aplicando la distribución gaussiana para poder tratar atributos numéricos y aplicadlo al conjunto de datos de las flores (subapartado 4.3.1).

12. Modificad la implementación (programa 4.3) del kNN (subapartado 4.3.1) para que vote teniendo en cuenta la ponderación de ejemplos en función de su cercanía al ejemplo de test.

13. Modificad la implementación (programa 4.3) del kNN (subapartado 4.3.1) para que utilice la distancia de Hamming y aplicadlo al conjunto de datos de las setas (subapartado 4.2.1).

14. Modificad la implementación (programa 4.4) del clasificador lineal basado en distancias (subapartado 4.3.2) para que utilice la distancia de Hamming y aplicadlo al conjunto de datos de las setas (subapartado 4.2.1).

15. Modificad la implementación (programa 4.5) del clasificador basado en *clustering* (subapartado 4.3.3) para que utilice la distancia de Hamming y aplicadlo al conjunto de datos de las setas (subapartado 4.2.1).

16. Modificad la implementación (programa 4.5) del clasificador basado en *clustering* (subapartado 4.3.3) cambiando el método de *clustering* (véase el subapartado 2.3).

17. Modificad la implementación (programa 4.6) de los árboles de decisión (subapartado 4.4.1) para que incluya el tratamiento de los puntos de corte de atributos numéricos y aplicadlo al conjunto de datos de las flores (subapartado 4.3.1).

18. Implementad el algoritmo ID3 (subapartado 4.4.1) partiendo del programa 4.6 y aplicadlo al conjunto de datos de las flores (subapartado 4.3.1).

19. Añadid la funcionalidad de la poda de ramas al programa realizado en el ejercicio anterior.

20. Diseñad las reglas débiles para tratar el problema de las flores (subapartado 4.3.1) y modificad el programa 4.7 para poder distinguir la clase iris-setosa de las demás.

21. Añadid la técnica del uno contra todos* (descrita en el subapartado 4.5.4) al programa 4.7 y aplicadlo al conjunto de datos de las flores (subapartado 4.3.1).

22. Partiendo de los dos ejercicios anteriores, mirad cuál es mejor sobre el conjunto de datos y si la diferencia es estadísticamente significativa.

23. Añadid el *kernel* polinómico —$(kx - zk2 + 1)d$ con d equivalente al grado del polinomio— al programa 4.8.

24. Modificad el programa 4.8 para comprobar los diferentes *kernels* disponibles en el módulo SVC.

25. Ampliad los programas 4.3 y 4.4 con la validación cruzada y comparad cuál de los dos va mejor para el conjunto de datos de las flores (subapartado 4.3.1) y si la diferencia es significativa utilizando el test de Student con un nivel de confianza del 95 %.

26. Repetid el ejercicio anterior utilizando el test no-paramétrico del Wilcoxon en lugar del paramétrico de Student. Comparad los resultados obtenidos.

27. Buscad ocho conjuntos de datos en repositorio de la UCI, añadidlos al de las flores y setas y utilizad el test de Friedman y su mejora para comparar los algoritmos: Naïve Bayes, kNN, lineal, árboles de decisión, AdaBoost y SVMs (códigos 4.1, 4.3, 4.4, 4.6, 4.7 y 4.8).

28. Utilizad el estadístico kappa dos a dos sobre los algoritmos: Naïve Bayes, kNN, lineal, árboles de decisión, AdaBoost y SVMs (códigos 4.1, 4.3, 4.4, 4.6, 4.7 y 4.8) para crear una matriz de valores sobre el tercer conjunto de datos del subapartado 4.1.1 y cread un dendrograma para comparar los diferentes algoritmos.

29. Resolved cada uno de los problemas de ejemplo de los métodos metaheurísticos con los restantes métodos. Comparad los resultados obtenidos y el tiempo de computación requerido en cada caso.

30. Utilizad una imagen en escala de grises a modo de mapa de alturas, donde los píxeles más claros representan puntos más elevados en un mapa. Encontrad el camino entre dos puntos del mapa que cumpla que el desnivel acumulado (total de subida y bajada) sea mínimo utilizando el algoritmo de la colonia de hormigas (subapartado 5.6). Nótese que previamente a aplicar el algoritmo hay que transformar el mapa de alturas en un grafo.

31. Escribid un programa que resuelva un sudoku utilizando el método de la búsqueda tabú visto en el subapartado 5.8. Para que la función objetivo cumpla las propiedades vistas en el subapartado 5.1 tiene que admitir sudokus con errores e ir eliminándolos progresivamente.

32. En un instituto con 4 grupos de alumnos y 5 profesores hay que encontrar una distribución de horarios compatible —ningún profesor puede dar clase en dos grupos a la vez y ningún grupo puede recibir dos clases a la vez. Cada grupo tiene 6 clases cada día durante cinco días a la semana, que se dividen en 10 asignaturas con 3 horas semanales cada una. En un mismo día no pueden tener más de una hora de la misma asignatura. Cada profesor imparte dos asignaturas en su totalidad, por lo que da un total de 24 horas de clase a la semana. Hay que obtener un horario válido utilizando alguno de los métodos vistos en este apartado.

33. Tomad los ejemplos de redes neuronales de clasificación de MNIST, tanto la versión FNN como la CNN. Comparad el rendimiento y el tiempo de entrenamiento empleado al utilizar abandono (*dropout*) frente a no utilizarlo.

34. Tomad el ejemplo de red neuronal recurrente que procesa las opiniones de IMDB. Cambiad el algoritmo de optimización y el número de palabras consideradas y estudiad los cambios en el rendimiento.

35. Programad una red neuronal convolucional para que procese y clasifique imágenes en color, por ejemplo del conjunto de datos CIFAR 10 (http://bit.ly/1QZAvsv).

36. Aplicad un clasificador CNN para imágenes de aviones, caras y motos utilizando la base de datos Caltech 101 (http://bit.ly/2GK5QA0).

37. Avanzado: en la página http://bit.ly/2DYohPX se describe cómo crear un sistema de aprendizaje por refuerzo en Keras utilizando un juego de un cochecito que debe salir de una hondonada. Intentad reproducir el código propuesto en la página.

Bibliografía

Bishop, C. M. (2006). *Pattern Recognition and Machine Learning*. EE. UU.: Springer.

Duda, R. O.; Hart, P. E.; Stork, D. G. (2001). *Pattern Classification* (2.ª ed.). USA: John Wiley and Sons, Inc.

Frank, A.; Asuncion, A. (2010). *UCI Machine Learning Repository* (en línea). USA: University of California, School of Information and Computer Science. [Fecha de consulta: 22 de enero de 2018]
<http://archive.ics.uci.edu/ml>

Géron, A. (2017). *Hands-On Machine Learning with Scikit-Learn and TensorFlow*. EE. UU.: O'Reilly Media.

Goodfellow, I.; Bengio, Y.; Courville, A. (2016). *Deep Learning*. EE. UU.: MIT Press.

Gulli, A.; Pal, S. (2017). *Deep Learning with Keras*. EE. UU.: Packt Publishing.

Mitchell, T. M. (1997). *Machine Learning*. EE. UU.: McGraw–Hill.

Segaran, T. (2007). *Programming Collective Intelligence*. EE. UU.: O'Reilly.

Shawe-Taylor, J.; Cristianini, N. (2004). *Kernel Methods for Pattern Analysis*. Reino Unido: Cambridge University Press.