MATHEMATICAL FOUNDATIONS
FOR
DATA ANALYSIS

JEFF M. PHILLIPS

2018

Preface

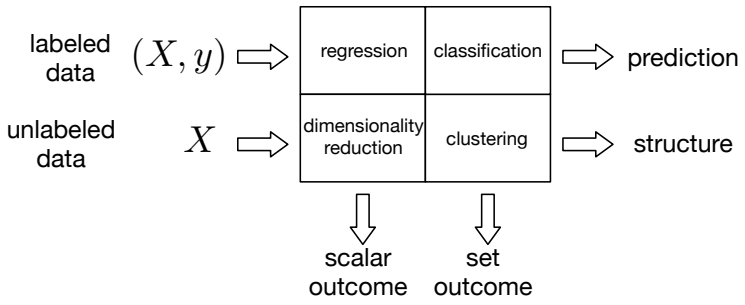
This book is meant for use with a self-contained course that introduces many basic principles and techniques needed for modern data analysis. In particular, it was constructed from material taught mainly in two courses. The first is an early undergraduate course which was designed to prepare students to succeed in rigorous Machine Learning and Data Mining courses. The second course is that advanced Data Mining course. It should be useful for any combination of such courses. The book introduces key conceptual tools which are often absent or brief in undergraduate curriculum, and for most students, helpful to see multiple times. On top of these, it introduces the generic versions of the most basic techniques that comprise the backbone of modern data analysis. And then it delves deeper in a few more advanced topics and techniques – still focusing on clear, intuitive, and lasting ideas, instead of specific details in the ever-evolving state-of-the-art.

Notation. Consistent, clear, and crisp mathematical notation is essential for intuitive learning. The domains which comprise modern data analysis (e.g., statistics, machine learning, algorithms) have matured somewhat separately with their own conventions for ways to write the same or similar concepts. Moreover, it is commonplace for researchers to adapt notation to best highlight ideas within specific papers. As such, much of the existing literature on the topics covered in this book is all over the place, inconsistent, and as a whole confusing. This text attempts to establish a common, simple, and consistent notation for these ideas, yet not veer too far from how concepts are consistently represented in research literature, and as they will be in more advanced courses. Indeed the most prevalent sources of confusion in earlier uses of this text in class have arisen around overloaded notation.

Interaction with other courses. It is recommended that students taking this class have calculus and a familiarity with programming and algorithms. They should have also taken some probability and/or linear algebra; but we also review key concepts in these areas, so as to keep the book more self-contained. Thus, it may be appropriate for students to take these classes before or concurrently. If appropriately planned for, it is the hope that this course could be taken at the undergraduate sophomore level so that more rigorous and advanced data analysis classes can already be taken during the junior year.

Although we touch on Bayesian Inference, we do not cover most of classical statistics; neither frequentist hypothesis testing or the similar Bayesian perspectives. Most universities have well-developed courses on these topics which while also very useful, provide a complimentary view of data analysis. Classical statistical modeling approaches are often essential when a practitioner needs to provide some modeling assumptions to harness maximum power from limited data. But in the era of big data this is not always necessary. Rather, the topics in this course provide tools for using some of the data to help choose the model.

Scope and topics. Vital concepts introduced include concentration of measure and PAC bounds, cross-validation, gradient descent, a variety of distances, principal component analysis, and graphs. These ideas are essential for modern data analysis, but not often taught in other introductory mathematics classes in a computer science or math department. Or if these concepts are taught, they are presented in a very different context.



We also survey basic techniques in supervised (regression and classification) and unsupervised (principal component analysis and clustering) learning. We make an effort to keep the presentation and concepts on these topics simple. We mainly stick to those which attempt to minimize sum of squared errors. We lead with classic but magical algorithms like Lloyd's algorithm for k-means, the power method for eigenvectors, and perceptron for linear classification. For many students (especially those in a computer science program), these are the first iterative, non-discrete algorithms they will have encountered. And sometimes the book ventures beyond these basics into concepts like regularization and lasso, locality sensitive hashing, multi-dimensional scaling, spectral clustering, and neural net basics. These can be sprinkled in, to allow courses to go deeper and more advanced as is suitable for the level of students.

On data. While this text is mainly focused on a mathematical preparation, what would data analysis be without data? As such we provide discussion on how to use these tools and techniques on actual data, with examples given in python. We choose python since it has increasingly many powerful libraries often with efficient backends in low level languages like C or Fortran. So for most data sets, this provides the proper interface for working with these tools.

But arguably more important than writing the code itself is a discussion on when and when-not to use techniques from the immense toolbox available. This is one of the main ongoing questions a data scientist must ask. And so, the text attempts to introduce the readers to this ongoing discussion.

Examples, Geometry, and Ethics. Three themes that this text highlights to try to aid in the understanding and broader comprehension of these fundamentals are examples, geometry, and ethical connotations. These are each offset in colored boxes.

Example: with focus on Simplicity

We try to provide numerous simple examples to demonstrate key concepts. We aim to be as simple as possible, and make data examples small, so they can be fully digested.

Geometry of Data and Proofs

Many of the ideas in this text are inherently geometric, and hence we attempt to provide many geometric illustrations which can illustrate what is going on. These boxes often go more in depth into what is going on, and include the most technical proofs.

Ethical Questions with Data Analysis

As data analysis nestles towards an abstract, automatic, but nebulous place within decision making everywhere, the surrounding ethical questions are becoming more important. We highlight various ethical questions which may arise in the course of using the analysis described in this text. We intentionally do not offer solutions, since there may be no single good answer to some of the dilemmas presented. Moreover, we believe the most important part of instilling positive ethics is to make sure analysts at least *think* about the consequences, which we hope these highlighting boxes achieves.

Thanks. I would like to thank gracious support from NSF in the form of grants CCF-1350888, IIS-1251019, ACI-1443046, CNS-1514520, CNS-1564287, and IIS-1816149, which have funded my cumulative research efforts during the writing of this text. I would also like to thank the University of Utah, as well as the Simons Institute for Theory of Computing, for providing excellent work environments while this text was written. And thanks to Natalie Cottrill for a careful reading and feedback.

This version. ... released online in December 2018, includes about 75 additional pages, and two new chapters (on Distances and on Graphs). Its goal was to expand the breadth and depth of the book. However, at this check point, these newly added topics may not be as polished as previous sections – this refinement will be the focus of the next update. Its not a final version, so please have patience and send thoughts, typos, suggestions!

Jeff M. Phillips
Salt Lake City, December 2018

Contents

1 Probability Review

Probability is a critical tool for modern data analysis. It arises in dealing with uncertainty, in randomized algorithms, and in Bayesian analysis. To understand any of these concepts correctly, it is paramount to have a solid and rigorous statistical foundation. Here we review some key definitions.

1.1 Sample Spaces

We define probability through set theory, starting with a *sample space* Ω. This represents the set of all things that might happen in the setting we consider. One such potential outcome $\omega \in \Omega$ is a *sample outcome*, it is an element of the set Ω. We are usually interested in an *event* that is a subset $A \subseteq \Omega$ of the sample space.

Example: Discrete Sample Space for a 6-Ssided Die

Consider rolling a single fair, 6-sided die. Then $\Omega = \{1, 2, 3, 4, 5, 6\}$. One roll may produce an outcome $\omega = 3$, rolling a 3. An event might be $A = \{1, 3, 5\}$, any odd number.
The probability of rolling an odd number is then $\textbf{Pr}(A) = |\{1, 3, 5\}|/|\{1, 2, 3, 4, 5, 6\}| = 1/2$.

A *random variable* $X : \Omega \to S$ is a **function** which maps from the sample space Ω to a domain S. In many cases $S \subseteq \mathbb{R}$, where $\mathbb{R}$ is the space of real numbers.

Example: Random Variables for a Fair Coin

Consider flipping a fair coin with $\Omega = \{H, T\}$. If I get a head H, then I get 1 point, and if I get a T, then I get 4 points. This describes the random variable X, defined $X(H) = 1$ and $X(T) = 4$.

The *probability* of an event $\textbf{Pr}(A)$ satisfies the following properties:

- $0 \leq \textbf{Pr}(A) \leq 1$ for any A,
- $\textbf{Pr}(\Omega) = 1$, and
- The probability of the union of disjoint events is equivalent to the sum of their individual probabilities. Formally, for any sequence $A_1, A_2, \ldots$ where for all $i \neq j$ that $A_i \cap A_j = \emptyset$, then

$$\textbf{Pr}\left(\bigcup_{i=1}^{\infty} A_i\right) = \sum_{i=1}^{\infty} \textbf{Pr}(A_i).$$

Example: Probability for a Biased Coin

Now consider flipping a biased coin with two possible events $\Omega = \{H, T\}$ (i.e., heads $H = A_1$ and tails $T = A_2$). The coin is biased so the probabilities $\textbf{Pr}(H) = 0.6$ and $\textbf{Pr}(T) = 0.4$ of these events are not equal. However we notice still $0 \leq \textbf{Pr}(T), \textbf{Pr}(H) \leq 1$, and that $\textbf{Pr}(\Omega) = \textbf{Pr}(H \cup T) = \textbf{Pr}(H) + \textbf{Pr}(T) = 0.6 + 0.4 = 1$. That is the sample space Ω is union of these two events, which cannot both occur (i.e., $H \cap T = \emptyset$) so they are disjoint. Thus Ω's probability can be written as the sum of the probability of those two events.

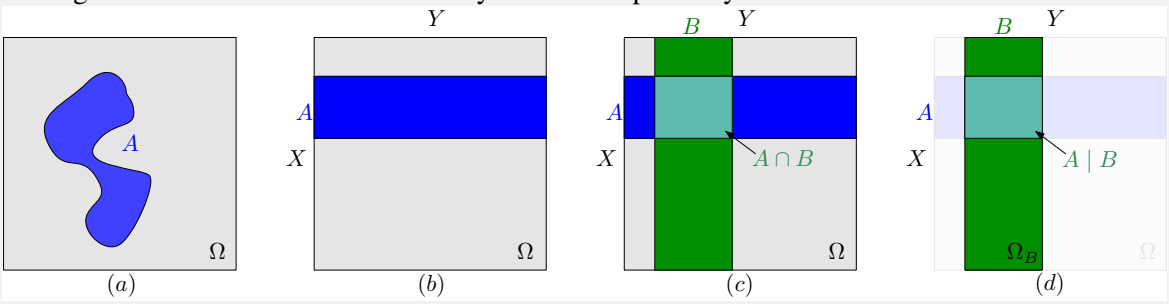
Sample spaces Ω can also be continuous, representing some quantity like water, time, or land mass which does not have discrete quantities. All of the above definitions hold for this setting.

Example: Continuous Sample Space

Assume you are riding a Swiss train that is always on time, but its departure is only specified to the minute (specifically, 1:37 pm). The true departure is then in the state space $\Omega = [1{:}37{:}00, 1{:}38{:}00)$. A continuous event may be $A = [1{:}37{:}00 - 1{:}37{:}40)$, the first 40 seconds of that minute.
Perhaps the train operators are risk averse, so $\mathbf{Pr}(A) = 0.80$. That indicates that 0.8 fraction of trains depart in the first $2/3$ of that minute (less than the 0.666 expected from a uniform distribution).

Geometry of Sample Spaces

It may be useful to generically imagine a sample space Ω as a square. Then, as shown in (a), an event A may be an arbitrary subset of this space.
When the sample space describes the joint sample space over two random variables X and Y, then it may be convenient to parameterize Ω so that the X value is represented along one side of the square, and the Y value along the other, as in (b). Then for an event $A \subset X$ which only pertains to the random variable X is represented as a rectangular strip defined by A's intersection with the domain of X.
If there is also an event $B \subset Y$ that only pertains to random variable Y, then another rectangular strip in the other direction defined by B's intersection with the domain of Y can be drawn as in (c). When these events are independent, then these strips intersect only in a another rectangle $A \cap B$. When X and Y are independent, then *all such strips*, defined by events $A \subset X$ and $B \subset Y$ intersect in a rectangle. If the events are not independent, then the associated picture will not look as clear, like in (a).
Given such independent events $A \subset X$ and $B \subset Y$, it is easy to see that $A \mid B$ can be realized, as in (d), with the rectangle $A \cap B$ restricted to the strip defined by B. Furthermore, imagining the area as being proportional to probability, it is also easy to see that $\mathbf{Pr}(A \mid B) = \mathbf{Pr}(A \cap B)/\mathbf{Pr}(B)$ since the strip B induces a new restricted sample space Ω_B, and an event only occurs in the strip-induced rectangle defined and further-restricted by A which is precisely $A \cap B$.



1.2 Conditional Probability and Independence

Now consider two events A and B. The *conditional probability* of A given B is written $\mathbf{Pr}(A \mid B)$, and can be interpreted as the probability of A, restricted to the setting where we know B is true. It is defined in simpler terms as $\mathbf{Pr}(A \mid B) = \frac{\mathbf{Pr}(A \cap B)}{\mathbf{Pr}(B)}$, that is the probability A and B are both true, divided by (normalized by) the probability B is true.

Two **events** A and B are *independent* of each other if and only if

$$\mathbf{Pr}(A \mid B) = \mathbf{Pr}(A).$$

Equivalently they are independent if and only if $\textbf{Pr}(B \mid A) = \textbf{Pr}(B)$ or $\textbf{Pr}(A \cap B) = \textbf{Pr}(A)\textbf{Pr}(B)$. By algebraic manipulation, it is not hard to see these are all equivalent properties. This implies that knowledge about B has no effect on the probability of A (and vice versa from A to B).

Example: Conditional Probability

Consider the two random variables. T is 1 if a test for cancer is positive, and 0 otherwise. Variable C is 1 if a patient has cancer, and 0 otherwise. The joint probability of the events is captured in the following table:

		cancer	no cancer
		$C = 1$	$C = 0$
tests positive for cancer	$T = 1$	0.1	0.02
tests negative for cancer	$T = 0$	0.05	0.83

Note that the sum of all cells (the joint sample space Ω) is 1. The conditional probability of having cancer, given a positive test is $\textbf{Pr}(C = 1 \mid T = 1) = \frac{0.1}{0.1+0.02} = 0.8333$. The probability of cancer (ignoring the test) is $\textbf{Pr}(C = 1) = 0.1 + 0.05 = 0.15$. Since $\textbf{Pr}(C = 1 \mid T = 1) \neq \textbf{Pr}(C = 1)$, then events $T = 1$ and $C = 1$ are not independent.

Two **random variables** X and Y are *independent* if and only if, for *all* possible events $A \subseteq \Omega_X$ and $B \subseteq \Omega_Y$ that A and B are independent: $\textbf{Pr}(A \cap B) = \textbf{Pr}(A)\textbf{Pr}(B)$.

1.3 Density Functions

Discrete random variables can often be defined through tables (as in the above cancer example). Or we can define a function $f_X(k)$ as the probability that random variable X is equal to k. For continuous random variables we need to be more careful: we will use calculus. We will next develop probability density functions (pdfs) and cumulative density functions (cdfs) for continuous random variables; the same constructions are sometimes useful for discrete random variables as well, which basically just replace a integral with a sum.

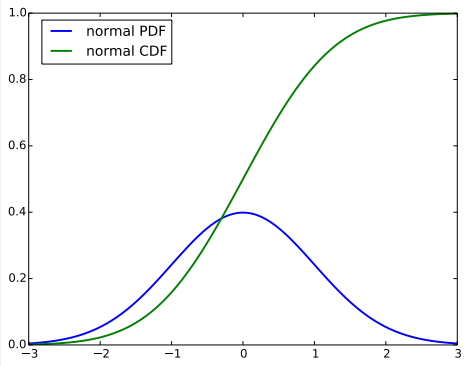
We consider a continuous sample space Ω, and a random variable X defined on that sample space. The probability density function of a random variable X is written f_X. It is defined with respect to any event A so that $\textbf{Pr}(X \in A) = \int_{\omega \in A} f_X(\omega) \mathrm{d}\omega$. The value $f_X(\omega)$ *is not equal to* $\textbf{Pr}(X = \omega)$ in general, since for continuous functions $\textbf{Pr}(X = \omega) = 0$ for any single value $\omega \in \Omega$. Yet, we can interpret f_X as a *likelihood* function; its value has no units, but they can be compared and larger ones are more likely.

Next we will define the *cumulative density function* $F_X(t)$; it is the probability that X takes on a value of t or smaller. Here it is typical to have $\Omega = \mathbb{R}$, the set of real numbers. Now define $F_X(t) = \int_{\omega=-\infty}^{t} f_X(\omega) \mathrm{d}\omega$.

We can also define a pdf in terms of a cdf as $f_X(\omega) = \frac{\mathrm{d}F_X(\omega)}{\mathrm{d}\omega}$.

Example: Normal Random Variable

A *normal* random variable X is a very common distribution to model noise. It has domain $\Omega = \mathbb{R}$. Its pdf is defined $f_X(\omega) = \frac{1}{\sqrt{2\pi}} \exp(-\omega^2/2) = \frac{1}{\sqrt{2\pi}} e^{-\omega^2/2}$, and its cdf has no closed form solution. We have plotted the cdf and pdf in the range $[-3, 3]$ where most of the mass lies:




```
import matplotlib as mpl
mpl.use('PDF')
import matplotlib.pyplot as plt
from scipy.stats import norm
import numpy as np
import math

mu = 0
variance = 1
sigma = math.sqrt(variance)
x = np.linspace(-3, 3, 201)

plt.plot(x, norm.pdf((x-mu)/sigma),linewidth=2.0, label='normal␣PDF')
plt.plot(x, norm.cdf((x-mu)/sigma),linewidth=2.0, label='normal␣CDF')
plt.legend(bbox_to_anchor=(.35,1))

plt.savefig('Gaussian.pdf', bbox_inches='tight')
```

1.4 Expected Value

The expected value of a random variable X in a domain Ω is a very important constant, basically a weighted average of Ω, weighted by the range of X. For a discrete random variable X it is defined as the sum over all outcomes ω in the sample space, or their value times their probability

$$\mathbf{E}[X] = \sum_{\omega \in \Omega} (\omega \cdot \mathbf{Pr}[X = \omega]).$$

For a continuous random variable X it is defined

$$\mathbf{E}[X] = \int_{\omega \in \Omega} \omega \cdot f_X(\omega) \mathrm{d}\omega.$$

Linearity of Expectation: An important property of expectation is that it is a linear operation. That means for two random variables X and Y we have $\mathbf{E}[X+Y] = \mathbf{E}[X] + \mathbf{E}[Y]$. For a scalar value α, we also $\mathbf{E}[\alpha X] = \alpha \mathbf{E}[X]$.

Example: Expectation

A fair die has a sample space of $\Omega = \{\omega_1 = 1, \omega_2 = 2, \omega_3 = 3, \omega_4 = 4, \omega_5 = 5, \omega_6 = 6\}$, and the probability of each outcome w_i is $\mathbf{Pr}[w_i] = 1/6$. The expected value able of a random variable D of the result of a roll of such a die is

$$\mathbf{E}[D] = \sum_{\omega_i \in \Omega} \omega_i \cdot \mathbf{Pr}[D = \omega_i] = 1 \cdot \frac{1}{6} + 2 \cdot \frac{1}{6} + 3 \cdot \frac{1}{6} + 4 \cdot \frac{1}{6} + 5 \cdot \frac{1}{6} + 6 \cdot \frac{1}{6} = \frac{21}{6} = 3.5$$

Example: Linearity of Expectation

Let H be the random variable of the height of a man in meters without shoes. Let the pdf f_H of H be a normal distribution with expected value $\mu = 1.755$m and with standard deviation 0.1m. Let S be the random variable of the height added by wearing a pair of shoes in centimeters (1 meter is 100 centimeters), its pdf is given by the following table:

$S = 1$	$S = 2$	$S = 3$	$S = 4$
0.1	0.1	0.5	0.3

Then the expected height of someone wearing shoes in centimeters is

$$\mathbf{E}[100 \cdot H + S] = 100 \cdot \mathbf{E}[H] + \mathbf{E}[S] = 100 \cdot 1.755 + (0.1 \cdot 1 + 0.1 \cdot 2 + 0.5 \cdot 3 + 0.3 \cdot 4) = 175.5 + 3 = 178.5$$

Note how the linearity of expectation allowed us to decompose the expression $100 \cdot H + S$ into its components, and take the expectation of each one individually. This trick is immensely powerful when analyzing complex scenarios with many factors.

1.5 Variance

The *variance* of a random variable X describes how spread out it is, with respect to its mean $\mathbf{E}[X]$. It is defined

$$\begin{aligned} \mathbf{Var}[X] &= \mathbf{E}[(X - \mathbf{E}[X])^2] \\ &= \mathbf{E}[X^2] - \mathbf{E}[X]^2. \end{aligned}$$

The equivalence of those two above common forms above uses that $\mathbf{E}[X]$ is a fixed scalar:

$$\mathbf{E}[(X - \mathbf{E}[X])^2] = \mathbf{E}[X^2 - 2X\mathbf{E}[X] + \mathbf{E}[X]^2] = \mathbf{E}[X^2] - 2\mathbf{E}[X]\mathbf{E}[X] + \mathbf{E}[X]^2 = \mathbf{E}[X^2] - \mathbf{E}[X]^2.$$

For any scalar $\alpha \in \mathbb{R}$, then $\mathbf{Var}[\alpha X] = \alpha^2 \mathbf{Var}[X]$.

Note that the variance does not have the same units as the random variable or the expectation, it is that unit squared. As such, we also often discuss the *standard deviation* $\sigma_X = \sqrt{\mathbf{Var}[X]}$. A low value of $\mathbf{Var}[X]$ or σ_X indicates that most values are close to the mean, while a large value indicates that the data has a higher probability of being further from the mean.

Example: Variance

Consider again the random variable S for height added by a shoe:

$S=1$	$S=2$	$S=3$	$S=4$
0.1	0.1	0.5	0.3

Its expected value is $\mathbf{E}[S] = 3$ (a fixed scalar), and its variance is

$$\begin{aligned}\mathbf{Var}[S] &= 0.1 \cdot (1-3)^2 + 0.1 \cdot (2-3)^2 + 0.5 \cdot (3-3)^2 + 0.3 \cdot (4-3)^2 \\ &= 0.1 \cdot (-2)^2 + 0.1 \cdot (-1)^2 + 0 + 0.3(1)^2 = 0.4 + 0.1 + 0.3 = 0.8.\end{aligned}$$

Then the standard deviation is $\sigma_S = \sqrt{0.8} \approx 0.894$.

The *covariance* of two random variables X and Y is defined $\mathbf{Cov}[X,Y] = \mathbf{E}[(X - \mathbf{E}[X])(Y - \mathbf{E}[Y])]$. It measures how much these random variables vary in accordance with each other; that is, if both are consistently away from the mean at the same time (in the same direction), then the covariance is high.

1.6 Joint, Marginal, and Conditional Distributions

We now extend some of these concepts to more than one random variable. Consider two random variables X and Y. Their *joint pdf* is defined $f_{X,Y} : \Omega_X \times \Omega_Y \to [0, \infty]$ where for discrete random variables this is defined by the probability $f_{X,Y}(x,y) = \mathbf{Pr}(X = x, Y = y)$. In this discrete case, the domain of $f_{X,Y}$ is restricted so $f_{X,Y} \in [0,1]$ and so $\sum_{x,y \in X \times Y} f_{X,Y}(x,y) = 1$, e.g., the sum of probabilities over the joint sample space is 1.

Similarly, when $\Omega_X = \Omega_Y = \mathbb{R}$, the *joint cdf* is defined $F_{X,Y}(x,y) = \mathbf{Pr}(X \leq x, Y \leq y)$. The *marginal cumulative distribution functions* of $F_{X,Y}$ are defined as $F_X(x) = \lim_{y \to \infty} F_{X,Y}(x,y)\mathrm{d}y$ and $F_Y(y) = \lim_{x \to \infty} F_{X,Y}(x,y)\mathrm{d}x$.

Similarly, when Y is discrete, the *marginal pdf* is defined $f_X(x) = \sum_{y \in \Omega_Y} f_{X,Y}(x,y) = \sum_{y \in \Omega_Y} \mathbf{Pr}(X = x, Y = y)$. When the random variables are continuous, we define $f_{X,Y}(x,y) = \frac{\mathrm{d}^2 F_{X,Y}(x,y)}{\mathrm{d}x\mathrm{d}y}$. And then the marginal pdf of X (when $\Omega_Y = \mathbb{R}$) is defined $f_X(x) = \int_{y=-\infty}^{\infty} f_{X,Y}(x,y)\mathrm{d}y$. Marginalizing removes the effect of a random variable (Y in the above definitions).

Now we can say random variables X and Y are independent if and only if $f_{X,Y}(x,y) = f_X(x) \cdot f_Y(y)$ for all x and y.

Then a *conditional distribution* of X given $Y = y$ is defined $f_{X|Y}(x \mid y) = f_{X,Y}(x,y)/f_Y(y)$ (given that $f_Y(y) \neq 0$).

Example: Marginal and Conditional Distributions

Consider someone who randomly chooses his pants and shirt every day (a to-remain-anonymous friend of the author's actually did this in college – all clothes were in a pile, clean or dirty; when the average article of clothing was too smelly, all were washed). Let P be a random variable for the color of pants, and S a random variable for the color of the shirt. Their joint probability is described by this table:

	S=green	S=red	S=blue
P=blue	0.3	0.1	0.2
P=white	0.05	0.2	0.15

Adding up along columns, the marginal distribution f_S for the color of the shirt is described by the following table:

S=green	S=red	S=blue
0.35	0.3	0.35

Isolating and renormalizing the middle "S=red" column, the conditional distribution $f_{P|S}(\cdot \mid S\text{= red})$ is described by the following table:

P=blue	P=white
$\frac{0.1}{0.3} = 0.3333$	$\frac{0.2}{0.3} = 0.6666$

Example: Gaussian Distribution

The Gaussian distribution is a d-variate distribution $\mathcal{G}_d : \mathbb{R}^d \to \mathbb{R}$ that generalizes the one-dimensional normal distribution. The definition of the symmetric version (we will generalize to non-trivial covariance later on) depends on a mean $\mu \in \mathbb{R}^d$ and a variance σ^2. For any vector $v \in \mathbb{R}^d$, it is defined

$$\mathcal{G}_d(v) = \frac{1}{\sigma^d \sqrt{2\pi^d}} \exp\left(-\|v - \mu\|^2/(2\sigma^2)\right).$$

For the 2-dimensional case where $v = (v_x, v_y)$ and $\mu = (\mu_x, \mu_y)$, then this is defined

$$\mathcal{G}_2(v) = \frac{1}{\sigma^2 \pi \sqrt{2}} \exp\left(-((v_x - \mu_x)^2 - (v_y - \mu_y)^2)/(2\sigma^2)\right).$$

A *magical* property about the Gaussian distribution is that all conditional versions of it are also Gaussian, of a lower dimension. For instance, in the two dimensional case $G_2(v_x \mid v_y = 1)$ is a 1-dimensional Gaussian, or a normal distribution. There are many other essential properties of the Gaussian that we will see throughout this text, including that it is invariant under all basis transformations and that it is the limiting distribution for central limit theorem bounds.

1.7 Bayes' Rule

Bayes' Rule is the key component in how to build likelihood functions, which when optimized are key to evaluating "models" based on data. Bayesian Reasoning is a much broader area that can go well beyond just finding the single "most optimal" model. This line of work, which this chapter will only introduce, reasons about the many possible models and can make predictions with this uncertainty taken into account.

Given two events M and D Bayes' Rule states

$$\mathbf{Pr}(M \mid D) = \frac{\mathbf{Pr}(D \mid M) \cdot \mathbf{Pr}(M)}{\mathbf{Pr}(D)}.$$

Mechanically, this provides an algebraic way to invert the direction of the conditioning of random variables, from (D given M) to (M given D). It assumes nothing about the independence of M and D (otherwise its pretty uninteresting). To derive this we use

$$\mathbf{Pr}(M \cap D) = \mathbf{Pr}(M \mid D)\mathbf{Pr}(D)$$

and also

$$\mathbf{Pr}(M \cap D) = \mathbf{Pr}(D \cap M) = \mathbf{Pr}(D \mid M)\mathbf{Pr}(M).$$

Combining these we obtain $\mathbf{Pr}(M \mid D)\mathbf{Pr}(D) = \mathbf{Pr}(D \mid M)\mathbf{Pr}(M)$, from which we can divide by $\mathbf{Pr}(D)$ to solve for $\mathbf{Pr}(M \mid D)$. So Bayes' Rule is uncontroversially true; any "frequentist vs. Bayesian" debate is about how to model data and perform analysis, not the specifics or correctness of this rule.

Example: Checking Bayes' Rule

Consider two events M and D with the following joint probability table:

	$M = 1$	$M = 0$
$D = 1$	0.25	0.5
$D = 0$	0.2	0.05

We can observe that indeed $\mathbf{Pr}(M \mid D) = \mathbf{Pr}(M \cap D)/\mathbf{Pr}(D) = \frac{0.25}{0.75} = \frac{1}{3}$, which is equal to

$$\frac{\mathbf{Pr}(D \mid M)\mathbf{Pr}(M)}{\mathbf{Pr}(D)} = \frac{\frac{.25}{.2+.25}(.2 + .25)}{.25 + .5} = \frac{.25}{.75} = \frac{1}{3}.$$

But Bayes' rule is not very interesting in the above example. In that example, it is actually *more* complicated to calculate the right side of Bayes' rule than it is the left side.

Example: Cracked Windshield

Consider you bought a new car and its windshield was cracked, the event W. If the car was assembled at one of three factories A, B or C, you would like to know which factory was the most likely point of origin.

Assume that in Utah 50% of cars are from factory A (that is $\mathbf{Pr}(A) = 0.5$) and 30% are from factory B ($\mathbf{Pr}(B) = 0.3$), and 20% are from factory C ($\mathbf{Pr}(C) = 0.2$).

Then you look up statistics online, and find the following rates of cracked windshields for each factory – apparently this is a problem! In factory A, only 1% are cracked, in factory B 10% are cracked, and in factory C 2% are cracked. That is $\mathbf{Pr}(W \mid A) = 0.01$, $\mathbf{Pr}(W \mid B) = 0.1$ and $\mathbf{Pr}(W \mid C) = 0.02$.

We can now calculate the probability the car came from each factory:

- $\mathbf{Pr}(A \mid W) = \mathbf{Pr}(W \mid A) \cdot \mathbf{Pr}(A)/\mathbf{Pr}(W) = 0.01 \cdot 0.5/\mathbf{Pr}(W) = 0.005/\mathbf{Pr}(W)$.
- $\mathbf{Pr}(B \mid W) = \mathbf{Pr}(W \mid B) \cdot \mathbf{Pr}(B)/\mathbf{Pr}(W) = 0.1 \cdot 0.3/\mathbf{Pr}(W) = 0.03/\mathbf{Pr}(W)$.
- $\mathbf{Pr}(C \mid W) = \mathbf{Pr}(W \mid C) \cdot \mathbf{Pr}(C)/\mathbf{Pr}(W) = 0.02 \cdot 0.2/\mathbf{Pr}(W) = 0.004/\mathbf{Pr}(W)$.

We did not calculate $\mathbf{Pr}(W)$, but it must be the same for all factory events, so to find the highest probability factory we can ignore it. The probability $\mathbf{Pr}(B \mid W) = 0.03/\mathbf{Pr}(W)$ is the largest, and B is the most likely factory.

1.7.1 Model Given Data

In data analysis, M represents a 'model' and D as 'data.' Then $\mathbf{Pr}(M \mid D)$ is interpreted as the probability of model M given that we have observed D. A *maximum a posteriori* (or MAP) estimate is the model $M \in \Omega_M$ that maximizes $\mathbf{Pr}(M \mid D)$. That is[1]

$$M^* = \operatorname*{argmax}_{M \in \Omega_M} \mathbf{Pr}(M \mid D) = \operatorname*{argmax}_{M \in \Omega_M} \frac{\mathbf{Pr}(D \mid M)\mathbf{Pr}(M)}{\mathbf{Pr}(D)} = \operatorname*{argmax}_{M \in \Omega_M} \mathbf{Pr}(D \mid M)\mathbf{Pr}(M).$$

Thus, by using Bayes' Rule, we can maximize $\mathbf{Pr}(M \mid D)$ using $\mathbf{Pr}(M)$ and $\mathbf{Pr}(M \mid D)$. We do not need $\mathbf{Pr}(D)$ since our data is given to us and fixed for all models.

In some settings we may also ignore $\mathbf{Pr}(M)$, as we may assume all possible models are equally likely. This is not always the case, and we will come back to this. In this setting we just need to calculate $\mathbf{Pr}(D \mid M)$. This function $L(M) = \mathbf{Pr}(D \mid M)$ is called the *likelihood* of model M.

So what is a 'model' and what is 'data?' A *model* is usually a simple pattern from which we think data is generated, but then observed with some noise. Examples:

- The model M is a single point in $\mathbb{R}^d$; the data is a set of points in $\mathbb{R}^d$ near M.
- **linear regression:** The model M is a line in $\mathbb{R}^2$; the data is a set of points such that for each x-coordinate, the y-coordinate is the value of the line at that x-coordinate with some added noise in the y-value.
- **clustering:** The model M is a small set of points in $\mathbb{R}^d$; the data is a large set of points in $\mathbb{R}^d$, where each point is near one of the points in M.

[1] Consider a set S, and a function $f : S \to \mathbb{R}$. The $\max_{s \in S} f(s)$ returns the *value* $f(s^*)$ for some element $s^* \in S$ which results in the largest valued $f(s)$. The $\operatorname{argmax}_{s \in S} f(s)$ returns the *element* $s^* \in S$ which results in the largest valued $f(s)$; if this is not unique, it may return any such $s^* \in S$.

- **PCA:** The model M is a k-dimensional subspace in $\mathbb{R}^d$ (for $k \ll d$); the data is a set of points in $\mathbb{R}^d$, where each point is near M.
- **linear classification:** The model M is a halfspace in $\mathbb{R}^d$; the data is a set of labeled points (with labels $+$ or $-$), so the $+$ points are mostly in M, and the $-$ points are mainly not in M.

Log-likelihoods. An important trick used in understanding the likelihood, and in finding the MAP model M^*, is to take the logarithm of the posterior. Since the logarithm operator $\log(\cdot)$ is monotonically increasing on positive values, and all probabilities (and more generally pdf values) are non-negative (treat $\log(0)$ as $-\infty$), then $\text{argmax}_{M\in\Omega_M} \textbf{Pr}(M \mid D) = \text{argmax}_{M\in\Omega_M} \log(\textbf{Pr}(M \mid D))$. It is commonly applied on only the likelihood function $L(M)$, and $\log(L(M))$ is called the log-likelihood. Since $\log(a\cdot b) = \log(a)+\log(b)$, this is useful in transforming definitions of probabilities, which are often written as products $\Pi_{i=1}^k P_i$ into sums $\log(\Pi_{i=1}^k P_i) = \sum_{i=1}^k \log(P_i)$, which are easier to manipulated algebraically.

Moreover, the base of the log is unimportant in model selection using the MAP estimate because $\log_{b_1}(x) = \log_{b_2}(x)/\log_{b_2}(b_1)$, and so $1/\log_{b_2}(b_1)$ is a coefficient that does not affect the choice of M^*. The same is true for the *maximum likelihood estimate (MLE):* $M^* = \text{argmax}_{M\in\Omega_M} L(M)$.

Example: Gaussian MLE

Let the data D be a set of points in $\mathbb{R}^1$: $\{1, 3, 12, 5, 9\}$. Let Ω_M be $\mathbb{R}$ so that the model is parametrized by a point $M \in \mathbb{R}$. If we assume that each data point is observed with independent Gaussian noise (with $\sigma = 2$, so its pdf is described as $g(x) = \frac{1}{\sqrt{8\pi}} \exp(-\frac{1}{8}(M-x)^2)$. Then

$$\textbf{Pr}(D \mid M) = \prod_{x\in D} g(x) = \prod_{x\in D} \left(\frac{1}{\sqrt{8\pi}} \exp(-\frac{1}{8}(M-x)^2)\right).$$

Recall that we can take the product $\Pi_{x\in D} g(x)$ since we assume independence of $x \in D$! To find $M^* = \text{argmax}_M \textbf{Pr}(D \mid M)$ is equivalent to $\text{argmax}_M \ln(\textbf{Pr}(D \mid M))$, the *log-likelihood* which is

$$\ln(\textbf{Pr}(D \mid M)) = \ln\left(\prod_{x\in D}\left(\frac{1}{\sqrt{8\pi}} \exp(-\frac{1}{8}(M-x)^2)\right)\right) = \sum_{x\in D}\left(-\frac{1}{8}(M-x)^2\right) + |D| \ln(\frac{1}{\sqrt{8\pi}}).$$

We can ignore the last term in the sum since it does not depend on M. The first term is maximized when $\sum_{x\in D}(M-x)^2$ is minimized, which occurs precisely as $\textbf{E}[D] = \frac{1}{|D|}\sum_{x\in D} x$, the mean of the data set D. That is, the maximum likelihood model is exactly the mean of the data D, and is quite easy to calculate.

1.8 Bayesian Inference

Bayesian inference focuses on a simplified version of Bayes's Rule:

$$\textbf{Pr}(M \mid D) \propto \textbf{Pr}(D \mid M) \cdot \textbf{Pr}(M).$$

The symbol $\propto$ means *proportional to*; that is there is a fixed (but possibly unknown) constant factor c multiplied on the right (in this case $c = 1/\textbf{Pr}(D)$) to make them equal: $\textbf{Pr}(M \mid D) = c\cdot\textbf{Pr}(D \mid M)\cdot\textbf{Pr}(M)$.

However, we may want to use continuous random variables, so then strictly using probability **Pr** at a single point is not always correct. So we can replace each of these with pdfs

$$\underset{\text{posterior}}{p(M \mid D)} \quad \propto \quad \underset{\text{likelihood}}{f(D \mid M)} \quad \cdot \quad \underset{\text{prior}}{\pi(M)}.$$

Each of these terms have common names. As above, the conditional probability or pdf $\mathbf{Pr}(D \mid M) \propto f(D \mid M)$ is called the *likelihood*; it evaluates the effectiveness of the model M using the observed data D. The probability or pdf of the model $\mathbf{Pr}(M) \propto \pi(M)$ is called the *prior*; it is the assumption about the relative propensity of a model M, before or independent of the observed data. And the left hand side $\mathbf{Pr}(M \mid D) \propto p(M \mid D)$ is called the *posterior*; it is the combined evaluation of the model that incorporates how well the observed data fits the model and the independent assumptions about the model.

Again it is common to be in a situation where, given a fixed model M, it is possible to calculate the likelihood $f(D \mid M)$. And again, the goal is to be able to compute $p(M \mid D)$, as this allows us to evaluate potential models M, given the data we have seen D.

The main difference is a careful analysis of $\pi(M)$, the prior – which is not necessarily assumed uniform or "flat". The prior allows us to encode our assumptions.

Example: Average Height

Lets estimate the height H of a typical Data University student. We can construct a data set $D = \{x_1, \ldots, x_n\}$ by measuring the height of everyone in this class in inches. There may be error in the measurement, and we are an incomplete set, so we don't entirely trust the data.
So we introduce a prior $\pi(M)$. Consider we read that the average height of a full grown person is $\mu_M = 66$ inches, with a standard deviation of $\sigma = 6$ inches. So we assume

$$\pi(M) = N(66, 6) = \frac{1}{\sqrt{\pi 72}} \exp(-(\mu_M - 66)^2/(2 \cdot 6^2)),$$

is normally distributed around 66 inches.
Now, given this knowledge we adjust the MLE example from last subsection using this prior.

- *What if our MLE estimate without the prior (e.g. $\frac{1}{|D|}\sum_{x \in D} x$) provides a value of 55?*
  The data is very far from the prior! Usually this means something is wrong. We could find $\text{argmax}_M\, p(M \mid D)$ using this information, but that may give us an estimate of say 20 (that does not seem correct). A more likely explanation is a mistake somewhere: probably we measured in feet instead of inches!

Another vestige of Bayesian inference is that we not only can calculate the maximum likelihood model M^*, but we can also provide a posterior value for any model! This value is not an absolute probability (its not normalized, and regardless it may be of measure 0), but it is powerful in other ways:

- We can say (under our model assumptions, which are now clearly stated) that one model M_1 is twice as likely as another M_2, if $p(M_1 \mid D)/p(M_2 \mid D) = 2$.
- We can define a range of parameter values (with more work and under our model assumptions) that likely contains the true model.
- We can now use more than one model for prediction of a value. Given a new data point x' we may want to map it onto our model as $M(x')$, or assign it a score of fit. Instead of doing this for just one "best" model M^*, we can take a weighted average of all models, weighted by their posterior; this is "marginalization."

Weight for Prior. So how important is the prior? In the average height example, it will turn out to be worth only $(1/9)$th of one student's measurement. But we can give it more weight.

Example: Weighted Prior for Height

Lets continue the example about the height of an average Data University student, and assume (as in the MLE example) the data is generated independently from a model M with Gaussian noise with $\sigma = 2$. Thus the likelihood of the model, given the data is

$$f(D \mid M) = \prod_{x \in D} g(x) = \prod_{x \in D} \left(\frac{1}{\sqrt{8\pi}} \exp(-\frac{1}{8}(\mu_M - x)^2) \right).$$

Now using that the prior of the model is $\pi(M) = \frac{1}{\sqrt{\pi 72}} \exp(-(\mu_M - 66)^2/72)$, the posterior is given by

$$p(M \mid D) \propto f(D \mid M) \cdot \frac{1}{\sqrt{\pi 72}} \exp(-(\mu_M - 66)^2/72).$$

It is again easier to work with the log-posterior which is monotonic with the posterior, using some unspecified constant C (which can be effectively ignored):

$$\begin{aligned} \ln(p(M \mid D)) &\propto \ln(f(D \mid M)) + \ln(\pi(M)) + C \\ &\propto \sum_{x \in D} \left(-\frac{1}{8}(\mu_M - x)^2) \right) - \frac{1}{72}(\mu_M - 66)^2 + C \\ &\propto -\sum_{x \in D} 9(\mu_M - x)^2 + (\mu_M - 66)^2 + C \end{aligned}$$

So the maximum likelihood estimator occurs at the average of 66 along with 9 copies of the student data.

Why is student measurement data worth so much more?

We assume the standard deviation of the measurement error is 2, where as we assumed that the standard deviation of the full population was 6. In other words, our measurements had variance $2^2 = 4$, and the population had variance $6^2 = 36$ (technically, this is best to interpret as the variance when adapted to various subpopulations, e.g., Data University students): that is 9 times as much.

If instead we assumed that the standard deviation of our prior is 0.1, with variance 0.01, then this is 400 times smaller than our class measurement error variance. If we were to redo the above calculations with this smaller variance, we would find that this assumption weights the prior 400 times the effect of each student measurement in the MLE.

In fact, a much smaller variance on the prior is probably more realistic since national estimates on height are probably drawn from a very large sample. And its important to keep in mind that we are estimating the *average height* of a population, not the *height of a single person* randomly drawn from the population. In the next topic (T3) we will see how averages of random variables have much smaller variance – are much more concentrated – than individual random variables.

So what happens with more data?

Lets say, this class gets really popular, and next year $1{,}000$ or $10{,}000$ students sign up! Then again the student data is overall worth more than the prior data. So with any prior, if we get enough data, it no longer becomes important. But with a small amount of data, it can have a large influence on our model.

Exercises

Q1.1: Consider the probability table below for the random variables X and Y. One entry is missing, but you should be able to derive it. Then calculate the following values.

1. **Pr**$(X = 3 \cap Y = 2)$
2. **Pr**$(Y = 1)$
3. **Pr**$(X = 2 \cap Y = 1)$
4. **Pr**$(X = 2 \mid Y = 1)$

	$X = 1$	$X = 2$	$X = 3$
$Y = 1$	0.25	0.1	0.15
$Y = 2$	0.1	0.2	??

Q1.2: An "adventurous" athlete has the following running routine every morning: He takes a bus to a random stop, then hitches a ride, and then runs all the way home. The bus, described by a random variable B, has four stops where the stops are at a distance of 1, 3, 4, and 7 miles from his house – he chooses each with probability $1/4$. Then the random hitchhiking takes him further from his house with a uniform distribution between -1 and 4 miles; that is it is represented as a random variable H with pdf described

$$f(H = x) = \begin{cases} 1/5 & \text{if } x \in [-1, 4] \\ 0 & \text{if } x \notin [-1, 4]. \end{cases}$$

What is the expected distance he runs each morning (all the way home)?

Q1.3: Consider rolling two fair die D_1 and D_2; each has a probability space of $\Omega = \{1, 2, 3, 4, 5, 6\}$ which each value equally likely. What is the probability that D_1 has a larger value than D_2? What is the expected value of the sum of the two die?

Q1.4: Let X be a random variable with a uniform distribution over $[0, 2]$; its pdf is described

$$f(X = x) = \begin{cases} 1/2 & \text{if } x \in [0, 2] \\ 0 & \text{if } x \notin [0, 2]. \end{cases}$$

What is the probability that $f(X = 1)$?

Q1.5: Use python to plot the pdf and cdf of the Laplace distribution ($f(x) = \frac{1}{2}\exp(-|x|)$) for values of x in the range $[-3, 3]$. The function `scipy.stats.laplace` may be useful.

Q1.6: Consider the random variables X and Y described by the joint probability table.

	$X = 1$	$X = 2$	$X = 3$
$Y = 1$	0.10	0.05	0.10
$Y = 2$	0.30	0.25	0.20

Derive the following values

1. **Pr**$(X = 1)$
2. **Pr**$(X = 2 \cap Y = 1)$
3. **Pr**$(X = 3 \mid Y = 2)$

Compute the following probability distributions.

4. What is the marginal distribution for X?
5. What is the conditional probability for Y, given that $X = 2$?

Answer the following question about the joint distribution.

6. Are random variables X and Y independent?
7. Is $\mathbf{Pr}(X = 1)$ independent of $\mathbf{Pr}(Y = 1)$?

Q1.7: Consider two models M_1 and M_2, where from prior knowledge we believe that $\mathbf{Pr}(M_1) = 0.25$ and $\mathbf{Pr}(M_2) = 0.75$. We then observe a data set D. Given each model we assess the likelihood of seeing that data given the model as $\mathbf{Pr}(D \mid M_1) = 0.5$ and $\mathbf{Pr}(D \mid M_2) = 0.01$. Now that we have the data, which model is has a higher probability of being correct?

Q1.8: Assume I observe 3 data points x_1, x_2, and x_3 drawn independently from an unknown distribution. Given a model M, I can calculate the likelihood for each data point as $\mathbf{Pr}(x_1 \mid M) = 0.5$, $\mathbf{Pr}(x_2 \mid M) = 0.1$, and $\mathbf{Pr}(x_3 \mid M) = 0.2$. What is the likelihood of seeing all of these data points, given the model M: $\mathbf{Pr}(x_1, x_2, x_3 \mid M)$?

Q1.9: Consider a data set D with 10 data points $\{-1, 6, 0, 2, -1, 7, 7, 8, 4, -2\}$. We want to find a model for M from a restricted sample space $\Omega = \{0, 2, 4\}$. Assume the data has Laplace noise defined, so from a model M a data point's probability distribution is described $f(x) = \frac{1}{4}\exp(-|M - x|/2)$. Also assume we have a prior assumption on the models so that $\mathbf{Pr}(M = 0) = 0.25$, $\mathbf{Pr}(M = 2) = 0.35$, and $\mathbf{Pr}(M = 4) = 0.4$. Assuming all data points in D are independent, which model is most likely.

2 Convergence and Sampling

This topic will overview a variety of extremely powerful analysis results that span statistics, estimation theorem, and big data. It provides a framework to think about how to aggregate more and more data to get better and better estimates. It will cover the *Central Limit Theorem* (CLT), Chernoff-Hoeffding bounds, Probably Approximately Correct (PAC) algorithms, as well as analysis of importance sampling techniques which improve the concentration of random samples.

2.1 Sampling and Estimation

Most data analysis starts with some data set; we will call this data set P. It will be composed of a set of n data points $P = \{p_1, p_2, \ldots, p_n\}$.

But underlying this data is almost always a very powerful assumption, that this data comes iid from a fixed, but usually unknown pdf, call this f. Lets unpack this: What does "iid" mean: Identically and Independently Distributed. The "identically" means each data point was drawn from the same f. The "independently" means that the first points have no bearing on the value of the next point.

Example: Polling

Consider a poll of $n = 1000$ likely voters in an upcoming election. If we assume each polled person is chosen iid, then we can use this to understand something about the underlying distribution f, for instance the distribution of all likely voters.
More generally, f could represent the outcome of a process, whether that is a randomized algorithm, a noisy sensing methodology, or the common behavior of a species of animals. In each of these cases, we essentially "poll" the process (algorithm, measurement, thorough observation) having it provide a sample, and repeat many times over.

Here we will talk about estimating the mean of f. To discuss this, we will now introduce a random variable $X \sim f$; a hypothetical new data point. The *mean* of f is the expected value of X: $\mathbf{E}[X]$.

We will estimate the mean of f using the *sample mean*, defined $\bar{P} = \frac{1}{n}\sum_{i=1}^{n} p_i$. The following diagram represents this common process: from a unknown process f, we consider n iid random variables $\{X_i\}$ corresponding to a set of n independent observations $\{p_i\}$, and take their average $\bar{P} = \frac{1}{n}\sum_{i=1}^{n} p_i$ to estimate the mean of f.

$$\bar{P} \underset{\frac{1}{n}\sum}{=} \{p_i\} \underset{\text{realize}}{\leftarrow} \{X_i\} \underset{\text{iid}}{\sim} f$$

Central Limit Theorem. The central limit theorem is about how well the sample mean approximates the true mean. But to discuss the sample mean $\bar{P}$ (which is a fixed value) we need to discuss random variables $\{X_1, X_2, \ldots, X_n\}$, and their mean $\bar{X} = \frac{1}{n}\sum_{i=1}^{n} X_i$. Note that again $\bar{X}$ is a random variable. If we are to draw a new iid data set P' and calculate a new sample mean $\bar{P}'$ it will likely not be exactly the same as $\bar{P}$; however, the distribution of where this $\bar{P}'$ is likely to be, is precisely $\bar{X}$. Arguably, this distribution is more important than $\bar{P}$ itself.

There are many formal variants of the central limit theorem, but the basic form is as follows:

Central Limit Theorem: Consider n iid random variables $X_1, X_2, \ldots, X_n$, where each $X_i \sim f$ for any fixed distribution f with mean μ and bounded variance σ^2. Then $\bar{X} = \frac{1}{n}\sum_{i=1}^{n} X_i$ converges to the normal distribution with mean $\mu = \mathbf{E}[X_i]$ and variance σ^2/n.

Lets highlight some important consequences:

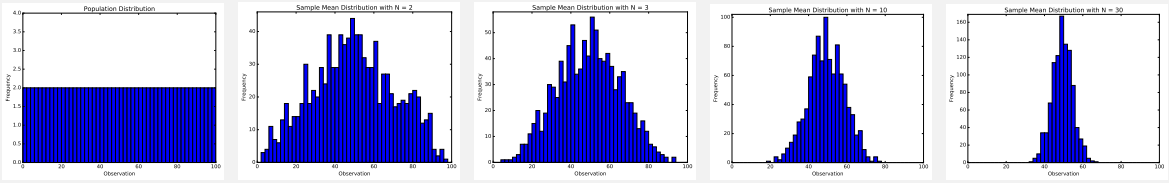
- For any f (that is not too crazy, since σ^2 is not infinite), then $\bar{X}$ looks like a normal distribution.
- The mean of the normal distribution, which is the expected value of $\bar{X}$ satisfies $\mathbf{E}[\bar{X}] = \mu$, the mean of f. This implies that we can use $\bar{X}$ (and then also $\bar{P}$) as a guess for μ.
- As n gets larger (we have more data points) then the variance of $\bar{X}$ (our estimator) decreases. So keeping in mind that although $\bar{X}$ has the right expected value it also has some error, this error is decreasing as n increases.

```
# adapted from:  https://github.com/mattnedrich/CentralLimitTheoremDemo
import random
import matplotlib as mpl
mpl.use('PDF')
import matplotlib.pyplot as plt

def plot_distribution(distribution, file, title, bin_min, bin_max, num_bins):
    bin_size = (bin_max - bin_min) / num_bins
    manual_bins = range(bin_min, bin_max + bin_size, bin_size)
    [n, bins, patches] = plt.hist(distribution, bins = manual_bins)
    plt.title(title)
    plt.xlim(bin_min, bin_max)
    plt.ylim(0, max(n) + 2)
    plt.ylabel("Frequency")
    plt.xlabel("Observation")
    plt.savefig(file, bbox_inches='tight')
    plt.clf()
    plt.cla()


minbin = 0
maxbin = 100
numbins = 50
nTrials = 1000

def create_uniform_sample_distribution():
    return range(maxbin)
sampleDistribution = create_uniform_sample_distribution()

# Plot the original population distribution
plot_distribution(sampleDistribution, 'output/SampleDistribution.pdf',
  "Population Distribution", minbin, maxbin, numbins)

# Plot a sampling distribution for values of N = 2, 3, 10, and 30
n_vals = [2, 3, 10, 30]
for N in n_vals:
  means = []
  for j in range(nTrials):
    sampleSum = 0;
    for i in range(N):
      sampleSum += random.choice(sampleDistribution)
```

```
    means.append(float(sampleSum)/ float(N))

title = "Sample Mean Distribution with N = %s" % N
file = "output/CLT-demo-%s.pdf" % N
plot_distribution(means, file, title, minbin, maxbin, numbins)
```

Example: Central Limit Theorem

Consider f as a uniform distribution over $[0, 100]$. If we create n samples $\{p_1, \ldots, p_n\}$ and their mean $\bar{P}$, then repeat this 1000 times, we can plot the output in histograms:



We see that starting at $n = 2$, the distributions look vaguely looks normal (in the technical sense of a normal distribution), and that their standard deviations narrow as n increases.

Remaining Mysteries. There should still be at least a few aspects of this not clear yet: (1) What does "convergence" mean? (2) How do we formalize or talk about this notion of error? (3) What does this say about our data $\bar{P}$?

First, *convergence* refers to what happens as some parameter increases, in this case n. As the number of data points increase, as n "goes to infinity" then the above statement ($\bar{X}$ looks like a normal distribution) becomes more and more true. For small n, the distribution may not quite look like a normal, it may be more bumpy, maybe even multi-modal. The statistical definitions of convergence are varied, and we will not go into them here, we will instead replace it with more useful phrasing in explaining aspects (2) and (3).

Second, the error now has two components. We cannot simply say that $\bar{P}$ is at most some distance ε from μ. Something crazy might have happened (the sample is random after all). And it is not useful to try to write the probability that $\bar{P} = \mu$; for equality in continuous distributions, this probability is indeed 0. But we can combine these notions. We can say the distance between $\bar{P}$ and μ is more than ε, with probability at most δ. This is called "probably approximately correct" or PAC.

Third, we want to generate some sort of PAC bound (which is far more useful than "$\bar{X}$ looks kind of like a normal distribution"). Whereas a frequentist may be happy with a confidence interval and a Bayesian a normal posterior, these two options are not directly available since again, $\bar{X}$ is not exactly a normal. So we will discuss some very common *concentration of measure* tools. These don't exactly capture the shape of the normal distribution, but provide upper bounds for its tails, and will allow us to state PAC bounds.

2.2 Probably Approximately Correct (PAC)

We will introduce shortly the three most common concentration of measure bounds, which provide increasingly strong bounds on the tails of distributions, but require more and more information about the underlying distribution f. Each provides a PAC bound of the following form:

$$\mathbf{Pr}[|\bar{X} - \mathbf{E}[\bar{X}]| \geq \varepsilon] \leq \delta.$$

That is, the probability that $\bar{X}$ (which is some random variable, often a sum of iid random variables) is further than ε to its expected value (which is μ, the expected value of f where $X_i \sim f$), is at most δ. Note we do not try to say this probability is *exactly* δ, this is often too hard. In practice there are a variety of tools, and a user may try each one, and see which ones gives the best bound.

It is useful to think of ε as the *error tolerance* and δ as the *probability of failure*, i.e., failure meaning that we exceed the error tolerance. However, often these bounds will allow us to write the required sample size n in terms of ε and δ. This allows us to trade these two terms off for any fixed known n; we can gaurantee a smaller error tolerance if we are willing to allow more probability of failure, and vice-versa.

2.3 Concentration of Measure

We will formally describe these bounds, and give some intuition of why they are true (but not full proofs). But what will be the most important is what they imply. If you just know the distance of the expectation from the minimal value, you can get a very weak bound. If you know the variance of the data, you can get a stronger bound. If you know that the distribution f has a small and bounded range, then you can make the probability of failure (the δ in PAC bounds) very very small.

Markov Inequality. Let X be a random variable such that $X \geq 0$, that is it cannot take on negative values. Then for any parameter $\alpha > 0$

$$\mathbf{Pr}[X > \alpha] \leq \frac{\mathbf{E}[X]}{\alpha}.$$

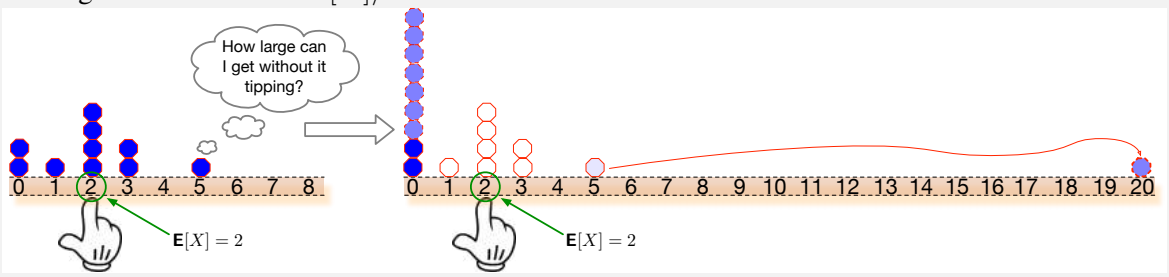
Note this is a PAC bound with $\varepsilon = \alpha - \mathbf{E}[X]$ and $\delta = \mathbf{E}[X]/\alpha$, or we can rephrase this bound as follows: $\mathbf{Pr}[X - \mathbf{E}[X] > \varepsilon] \leq \delta = \mathbf{E}[X]/(\varepsilon + \mathbf{E}[X])$.

Geometry of the Markov Inequality

Consider balancing the pdf of some random variable X on your finger at $\mathbf{E}[X]$, like a waitress balances a tray. If your finger is not under a value μ so $\mathbf{E}[X] = \mu$, then the pdf (and the waitress's tray) will tip, and fall in the direction of μ – the "center of mass."
Now for some amount of probability α, how large can we increase its location so we retain $\mathbf{E}[X] = \mu$. For each part of the pdf we increase, we must decrease some in proportion. However, by the assumption $X \geq 0$, the pdf must not be positive below 0. In the limit of this, we can set $\mathbf{Pr}[X = 0] = 1 - \alpha$, and then move the remaining α probability as large as possible, to a location δ so $\mathbf{E}[X] = \mu$. That is

$$\mathbf{E}[X] = 0 \cdot \mathbf{Pr}[X = 0] + \delta \cdot \mathbf{Pr}[X = \delta] = 0 \cdot (1 - \alpha) + \delta \cdot \alpha = \delta \cdot \alpha.$$

Solving for δ we find $\delta = \mathbf{E}[X]/\alpha$.

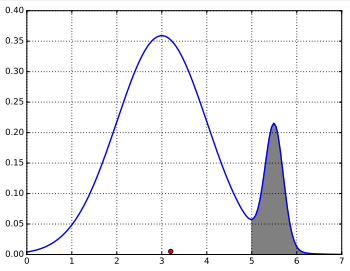
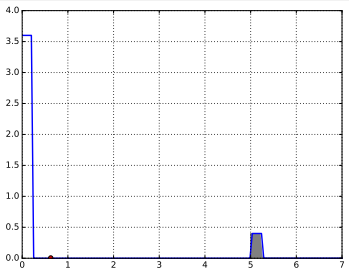


Imagine having 10 α-balls each representing $\alpha = 1/10$th of the probability mass. As in the figure, if these represent a distribution with $\mathbf{E}[X] = 2$ and this must stay fixed, how far can one ball increase if all others balls must take a value at least 0? One ball can move to 20.

If we instead know that $X \geq b$ for some constant b (instead of $X \geq 0$), then we state more generally $\mathbf{Pr}[X > \alpha] \leq (\mathbf{E}[X] - b)/(\alpha - b)$.

Example: Markov Inequality

Consider the pdf f drawn in blue in the following figures, with $\mathbf{E}[X]$ for $X \sim f$ marked as a red dot. The probability that X is greater than 5 (e.g. $\mathbf{Pr}[X \geq 5]$) is the shaded area.

Notice that in both cases that $\mathbf{Pr}[X \geq 5]$ is about 0.1. This is the quantity we want to bound by above by δ. But since $\mathbf{E}[X]$ is much larger in the first case (about 2.25), then the bound $\delta = \mathbf{E}[X]/\alpha$ is much larger, about 0.45. In the second case, $\mathbf{E}[X]$ is much smaller (about 0.6) so we get a much better bound of $\delta = 0.12$.

Example: Markov Inequality and Coffee

Let C be a random variable describing the number of liters of coffee the faculty at Data University will drink in a week. Say we know $\mathbf{E}[C] = 20$.
We use the Markov Inequality to bound the probability that the coffee consumed will be more than 50 liters as

$$\mathbf{Pr}[C \geq 50] \leq \frac{\mathbf{E}[C]}{50} = \frac{20}{50} = 0.4$$

Hence, based on the expected value alone, we can bound the with probability less than 0.4, the faculty at DU will drink less than 50 liters of coffee.

Chebyshev Inequality. Now let X be a random variable where we know $\mathbf{Var}[X]$, and $\mathbf{E}[X]$. Then for any parameter $\varepsilon > 0$

$$\mathbf{Pr}[|X - \mathbf{E}[X]| \geq \varepsilon] \leq \frac{\mathbf{Var}[X]}{\varepsilon^2}.$$

Again, this clearly is a PAC bound with $\delta = \mathbf{Var}[X]/\varepsilon^2$. This bound is typically stronger than the Markov one since δ decreases quadratically in ε instead of linearly.

Example: Chebyshev Inequality and Coffee

Again let C be a random variable for the liters of coffee that faculty at Data University will drink in a week with $\mathbf{E}[C] = 20$. If we also know that the variance is not too large, specifically $\mathbf{Var}[C] = 9$ (liters squared), then we can apply the Chebyshev inequality to get an improved bound.

$$\mathbf{Pr}[C \geq 50] \leq \mathbf{Pr}[|C - \mathbf{E}[C]| \geq 30] \leq \frac{\mathbf{Var}[C]}{30^2} = \frac{9}{900} = 0.01$$

That is, by using the expectation ($\mathbf{E}[C] = 20$) and variance ($\mathbf{Var}[C] = 9$) we can reduce the probability of exceeding 50 liters to at most probability 0.01.
Note that in the first inequality we convert from a one-sided expression $C \geq 50$ to a two-sided expression $|C - \mathbf{E}[C]| \geq 30$ (that is either $C - \mathbf{E}[C] \geq 30$ or $\mathbf{E}[C] - C \geq 30$). This is a bit wasteful, and stricter one-sided variants of Chebyshev inequality exist; we will not discuss these here in an effort for simplicity.

Recall that for an average of random variables $\bar{X} = (X_1 + X_2 + \ldots + X_n)/n$, where the X_is are iid, and have variance σ^2, then $\mathbf{Var}[\bar{X}] = \sigma^2/n$. Hence

$$\mathbf{Pr}[|\bar{X} - \mathbf{E}[X_i]| \geq \varepsilon] \leq \frac{\sigma^2}{n\varepsilon^2}.$$

Consider now that we have input parameters ε and δ, our desired error tolerance and probability of failure. If we can draw $X_i \sim f$ (iid) for an unknown f (with known expected value and variance σ), then we can solve for how large n needs to be: $n = \sigma^2/(\varepsilon^2\delta)$.

Since $\mathbf{E}[\bar{X}] = \mathbf{E}[X_i]$ for iid random variables $X_1, X_2, \ldots, X_n$, there is not a similar meaningfully-different extension for the Markov inequality.

Chernoff-Hoeffding Inequality. Following the above extension of the Chebyshev inequality, we can consider a set of n iid random variables $X_1, X_2, \ldots, X_n$ where $\bar{X} = \frac{1}{n}\sum_{i=1}^{n} X_i$. Now assume we know that each X_i lies in a bounded domain $[b, t]$, and let $\Delta = t - b$. Then for any parameter $\varepsilon > 0$

$$\mathbf{Pr}[|\bar{X} - \mathbf{E}[\bar{X}]| > \varepsilon] \leq 2\exp\left(\frac{-2\varepsilon^2 n}{\Delta^2}\right).$$

Again this is a PAC bound, now with $\delta = 2\exp(-2\varepsilon^2 n/\Delta^2)$. For a desired error tolerance ε and failure probability δ, we can set $n = (\Delta^2/(2\varepsilon^2))\ln(2/\delta)$. Note that this has a similar relationship with ε as the Chebyshev bound, but the dependence of n on δ is exponentially less for this bound.

Example: Chernoff-Hoeffding and Dice

Consider rolling a fair die 120 times, and recording how many times a 3 is returned. Let T be the random variable for the total number of 3s rolled. Each roll is a 3 with probability $1/6$, so the expect number of 3s is $\mathbf{E}[T] = 20$. We would like to answer what is the probability that more than 40 rolls return a 3.

To do so, we analyze $n = 120$ iid random variables $T_1, T_2, \ldots, T_n$ associated with each roll. In particular $T_i = 1$ if the ith roll is a 3 and is 0 otherwise. Thus $\mathbf{E}[T_i] = 1/6$ for each roll. Using $\bar{T} = T/n = \frac{1}{n}\sum_{i=1}^{n} T_i$ and noting that $\mathbf{Pr}[T \geq 40] = \mathbf{Pr}[\bar{T} \geq 1/3]$, we can now apply our Chernoff-Hoeffding bound as

$$\mathbf{Pr}[T \geq 40] \leq \mathbf{Pr}\left[\left|\bar{T} - \mathbf{E}[T_i]\right| \geq \frac{1}{6}\right] \leq 2\exp\left(\frac{-2(1/6)^2 \cdot 120}{1^2}\right) = 2\exp(-20/3) \leq 0.0026$$

So we can say that less than 3 out of 1000 times of running these 120 rolls should we see more than 40 returned 3s.

In comparison, we could have also applied a Chebyshev inequality. The variance of a single random variable T_i is $\mathbf{Var}[T_i] \leq 5/36$, and hence $\mathbf{Var}[T] = n \cdot \mathbf{Var}[T_i] = 50/3$. Hence we can bound

$$\mathbf{Pr}[T \geq 40] \leq \mathbf{Pr}[|T - 20| \geq 20] \leq (50/3)/20^2 \leq 0.042$$

That is, using the Chebyshev Inequality we were only able to claim that this event should occur at most 42 times out of 1000 trials.

Finally, we note that in both of these analysis we only seek to bound the probability that the number of rolls of 3 exceeds some threshold (≥ 40), whereas the inequality we used bounded the absolute value of the deviation from the mean. That is, our goal was one-way, and the inequality was a stronger two-way bound. Indeed these results can be improved by roughly a factor of 2 by using similar one-way inequalities that we do not formally state here.

Relating this all back to the Gaussian distribution in the CLT, the Chebyshev bound only uses the variance information about the Gaussian, but the Chernoff-Hoeffding bound uses all of the "moments": this allows the probability of failure to decay exponentially.

These are the most basic and common PAC concentration of measure bounds, but are by no means exhaustive.

Example: Concentration on Samples from the Uniform Distribution

Consider a random variable $X \sim f$ where $f(x) = \{\frac{1}{2}$ if $x \in [0,2]$ and 0 otherwise.$\}$, i.e, the uniform distribution on $[0,2]$. We know $\mathbf{E}[X] = 1$ and $\mathbf{Var}[X] = \frac{1}{3}$.

- Using the Markov Inequality, we can say $\mathbf{Pr}[X > 1.5] \leq 1/(1.5) \approx 0.6666$ and $\mathbf{Pr}[X > 3] \leq 1/3 \approx 0.33333$.
  or $\mathbf{Pr}[X - \mu > 0.5] \leq \frac{2}{3}$ and $\mathbf{Pr}[X - \mu > 2] \leq \frac{1}{3}$.
- Using the Chebyshev Inequality, we can say that $\mathbf{Pr}[|X - \mu| > 0.5] \leq (1/3)/0.5^2 = \frac{4}{3}$ (which is meaningless). But $\mathbf{Pr}[|X - \mu| > 2] \leq (1/3)/(2^2) = \frac{1}{12} \approx 0.08333$.

Now consider a set of $n = 100$ random variables $X_1, X_2, \ldots, X_n$ all drawn iid from the same pdf f as above. Now we can examine the random variable $\bar{X} = \frac{1}{n}\sum_{i=1}^{n} X_i$. We know that $\mu_n = \mathbf{E}[\bar{X}] = \mu$ and that $\sigma_n^2 = \mathbf{Var}[\bar{X}] = \sigma^2/n = 1/(3n) = 1/300$.

- Using the Chebyshev Inequality, we can say that $\mathbf{Pr}[|\bar{X} - \mu| > 0.5] \leq \sigma_n^2/(0.5)^2 = \frac{1}{75} \approx 0.01333$, and $\mathbf{Pr}[|\bar{X} - \mu| > 2] \leq \sigma_n^2/2^2 = \frac{1}{1200} \approx 0.0008333$.
- Using the Chernoff-Hoeffding bound, we can say that $\mathbf{Pr}[|\bar{X} - \mu| > 0.5] \leq 2\exp(-2(0.5)^2 n/\Delta^2) = 2\exp(-100/8) \approx 0.0000074533$, and $\mathbf{Pr}[|\bar{X} - \mu| > 2] \leq 2\exp(-2(2)^2 n/\Delta^2) = 2\exp(-200) \approx 2.76 \cdot 10^{-87}$.

2.3.1 Union Bound and Examples

The *union bound* is the "Robin" to Chernoff-Hoeffding's "Batman." It is a simple helper bound that allows Chernoff-Hoeffding to be applied to much larger and more complex situations[1]. It may appear that is a crude and overly simplistic bound, but it can usually only be significantly improved if fairly special structure is available for the specific class of problems considered.

Union bound. Consider s possibly *dependent* random events $Z_1, \ldots, Z_s$. The probability that all events occur is at least

$$1 - \sum_{j=1}^{s} (1 - \mathbf{Pr}[Z_j]).$$

That is, all events are true if no event is not true.

[1] I suppose these situations are the villains in this analogy, like "Riddler," "Joker," and "Poison Ivy." The union bound can also aid other concentration inequalities like Chebyshev, which I suppose is like "Catwoman."

Example: Union Bound and Dice

Returning to the example of rolling a fair die $n = 120$ times, and bounding the probability that a 3 was returned more than 40 times. Lets now consider the probability that *no number* was returned more than 40 times. Each number corresponds with a random event $Z_1, Z_2, Z_3, Z_4, Z_5, Z_6$, of that number occurring at most 40 times. These events are not independent, but nonetheless we can apply the union bound.

Using our Chebyshev Inequality result that $\mathbf{Pr}[Z_3] \geq 1 - 0.042 = 0.958$ we can apply this symmetrically to all Z_j. Then by the union bound, we have that the probability all numbers occur less than 40 times on $n = 120$ independent rolls is least

$$1 - \sum_{j=1}^{6}(1 - 0.958) = 0.748$$

Alternatively, we can use the result from the Chernoff-Hoeffding bound that $\mathbf{Pr}[Z_j] \geq 1 - 0.0026 = 0.9974$ inside the union bound to obtain that all numbers occur no more than 40 times with probability at least

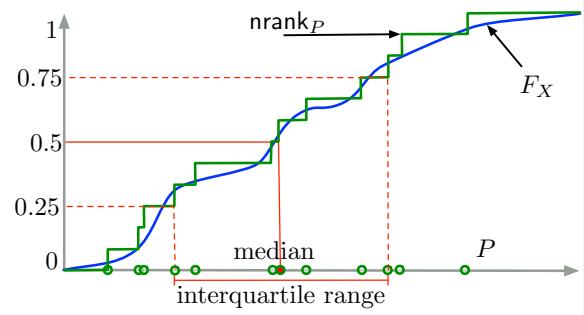
$$1 - \sum_{j=1}^{6}(1 - 0.9974) = 0.9844$$

So this joint event (of all numbers occurring at most 40 times) occurs more than 98% of the time, but using Chebyshev, we were unable to claim it happened more than 75% of the time.

Quantile Estimation. An important use case of concentration inequalities and the union bound is to estimate distributions. For random variable X, let f_X describe its pdf and F_X its cdf. Suppose now we can draw iid samples $P = \{p_1, p_2, \ldots, p_n\}$ from f_X, then we can use these n data points to estimate the cdf F_X. To understand this approximation, recall that $F_X(t)$ is the probability that random variable X takes a value at most t. For any choice of t we can estimate this from P as $\mathsf{nrank}_P(t) = |\{p_i \in P \mid p_i \leq t\}|/n$; i.e., as the fraction of samples with value at most t. The quantity $\mathsf{nrank}_P(t)$ is the *normalized rank* of P at value t, and the value of t for which $\mathsf{nrank}_P(t) \leq \phi < \mathsf{nrank}_P(t + \eta)$, for any $\eta > 0$, is known as the *ϕ-quantile* of P. For instance, when $\mathsf{nrank}_P(t) = 0.5$, it is the 0.5-quantile and thus the *median* of the dataset. And the interval $[t_1, t_2]$ such that t_1 is the 0.25-quantile and t_2 is the 0.75-quantile is known as the *interquartile range*. We can similarly define the ϕ-quantile (and hence the median and interquartile range) for a distribution f_X as the value t such that $F_X(t) = \phi$.

Example: CDF and Normalized Rank

The following illustration shows a cdf F_X (in blue) and its approximation via normalized rank nrank_P on a sampled point set P (in green). The median of the P and its interquartile range are marked (in red).



For a given, value t we can quantify how well $\mathsf{nrank}_P(t)$ approximates $F_X(t)$ using a Chernoff-Hoeffding bound. For a given t, for each sample p_i, describe a random variable Y_i which is 1 if $p_i \leq t$ and 0 otherwise. Observe that $\mathbf{E}[Y_i] = F_X(t)$, since it is precisely the probability that a random variable $X_i \sim f_X$ (representing data point p_i, not yet realized) is less than t. Moreover, the random variable for $\mathsf{nrank}_P(t)$ on a future iid sample P is precisely $\bar{Y} = \frac{1}{n}\sum_{i=1}^{n} Y_i$. Hence we can provide a PAC bound, on the probability (δ) of achieving more than ε error for any given t, as

$$\mathbf{Pr}[|\bar{Y} - F_X(t)| \geq \varepsilon] \leq 2\exp\left(\frac{-2\varepsilon^2 n}{1^2}\right) = \delta.$$

If we have a desired error ε (e.g., $\varepsilon = 0.05$) and probability of failure δ (e.g, $\delta = 0.01$), we can solve for how many sample are required: these values are satisfied with

$$n \geq \frac{1}{2\varepsilon^2}\ln\frac{2}{\delta}.$$

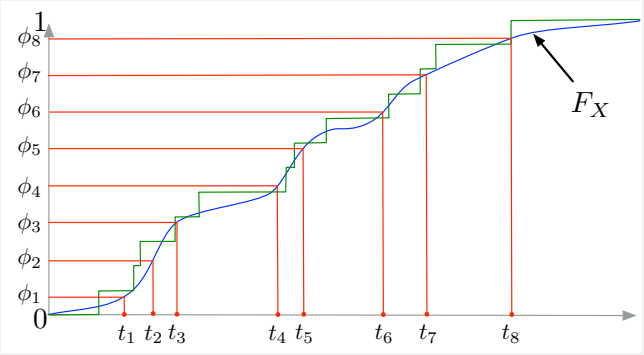
Approximating *all* quantiles. However, the above analysis only works for a single value of t. What if we wanted to show a similar analysis simultaneously for *all* values of t; that is, with how many samples n, can we then ensure that with probability at least $1-\delta$, for all values of t we will have $|\mathsf{nrank}_P(t) - F_X(t)| \leq \varepsilon$?

We will apply the union bound, but, there is another challenge we face: there are an infinite number of possible values t for which we want this bound to hold! We address this by splitting the error component into two pieces ε_1 and ε_2 so $\varepsilon = \varepsilon_1 + \varepsilon_2$; we can set $\varepsilon_1 = \varepsilon_2 = \varepsilon/2$. Now we consider $1/\varepsilon_1$ different quantiles $\{\phi_1, \phi_2, \ldots, \phi_{1/\varepsilon_1}\}$ where $\phi_j = j \cdot \varepsilon_1 + \varepsilon_1/2$. This divides up the probability space (i.e., the interval $[0,1]$) into $1/\varepsilon_1 + 1$ parts, so the gap between the boundaries of the parts is ε_1.

We will guarantee that each of these ϕ_j-quantiles are ε_2-approximated. Each ϕ_j corresponds with a t_j so t_j is the ϕ_j-quantile of f_X. We do not need to know what the precise values of the t_j are; however we do know that $t_j \leq t_{j+1}$, so they grow monotonically. In particular, this implies that for any t so $t_j \leq t \leq t_{j+1}$, then it must be that $F_X(t_j) \leq F_X(t) \leq F_X(t_{j+1})$; hence if both $F_X(t_j)$ and $F_X(t_{j+1})$ are within ε_2 of their estimated value, then $F_X(t)$ must be within $\varepsilon_2 + \varepsilon_1 = \varepsilon$ of its estimated value.

Example: Set of ϕ_j-quantiles to approximate

The illustration shows the set $\{\phi_1, \phi_2, \ldots, \phi_8\}$ of quantile points overlayed on the cdf F_X. With $\varepsilon_2 = 1/8$, these occur at values $\phi_1 = 1/16$, $\phi_2 = 3/16$, $\phi_3 = 5/16$, ..., and evenly divide the y-axis. The corresponding values $\{t_1, t_2, \ldots, t_8\}$ non-uniformly divide the x-axis. But as long as any consecutive pair t_j and t_{j+1} is approximated, because the cdf F_X and nrank are monotonic, then all intermediate values $t \in [t_j, t_{j+1}]$ are also approximated.



So what remains is to show that for all $t \in T = \{t_1, \ldots, t_{1/\varepsilon_1}\}$ that a random variable $\bar{Y}(t_j)$ for $\mathsf{nrank}_P(t_j)$ satisfies $\mathbf{Pr}[|\bar{Y}(t_j) - F_X(t_j)| \leq \varepsilon_2] \leq \delta$. By the above Chernoff-Hoeffding bound, this holds with probability $1 - \delta' = 1 - 2\exp(-2(\varepsilon_2)^2 n)$ for each t_j. Applying the union bound over these $s = 1/\varepsilon_1$ events we find that they all hold with probability

$$1 - \sum_{j=1}^{s} 2\exp(-2(\varepsilon_2)^2 n) = 1 - \frac{1}{\varepsilon_1} 2\exp(-2(\varepsilon_2)^2 n).$$

Setting, $\varepsilon_1 = \varepsilon_2 = \varepsilon/2$ we have that the probability that a sample of size n will provide a nrank_P function so for any t we have $|\mathsf{nrank}_P(t) - F_X(t)| \leq \varepsilon$, is at least $1 - \frac{4}{\varepsilon}\exp(-\frac{1}{2}\varepsilon^2 n)$. Setting the probability of failure $\frac{4}{\varepsilon}\exp(-\varepsilon^2 n/2) = \delta$, we can solve for n to see that we get at most ε error with probability at least $1 - \delta$ using $n = \frac{2}{\varepsilon^2}\ln(\frac{4}{\varepsilon\delta})$ samples.

2.4 Importance Sampling

Many important convergence bounds deal with approximating the mean of a distribution. When the samples are all uniform from an unknown distribution, then the above bounds (and their generalizations) are the best way to understand convergence, up to small factors. In particular, this is true when the only access to the data is a new iid sample.

However, when more control can be made over how the sample is generated, than in some cases simple changes can dramatically improve the accuracy of the estimates. The key idea is called importance sampling, and has the following principle: *sample larger variance data points more frequently, but in the estimate weight them inverse to the sampling probability.*

Sample average of weights. Consider a discrete and very large set $A = \{a_1, a_2, \ldots, a_n\}$ where each element a_i has an associated weight $w(a_i)$. Our goal will be to estimate the expected (or average) weight

$$\bar{w} = \mathbf{E}[w(a_i)] = \frac{1}{n}\sum_{a_i \in A} w(a_i).$$

This set may be so large that we do not want to explicitly compute the sum, or perhaps soliciting the weight is expensive (e.g., like conducting a customer survey). So we want to avoid doing this calculation over all items. Rather, we sample k items iid $\{a'_1, a'_2, \ldots, a'_k\}$ (each a'_j uniformly and independently chosen from A so some may be taken more than once), solicit the weight w'_j of each a'_j, and estimate the average weight as

$$\hat{w} = \frac{1}{k}\sum_{j=1}^{k} w(a'_j).$$

How accurately does $\hat{w}$ approximate $\bar{w}$? If all of the weights are roughly uniform or well-distributed in $[0, 1]$, then we can apply a Chernoff-Hoeffding bound so

$$\mathbf{Pr}[|\bar{w} - \hat{w}| \geq \varepsilon] \leq 2\exp(-2\varepsilon^2 k).$$

So with probability at least $1 - \delta$, we have no more than ε error using $k = \frac{1}{2\varepsilon^2}\ln\frac{2}{\delta}$ samples.

However, if we do not have a bound on the weights relative to the estimate, then this provides a poor approximation. For instance, if $\bar{w} = 0.0001$ since most of the weights are very small, then an $(\varepsilon = 0.01)$-approximation may not be very useful. Or similarly, if most weights are near 1, and so $\bar{w} = 2$, but there are a few outlier weights with value $\Delta = 1{,}000$, then the Chernoff-Hoeffding bound only states

$$\mathbf{Pr}[|\bar{w} - \hat{w}| \geq \varepsilon] \leq 2\exp\left(\frac{-2\varepsilon^2 k}{\Delta^2}\right) = \delta.$$

So this implies that instead we require $k = \frac{\Delta^2}{2\varepsilon^2}\ln(2/\delta)$, which is a factor $\Delta^2 = 1{,}000{,}000$ more samples than before!

Importance sampling. We slightly recast the problem assuming a bit more information. There is large set of items $A = \{a_1, a_2, \ldots, a_n\}$, and on sampling an item a'_j, its weight $w(a'_j)$ is revealed. Our goal is to estimate $\bar{w} = \frac{1}{n}\sum_{a_i \in A} w(a_i)$. We can treat W_i as a random variable for each the $w(a_i)$, then $\bar{w} = \frac{1}{n}\sum_{i=1}^{n} W_i$ is also a random variable. In this setting, we also know for each a_i (before sampling) some information about the range of its weight. That is, we have a bounded range $[0, \psi_i]$ so $0 \leq w(a_i) \leq \psi_i$[2]. This upper bound serves as an *importance* ψ_i for each a_i. Let $\Psi = \sum_{i=1}^{n}\psi_i$ be the sum of all importances.

As alluded to, the solution is the following two-step procedure called *importance sampling*.

Importance Sampling:

1. Sample k items $\{a'_1, a'_2, \ldots, a'_k\}$ independently from A, proportional to their importance ψ_i.
2. Estimate $w_I = \frac{1}{k}\sum_{j=1}^{k}\left(\frac{\Psi}{n\psi_j}\cdot w(a'_j)\right)$; where $\Psi = \sum_{i=1}^{n}\psi_i$.

We will first show that importance sampling provides an unbiased estimate; that is $\mathbf{E}[w_I] = \bar{w}$. Define a random variable Z_j to be the value $\frac{\Psi}{n\psi_j}w(a'_j)$. By linearity of expectation and the independence of the samples, $\mathbf{E}[w_I] = \frac{1}{k}\sum_{j=1}^{k}\mathbf{E}[Z_j] = \mathbf{E}[Z_j]$. Sampling proportional to ψ_i, means object a_i is chosen with probability ψ_i/Ψ. Summing over all elements,

$$\mathbf{E}[w_I] = \mathbf{E}[Z_j] = \sum_{i=1}^{n}\mathbf{Pr}[a'_j = a_i]\cdot\frac{\Psi}{n\psi_i}w(a_i) = \sum_{i=1}^{n}\frac{\psi_i}{\Psi}\cdot\frac{\Psi}{n\psi_i}w(a_i) = \frac{1}{n}\sum_{i=1}^{n}w(a_i) = \bar{w}.$$

[2]We can more generally allow $w(a_i)$ to have any bounded range $[L_i, U_i]$. In this case we set $\psi_i = U_i - L_i$, add $\frac{1}{n}\sum_{i=1}^{n}L_i$ to the final estimate, and if a_i is the jth sample let $w(a'_j) = w(a_i) - L_i$.

Note that this worked for any choice of ψ_i. Indeed, uniform sampling (which implicitly has $\psi_i = 1$ for each a_i) also is an unbiased estimator. The real power of importance sampling is that it improves the concentration of the estimates.

Improved concentration. To improve the concentration, the critical step is to analyze the range of each estimator $\frac{\Psi}{n\psi_j} \cdot w(a'_j)$. Since we have that $w(a'_j) \in [0, \psi_j]$, then as a result

$$\frac{\Psi}{n\psi_j} \cdot w(a'_j) \in \frac{\Psi}{n\psi_j} \cdot [0, \psi_j] = [0, \frac{\Psi}{n}].$$

Now applying a Chernoff-Hoeffding bound, we can upper bound the probability that w_I has more than ε error with respect to $\bar{w}$.

$$\mathbf{Pr}[|\bar{w} - \hat{w}| \geq \varepsilon] \leq 2\exp\left(\frac{-2\varepsilon^2 k}{(\Psi/n)^2}\right) = \delta$$

Fixing the allowed error ε and probability of failure δ we can solve for the number of samples required as

$$k = \frac{(\Psi/n)^2}{2\varepsilon^2}\ln(2/\delta).$$

Now instead of depending quadratically on the *largest* possible value Δ as in the uniform sampling, this now depends quadratically on the *average* upper bound on all values Ψ/n. In other words, with importance sampling, we reduce the sample complexity from depending on the maximum importance $\max_i \psi_i$ to on the *average importance* Ψ/n.

Example: Company Salary Estimation

Consider a company with 10,000 employees and we want to estimate the average salary. However the salaries are very imbalanced, the CEO makes way more than the typical employee. Say we know the CEO makes at most 2 million a year, but the other 99 employees make at most 50 thousand a year.

Using just uniform sampling of $k = 100$ employees we can apply a Chenoff-Hoeffding bound to estimate the average salary $\hat{w}$ from the true average salary $\bar{w}$ with error more than 10,000 with probability

$$\mathbf{Pr}[|\hat{w} - \bar{w}| \geq 10{,}000] \leq 2\exp\left(\frac{-2(10{,}000)^2 \cdot 100}{(2\text{ million})^2}\right) = 2\exp\left(\frac{-1}{200}\right) \approx 1.99$$

This is a useless bound, since the probability is greater than 1. If we increase the error tolerance to half a million, we still only get a good estimate with probability 0.42. The problem hinges on if we sample the CEO, our estimate is too high. If we do not, then the estimate is too low.

Now using importance sampling, the CEO gets an importance of 2 million, and the other employees all get an importance of 50 thousand. The average importance is now $\Psi/n = 51{,}950$, and we can bound the probability the new estimate w_I is more than 10,000 from $\bar{w}$ is at most

$$\mathbf{Pr}[|w_I - \bar{w}| \geq 10{,}000] \leq 2\exp\left(\frac{-2(10{,}000)^2 \cdot 100}{(51{,}950)^2}\right) \leq 0.0013.$$

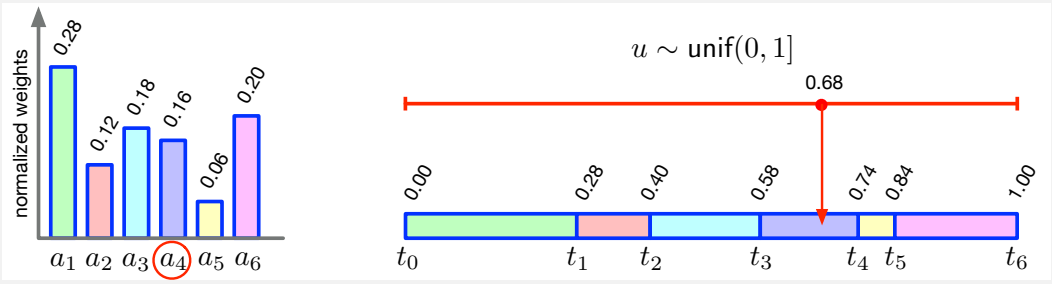
So now 99.87% of the time we get an estimate within 10,000. In fact, we get an estimate within 4,000 at least 38% of the time. Basically, this works because we expect to sample the CEO about twice, but then weight that contribution slightly higher. On the other hand, when we sample a different employee, we increase the effect of their salary by about 4%.

Implementation. It is standard for most programming languages to have built in functions to generate random numbers in a uniform distribution $u \sim \mathsf{unif}(0,1]$. This can easily be transformed to select an element from a large discrete set. If there are k elements in the set, then $i = \lceil uk \rceil$ (multiply by k and take the ceiling[3]) is a uniform selection of an integer from 1 to k, and represents selecting the ith element from the set.

This idea can easily be generalized to selecting an element proportional to its weight. Let $W = n\bar{w} = \sum_{i=1}^{n} w(a_i)$. Our goal is to sample element a_i with probability $w(a_i)/W$. We can also define a probability $t_j = \sum_{i=1}^{j} w(a_i)/W$, the probability that an object of index j or less should be selected. Once we calculate t_j for each a_j, and set $t_0 = 0$, a uniform random value $u \sim \mathsf{unif}(0,1)$ is all that is needed to select a object proportional to its weight. We just return the item a_j such that $t_{j-1} \leq u \leq t_j$ (the correct index j can be found in time proportional to $\log n$ if these t_j are stored in a balanced binary tree).

Geometry of Partition of Unity

In this illustration 6 elements with normalized weights $w(a_i)/W$ are depicted in a bar chart on the left. These bars are then stacked end-to-end in a unit interval on the right; the precisely stretch from 0.00 to 1.00. The t_i values mark the accumulation of probability that one of the first i values is chosen. Now when a random value $u \sim \mathsf{unif}(0,1]$ is chosen at random, it maps into this "partition of unity" and selects an item. In this case it selects item a_4 since $u = 0.68$ and $t_3 = 0.58$ and $t_4 = 0.74$ for $t_3 < u \leq t_4$.



2.4.1 Sampling Without Replacement with Priority Sampling

Many examples discussed in this book analyze data assumed to be k elements drawn iid from a distribution. However, when algorithmically generating samples it can be advantageous to sample k elements without replacement from a known distribution. While this can make the analysis slightly more complicated, variants of Chernoff-Hoeffding bound exists for without-replacement random samples instead of independent random variables. Moreover, in practice the concentration and approximation quality is often improved using without-replacement samples and is especially true when drawing weighted samples.

When sampling data proportional to weights, if elements exist with sufficiently large weights, then it is best to always sample these high-weight elements. The low-weight ones need to be selected with some probability, and this should be proportional to their weights, and then re-weighted as in importance sampling. A technique called *priority sampling* elegantly combines these properties.

[3]The ceiling operation $\lceil x \rceil$ returns the smallest integer larger than x. For instance $\lceil 7.342 \rceil = 8$.

Priority Sampling:

1. For item $a_i \in A$ with weight $w(a_i)$ generate $u_i \sim \mathsf{unif}(0,1]$. Set priority $\rho_i = w(a_i)/u_i$.
2. Let $\tau = \rho'_{k+1}$, the $(k+1)$th largest priority.
3. Assign new weights $w'(a_i) = \begin{cases} \max(w(a_i), \tau) & \text{if } \rho_i > \tau \\ 0 & \text{if } \rho_i \leq \tau \end{cases}$.

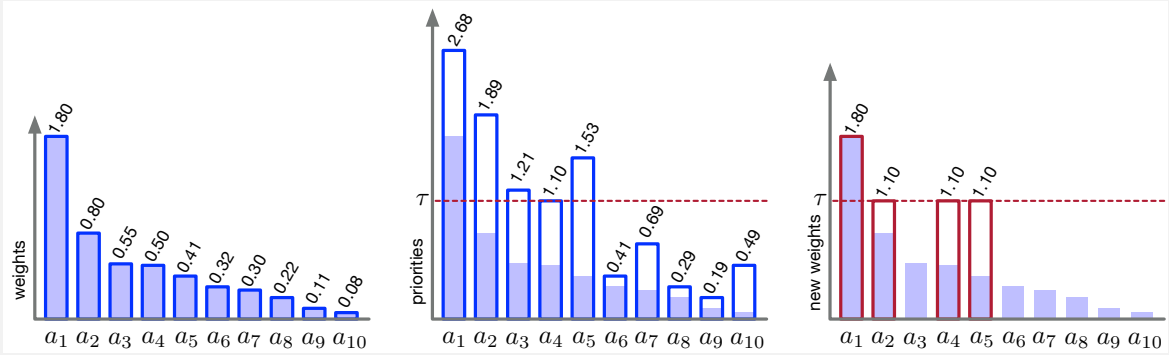
The new weight function $w' : A \to [0, \infty)$ has only k items with non-zero values, only those need to be retained in the sample A'. This has many nice properties. Most importantly $\mathbf{E}[w'(a_i)] = w(a_i)$. Thus for any subset $S \subset A$, we can estimate the sum of weights in that subset $\sum_{a_i \in S} w(a_i)$ using only $\sum_{a_i \in S \cap A'} w'(a_i)$ and this has the correct expected value. Thus for $w_P = \frac{1}{n}\sum_{i=1}^{n} w'(a_i)$ as an analog to importance sampling we also have $\mathbf{E}[w_P] = \bar{w}$.

Additionally, the elements with very large weights (those with weight above τ) are always retained. This is because $\rho_i \geq w(a_i)$ for all i (since $1/u_i \geq 1$), so if $w(a_i) > \tau$ then $\rho_i > \tau$ and it is always chosen, and its new weight is $w'(a_i) = \max(w(a_i), \tau) = w(a_i)$ is the same as before. Hence, for a fixed τ this item has no variance in its effect on the estimate. The remaining items have weights assigned as if in importance sampling, and so the overall estimate has small (and indeed near-optimal) variance.

Example: Priority Sampling

In this example, 10 items are shown with weights from $w(a_{10}) = 0.08$ to $w(a_1) = 1.80$. For a clearer picture, they are sorted in decreasing order. Each is then given a priority by dividing the weight by a different $u_i \sim \mathsf{unif}(0,1]$ for each element. To sample $k = 4$ items, the 5th-largest priority value $\rho_4 = \tau = 1.10$ (belonging to a_4) is marked by a horizontal dashed line. Then all elements with priorities above τ are given non-zero weights. The largest weight element a_1 retains its original weight $w(a_1) = w'(a_1) = 1.80$ because it is above τ. The other retained elements have weight below τ so are given new weights $w'(a_2) = w'(a_4) = w'(a_5) = \tau = 1.10$. The other elements are implicitly given new weights of 0.

Notice that $W' = \sum_{i=1}^{10} w'(a_i) = 5.10$ is very close to $W = \sum_{i=1}^{10} w(a_i) = 5.09$.



Its useful to understand why the new estimate W' does not necessarily increase if more elements are retained. In this case if $k = 5$ elements are retained instead of $k = 4$, then τ would be become $\rho_7 = 0.69$, the 6th largest priority. So then the new weights for several of the elements would decrease from 1.10 to 0.69.

Exercises

Q2.1: Consider a pdf f so that a random variable $X \sim f$ has expected value $\mathbf{E}[X] = 3$ and variance $\mathbf{Var}[X] = 10$. Now consider $n = 10$ iid random variables $X_1, X_2, \ldots, X_{10}$ drawn from f. Let $\bar{X} = \frac{1}{10}\sum_{i=1}^{10} X_i$.

1. What is $\mathbf{E}[\bar{X}]$?
2. What is $\mathbf{Var}[\bar{X}]$?
3. What is the standard deviation of $\bar{X}$?
4. Which is larger $\mathbf{Pr}[X > 4]$ or $\mathbf{Pr}[\bar{X} > 4]$?
5. Which is larger $\mathbf{Pr}[X > 2]$ or $\mathbf{Pr}[\bar{X} > 2]$?

Q2.2: Let X be a random variable that you know is in the range $[-1, 2]$ and you know has expected value of $\mathbf{E}[X] = 0$. Use the Markov Inequality to upper bound $\mathbf{Pr}[X > 1.5]$?
(Hint: you will need to use a change of variables.)

Q2.3: Consider a pdf f so that a random variable $X \sim f$ has expected value $\mathbf{E}[X] = 5$ and variance $\mathbf{Var}[X] = 100$. Now consider $n = 16$ iid random variables $X_1, X_2, \ldots, X_{16}$ drawn from f. Let $\bar{X} = \frac{1}{16}\sum_{i=1}^{16} X_i$.

1. What is $\mathbf{E}[\bar{X}]$?
2. What is $\mathbf{Var}[\bar{X}]$?

Assume we know that X is never smaller than 0 and never larger than 20.

3. Use the Markov inequality to upper bound $\mathbf{Pr}[\bar{X} > 8]$.
4. Use the Chebyshev inequality to upper bound $\mathbf{Pr}[\bar{X} > 8]$.
5. Use the Chernoff-Hoeffding inequality to upper bound $\mathbf{Pr}[\bar{X} > 8]$.
6. If we increase n to 100, how will the above three bounds be affected.

Q2.4: Consider a (parked) self-driving car that returns n iid estimates to the distance of a tree. We will model these n estimates as a set of n scalar random variables $X_1, X_2, \ldots, X_n$ taken iid from an unknown pdf f, which we assume models the true distance plus unbiased noise. (The sensor can take many iid estimates in rapid fire fashion.) The sensor is programmed to only return values between 0 and 20 feet, and that the variance of the sensing noise is 64 feet squared. Let $\bar{X} = \frac{1}{n}\sum_{i=1}^{n} X_i$. We want to understand as a function of n how close $\bar{X}$ is to μ, which is the true distance to the tree.

1. Use Chebyshev's Inequality to determine a value n so that $\mathbf{Pr}[|\bar{X} - \mu| \geq 1] \leq 0.5$.
2. Use Chebyshev's Inequality to determine a value n so that $\mathbf{Pr}[|\bar{X} - \mu| \geq 0.1] \leq 0.1$.
3. Use the Chernoff-Hoeffding bound to determine a value n so that $\mathbf{Pr}[|\bar{X} - \mu| \geq 1] \leq 0.5$.
4. Use the Chernoff-Hoeffding bound to determine a value n so that $\mathbf{Pr}[|\bar{X} - \mu| \geq 0.1] \leq 0.1$.

Q2.5: Consider two random variables C and T describing how many coffees and teas I will buy in the coming week; clearly neither can be smaller than 0. Based on personal experience, I know the following summary statistics about my coffee and tea buying habits: $\mathbf{E}[C] = 3$ and $\mathbf{Var}[C] = 1$ also $\mathbf{E}[T] = 2$ and $\mathbf{Var}[T] = 5$.

1. Use Markov's Inequality to upper bound the probability I buy 4 or more coffees, and the same for teas: $\mathbf{Pr}[C \geq 4]$ and $\mathbf{Pr}[T \geq 4]$.

2. Use Chebyshev's Inequality to upper bound the probability I buy 4 or more coffees, and the same for teas: $\mathbf{Pr}[C \geq 4]$ and $\mathbf{Pr}[T \geq 4]$.

Q2.6: The average score on a test is 82 with a standard deviation of 4 percentage points. All tests have scores between 0 and 100.

1. Using Chebyshev's inequality, at least what percentage of the tests have a grade of at least 70 and at most 94?
2. Using Markov's inequality, what is the highest percentage of tests which could have score less than 60?

3 Linear Algebra Review

For this topic we quickly review many key aspects of linear algebra that will be necessary for the remainder of the text.

3.1 Vectors and Matrices

For the context of data analysis, the critical part of linear algebra deals with vectors and matrices of real numbers.

In this context, a *vector* $v = (v_1, v_2, \ldots, v_d)$ is equivalent to a point in $\mathbb{R}^d$. By default a vector will be a column of d numbers (where d is context specific)

$$v = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix}$$

but in some cases we will assume the vector is a row

$$v^T = [v_1 \ v_2 \ \ldots \ v_n].$$

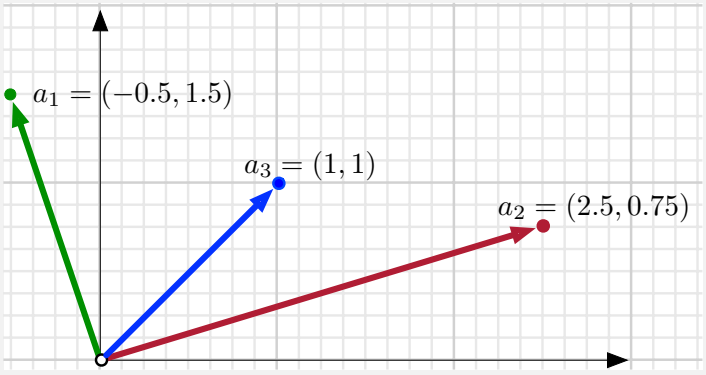
An $n \times d$ *matrix* A is then an ordered set of n row vectors $a_1, a_2, \ldots a_n$

$$A = [a_1; \ a_2; \ \ldots \ a_n] = \begin{bmatrix} - & a_1 & - \\ - & a_2 & - \\ & \vdots & \\ - & a_n & - \end{bmatrix} = \begin{bmatrix} A_{1,1} & A_{1,2} & \ldots & A_{1,d} \\ A_{2,1} & A_{2,2} & \ldots & A_{2,d} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n,1} & A_{n,2} & \ldots & A_{n,d} \end{bmatrix},$$

where vector $a_i = [A_{i,1}, A_{i,2}, \ldots, A_{i,d}]$, and $A_{i,j}$ is the element of the matrix in the ith row and jth column. We can write $A \in \mathbb{R}^{n \times d}$ when it is defined on the reals.

Geometry of Vectors and Matrices

It will be convenient to image length d vectors v as points in $\mathbb{R}^d$. And subsequently it will be convenient to think of a $n \times d$ matrix A by each of its rows $a_1, \ldots, a_n$ as each a point in $\mathbb{R}^d$. The "vector" is then the "arrow" from the origin $\mathbf{0} = (0, 0, \ldots, 0)$ to that point.



This picture with $n = 3$ points in $d = 2$ dimensions is equivalent to the 3×2 matrix representation

$$A = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} -0.5 & 1.5 \\ 2.5 & 0.75 \\ 1 & 1 \end{bmatrix}.$$

A *transpose* operation $(\cdot)^T$ reverses the roles of the rows and columns, as seen above with vector v. For a matrix, we can write:

$$A^T = \begin{bmatrix} | & | & & | \\ a_1 & a_2 & \ldots & a_n \\ | & | & & | \end{bmatrix} = \begin{bmatrix} A_{1,1} & A_{2,1} & \ldots & A_{n,1} \\ A_{1,2} & A_{2,2} & \ldots & A_{n,2} \\ \vdots & \vdots & \ddots & \vdots \\ A_{1,d} & A_{2,d} & \ldots & A_{n,d} \end{bmatrix}.$$

Example: Linear Equations

A simple place these objects arise is in linear equations. For instance

$$\begin{array}{rrrl} 3x_1 & -7x_2 & +2x_3 & = -2 \\ -1x_1 & +2x_2 & -5x_3 & = 6 \end{array}$$

is a system of $n = 2$ linear equations, each with $d = 3$ variables. We can represent this system in matrix-vector notation as

$$Ax = b$$

where

$$b = \begin{bmatrix} -2 \\ 6 \end{bmatrix} \quad x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \quad \text{and} \quad A = \begin{bmatrix} 3 & -7 & 2 \\ -1 & 2 & -5 \end{bmatrix}.$$

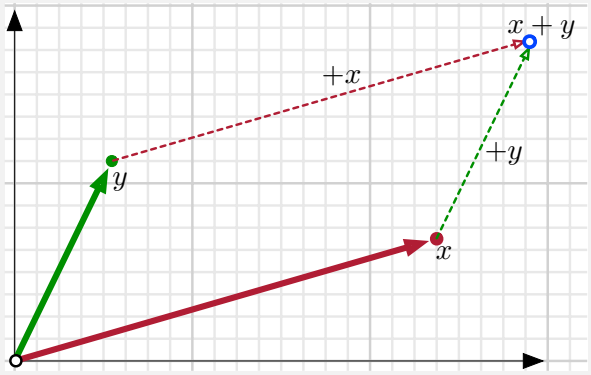
3.2 Addition and Multiplication

We can add together two vectors or two matrices only if they have the same dimensions. For vectors $x = (x_1, x_2, \ldots, x_d) \in \mathbb{R}^d$ and $y = (y_1, y_2, \ldots, y_d) \in \mathbb{R}^d$, then vector

$$z = x + y = (x_1 + y_1, x_2 + y_2, \ldots, x_d + y_d) \in \mathbb{R}^d.$$

Geometry of Vector Addition

Vector addition can be geometrically realized as just chaining two vectors together. It is easy to see that this operation is commutative. That is $x + y = y + x$, since it does not matter which order we chain the vectors, both result in the same summed-to point.



Similarly for two matrices $A, B \in \mathbb{R}^{n \times d}$, then $C = A + B$ is defined where $C_{i,j} = A_{i,j} + B_{i,j}$ for all i, j.

Multiplication only requires alignment along one dimension. For two matrices $A \in \mathbb{R}^{n \times d}$ and $B \in \mathbb{R}^{d \times m}$ we can obtain a new matrix $C = AB \in \mathbb{R}^{n \times m}$ where $C_{i,j}$, the element in the ith row and jth column of C is defined

$$C_{i,j} = \sum_{k=1}^{d} A_{i,k} B_{k,j}.$$

To multiply A times B (where A is to the left of B, the order matters!) then we require the row dimension d of A to match the column dimension d of B. If $n \neq m$, then we *cannot* multiply BA. Keep in mind:

- Matrix multiplication is *associative* $(AB)C = A(BC)$.
- Matrix multiplication is *distributive* $A(B + C) = AB + AC$.
- Matrix multiplication is ***not*** *commutative* $AB \neq BA$.

We can also multiply a matrix A by a scalar α. In this setting $\alpha A = A\alpha$ and is defined by a new matrix B where $B_{i,j} = \alpha A_{i,j}$.

vector-vector products. There are two types of vector-vector products, and their definitions follow directly from that of matrix-matrix multiplication (since a vector is a matrix where one of the dimensions is 1). But it is worth highlighting these.

Given two column vectors $x, y \in \mathbb{R}^d$, the *inner product* or *dot product* is written

$$x^T y = x \cdot y = \langle x, y \rangle = [x_1\ x_2\ \ldots\ x_d] \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_d \end{bmatrix} = \sum_{i=1}^{d} x_i y_i,$$

where x_i is the ith element of x and similar for y_i. This text will prefer the last notation $\langle x, y \rangle$ since the same can be used for row vectors, and there is no confusion with scalar multiplication in using $x \cdot y$. Whether a vector is a row or a column is often arbitrary, in a computer they are typically stored the same way in memory.

Note that this dot product operation produces a single scalar value. And it is a linear operator. So this means for any scalar value α and three vectors $x, y, z \in \mathbb{R}^d$ we have

$$\langle \alpha x, y + z \rangle = \alpha \langle x, y + z \rangle = \alpha \left(\langle x, y \rangle + \langle x, z \rangle \right).$$

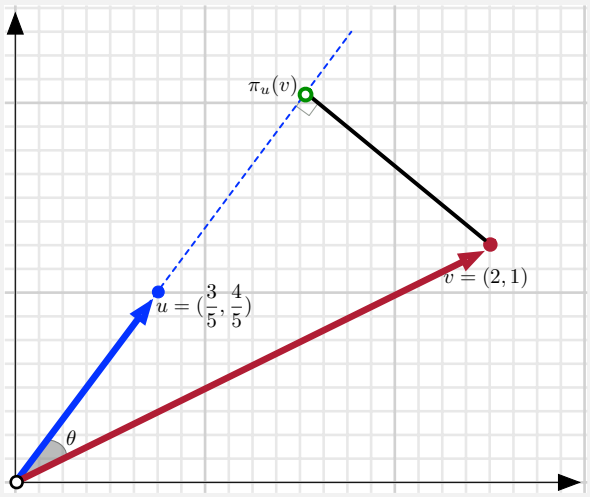
This operation is associative, distributive, and commutative.

Geometry of the Dot Product

A dot product is one of my favorite mathematical operations! It encodes a lot of geometry. Consider two vectors $u = (\frac{3}{5}, \frac{4}{5})$ and $v = (2, 1)$, with an angle θ between them. Then it holds

$$\langle u, v \rangle = \text{length}(u) \cdot \text{length}(v) \cdot \cos(\theta).$$

Here $\text{length}(\cdot)$ measures the distance from the origin. We'll see how to measure length with a "norm" $\| \cdot \|$ soon.



Moreover, since $\|u\| = \text{length}(u) = 1$, then we can also interpret $\langle u, v \rangle$ as the length of v projected onto the line through u. That is, let $\pi_u(v)$ be the closest point to v on the line through u (the line through u and the line segment from v to $\pi_u(v)$ make a right angle). Then

$$\langle u, v \rangle = \text{length}(\pi_u(v)) = \|\pi_u(v)\|.$$

For two column vectors $x \in \mathbb{R}^n$ and $y \in \mathbb{R}^d$, the *outer product* is written

$$y^T x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} [y_1 \; y_2 \; \ldots \; y_d] = \begin{bmatrix} x_1 y_1 & x_1 y_2 & \ldots & x_1 y_d \\ x_2 y_1 & x_2 y_2 & \ldots & x_2 y_d \\ \vdots & \vdots & \ddots & \vdots \\ x_n y_1 & x_n y_2 & \ldots & x_n y_d \end{bmatrix} \in \mathbb{R}^{n \times d}.$$

Note that the result here is a matrix, not a scalar. The dimensions are not required to match.

matrix-vector products. Another important and common operation is a matrix-vector product. Given a matrix $A \in \mathbb{R}^{n \times d}$ and a vector $x \in \mathbb{R}^d$, their product $y = Ax \in \mathbb{R}^n$.

When A is composed of row vectors $[a_1; a_2; \ldots; a_n]$, then it is useful to imagine this as transposing x (which should be a column vector here, so a row vector after transposing), and taking the dot product with each row of A. I like to think of this as x^T sliding down the rows of A, and for each row a_i outputting a scalar value $\langle a_i, x \rangle$ into the corresponding output vector.

$$y = Ax = \begin{bmatrix} - & a_1 & - \\ - & a_2 & - \\ & \vdots & \\ - & a_n & - \end{bmatrix} x = \begin{bmatrix} \langle a_1, x \rangle \\ \langle a_2, x \rangle \\ \vdots \\ \langle a_n, x \rangle \end{bmatrix}.$$

3.3 Norms

The standard *Euclidean norm* (think "length") of a vector $v = (v_1, v_2, \ldots, v_d) \in \mathbb{R}^d$ is defined

$$\|v\| = \sqrt{\sum_{i=1}^{d} v_i^2} = \sqrt{v_1 v_1 + v_2 v_2 + \ldots + v_d v_d} = \sqrt{\langle v, v \rangle}.$$

This measures the "straight-line" distance from the origin to the point at v. A vector v with norm $\|v\| = 1$ is said to be a *unit vector*; sometimes a vector x with $\|x\| = 1$ is said to be *normalized*.

However, a "norm" is a more generally concept. A class called L_p *norms* are well-defined for any parameter $p \in [1, \infty)$ as

$$\|v\|_p = \left(\sum_{i=1}^{d} |v_i|^p \right)^{1/p}.$$

Thus, when no p is specified, it is assumed to be $p = 2$. It is also common to denote $\|v\|_\infty = \max_{i=1}^{d} |v_i|$, which is also a norm. Indeed this is the result of taking the limit of p to ∞.

We can also define norms for matrices A. These take on slightly different notational conventions. The two most common are the spectral norm $\|A\| = \|A\|_2$ and the Frobenius norm $\|A\|_F$. The *Frobenius norm* is the most natural extension of the $p = 2$ norm for vectors, but uses a subscript F instead. It is defined for matrix $A \in \mathbb{R}^{n \times d}$ as

$$\|A\|_F = \sqrt{\sum_{i=1}^{n} \sum_{j=1}^{d} A_{i,j}^2} = \sqrt{\sum_{i=1}^{n} \|a_i\|^2},$$

where $A_{i,j}$ is the element in the ith row and jth column of A, and where a_i is the ith row vector of A. The *spectral norm* is defined for a matrix $A \in \mathbb{R}^{n \times d}$ as

$$\|A\| = \|A\|_2 = \max_{\substack{x \in \mathbb{R}^d \\ \|x\| \neq 0}} \|Ax\| / \|x\| = \max_{\substack{y \in \mathbb{R}^n \\ \|y\| \neq 0}} \|yA\| / \|y\|.$$

Its useful to think of these x and y vectors as being unit vectors, then the denominator can be ignored (as they are 1). Then we see that x and y only contain "directional" information, and the $\arg\max$ vector (e.g., the x which maximizes $\|Ax\|/\|x\|$) point in the directions that maximize the norm.

3.4 Linear Independence

Consider a set of k vectors $x_1, x_2, \ldots, x_k \in \mathbb{R}^d$, and a set of k scalars $\alpha_1, \alpha_2, \ldots, \alpha_k \in \mathbb{R}$. Then because of linearity of vectors, we can write a new vector in $\mathbb{R}^d$ as

$$z = \sum_{i=1}^{k} \alpha_i x_i.$$

For a set of vectors $X = \{x_1, x_2, \ldots, x_k\}$, for any vector z such that there exists a set of scalars α so z can be written as the above summation, then we say z is *linearly dependent* on X. If z **cannot** be written with any choice of α_is, the we say z is *linearly independent* of X. All vectors $z \in \mathbb{R}^d$ which are linearly dependent on X are said to be in its *span*.

$$\mathsf{span}(X) = \left\{ z \;\middle|\; z = \sum_{i=1}^{k} \alpha_i x_i, \quad \alpha_i \in \mathbb{R} \right\}.$$

If $\mathsf{span}(X) = \mathbb{R}^d$ (that is for vectors $X = x_1, x_2, \ldots, x_k \in \mathbb{R}^d$ *all* vectors are in the span), then we say X forms a *basis*.

Example: Linear Independence

Consider input vectors in a set X as

$$x_1 = \begin{bmatrix} 1 \\ 3 \\ 4 \end{bmatrix} \qquad x_2 = \begin{bmatrix} 2 \\ 4 \\ 1 \end{bmatrix}$$

And two other vectors

$$z_1 = \begin{bmatrix} -3 \\ -5 \\ 2 \end{bmatrix} \qquad z_2 = \begin{bmatrix} 3 \\ 7 \\ 1 \end{bmatrix}$$

Note that z_1 is linearly dependent on X since it can be written as $z_1 = x_1 - 2x_2$ (here $\alpha_1 = 1$ and $\alpha_2 = -2$). However z_2 is linearly independent from X since there are no scalars α_1 and α_2 so that $z_2 = \alpha_1 x_1 + \alpha_2 x_2$ (we need $\alpha_1 = \alpha_2 = 1$ so the first two coordinates align, but then the third coordinate cannot).
Also the set X is linearly independent, since there is no way to write $x_2 = \alpha_1 x_1$.

A set of vectors $X = \{x_1, x_2, \ldots, x_n\}$ is *linearly independent* if there is no way to write any vector $x_i \in X$ in the set with scalars $\{\alpha_1, \ldots, \alpha_{i-1}, \alpha_{i+1}, \ldots, \alpha_n\}$ as the sum

$$x_i = \sum_{\substack{j=1 \\ j \neq i}}^{n} \alpha_j x_j$$

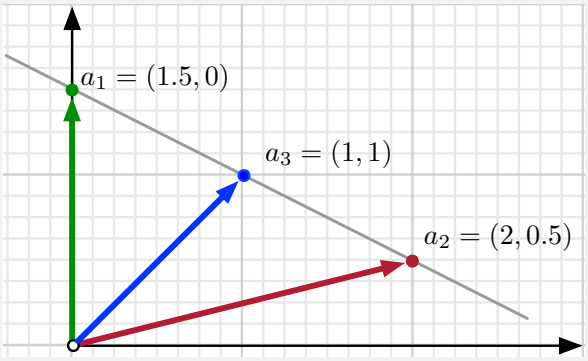
of the other vectors in the set.

Geometry of Linear Dependence

A geometric way to understand linear dependence is if there is a lower-dimensional subspace that passes through all of the points. Consider the example of the 3×2 matrix A which corresponds with 3 points in 2 dimensions.

$$A = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} 1.5 & 0 \\ 2 & 0.5 \\ 1 & 1 \end{bmatrix}.$$

Then there points are linearly dependent because there exists a line $\ell : y = -0.5x + 1.5$ which passes through all points.



3.5 Rank

The *rank* of a set of vectors $X = \{x_1, \ldots, x_n\}$ is the size of the largest subset $X' \subset X$ which are linearly independent. Usually we report $\mathsf{rank}(A)$ as the rank of a matrix A. It is defined as the rank of the rows of the matrix, or the rank of its columns; it turns out these quantities are always the same.

If $A \in \mathbb{R}^{n \times d}$, then $\mathsf{rank}(A) \leq \min\{n, d\}$. If $\mathsf{rank}(A) = \min\{n, d\}$, then A is said to be *full rank*. For instance, if $d < n$, then using the rows of $A = [a_1; a_2; \ldots; a_n]$, we can describe *any* vector $z \in \mathbb{R}^d$ as the linear combination of these rows: $z = \sum_{i=1}^{n} \alpha_i a_i$ for some set $\{\alpha_1, \ldots, \alpha_n\}$. In fact, if A is full rank we can do so and set all but d of these scalars to 0.

3.6 Inverse

A matrix A is said to be *square* if it has the same number of column as it has rows. A square matrix $A \in \mathbb{R}^{n \times n}$ may have an *inverse* denoted A^{-1}. If it exists, it is a unique matrix which satisfies:

$$A^{-1}A = I = AA^{-1}$$

where I is the $n \times n$ identity matrix

$$I = \begin{bmatrix} 1 & 0 & \ldots 0 & 0 & \\ 0 & 1 & \ldots 0 & 0 & \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \ldots & 1 & 0 \\ 0 & 0 & \ldots & 0 & 1 \end{bmatrix} = \mathsf{diag}(1, 1, \ldots, 1).$$

Note that I serves the purpose of 1 in scalar algebra, so for any (non-zero) scalar α then using $\alpha^{-1} = \frac{1}{\alpha}$ we have $\alpha\alpha^{-1} = 1 = \alpha^{-1}\alpha$.

A matrix is said to be *invertable* if it has an inverse. Only square, full-rank matrices are invertable; and a matrix is always invertable if it is square and full rank. If a matrix is not square, the inverse is not defined. If a matrix is not full rank, then it does not have an inverse.

3.7 Orthogonality

Two vectors $x, y \in \mathbb{R}^d$ are *orthogonal* if $\langle x, y \rangle = 0$. This means those vectors are at a right angle to each other.

Example: Orthongonality

Consider two vectors $x = (2, -3, 4, -1, 6)$ and $y = (4, 5, 3, -7, -2)$. They are orthogonal since

$$\langle x, y \rangle = (2 \cdot 4) + (-3 \cdot 5) + (4 \cdot 3) + (-1 \cdot -7) + (6 \cdot -2) = 8 - 15 + 12 + 7 - 12 = 0.$$

A square matrix $U \in \mathbb{R}^{n \times n}$ is *orthogonal* if all of its columns $[u_1, u_2, \ldots, u_n]$ are normalized and are all orthogonal with each other. It follows that

$$U^T U = I = U U^T$$

since for any normalized vector u that $\langle u, u \rangle = \|u\| = 1$, and any two distinct columns $u_i \neq u_j$ then $\langle u_i, u_j \rangle = 0$.

A set of columns (for instance those of an orthogonal U) which are normalized and all orthogonal to each other are said to be *orthonormal*. If $U \in \mathbb{R}^{n \times d}$ and has orthonormal columns, then $U^T U = I$ (here I is $d \times d$) but $UU^T \neq I$.

Orthogonal matrices are norm preserving under multiplication. That means for an orthogonal matrix $U \in \mathbb{R}^{n \times n}$ and any vector $x \in \mathbb{R}^n$, then $\|Ux\| = \|x\|$.

Moreover, the columns $[u_1, u_2, \ldots, u_n]$ of an orthogonal matrix $U \in \mathbb{R}^{n \times n}$ form an *basis* for $\mathbb{R}^n$. This means that for any vector $x \in \mathbb{R}^n$, there exists a set of scalars $\alpha_1, \ldots, \alpha_n$ such that $x = \sum_{i=1}^n \alpha_i u_i$. More interestingly, we also have $\|x\|^2 = \sum_{i=1}^n \alpha_i^2$.

This can be interpreted as U describing a *rotation* (with possible mirror flips) to a new set of coordinates. That is the old coordinates of x are $(x_1, x_2, \ldots, x_n)$ and the coordinates in the new orthogonal basis $[u_1, u_2, \ldots, u_n]$ are $(\alpha_1, \alpha_2, \ldots, \alpha_n)$.

```
import numpy as np
from numpy import linalg as LA

#create an array, a row vector
v = np.array([1,2,7,5])
print v
#[1 2 7 5]
print v[2]
#7

#create a n=2 x d=3 matrix
A = np.array([[3,4,3],[1,6,7]])
print A
#[[3 4 3]
# [1 6 7]]
print A[1,2]
#7
print A[:, 1:3]
#[[4 3]
# [6 7]]
```

```
#adding and multiplying vectors
u = np.array([3,4,2,2])
#elementwise add
print v+u
#[4 6 9 7]
#elementwise multiply
print v*u
#[ 3  8  14 10]
# dot product
print v.dot(u)
# 35
print np.dot(u,v)
# 35

#matrix multiplication
B = np.array([[1,2],[6,5],[3,4]])
print A.dot(B)
#[[36 38]
# [58 60]]
x = np.array([3,4])
print B.dot(x)
#[11 38 25]

#norms
print LA.norm(v)
#8.88819441732
print LA.norm(v,1)
#15.0
print LA.norm(v,np.inf)
#7.0
print LA.norm(A, 'fro')
#10.9544511501
print LA.norm(A,2)
#10.704642743

#transpose
print A.T
#[[3 1]
# [4 6]
# [3 7]]
print x.T
#[3 4]   (always prints in row format)

print LA.matrix_rank(A)
#2
C = np.array([[1,2],[3,5]])
print LA.inv(C)
#[[-5.  2.]
# [ 3. -1.]]
print C.dot(LA.inv(C))
#[[  1.00000000e+00   2.22044605e-16]       (nearly  [[1 0]
# [  0.00000000e+00   1.00000000e+00]]                [0 1]] )
```

Exercises

Q3.1: Consider a matrix

$$A = \begin{bmatrix} 2 & 2 & 3 \\ -2 & 7 & 4 \\ -3 & -3 & -4 \\ -8 & 2 & 3 \end{bmatrix}.$$

1. Add a column to A so that it is invertable.
2. Remove a row from A so that it is invertable.
3. Is AA^T invertable?
4. Is A^TA invertable?

Q3.2: Consider two vectors $u = (0.5, 0.4, 0.4, 0.5, 0.1, 0.4, 0.1)$ and $v = (-1, -2, 1, -2, 3, 1, -5)$.

1. Check if u or v is a unit vector.
2. Calculate the dot product $\langle u, v \rangle$.
3. Are u and v orthogonal?

Q3.3: Consider the following 3 vectors in $\mathbb{R}^9$:

$$\begin{aligned} v &= (1, 2, 5, 2, -3, 1, 2, 6, 2) \\ u &= (-4, 3, -2, 2, 1, -3, 4, 1, -2) \\ w &= (3, 3, -3, -1, 6, -1, 2, -5, -7) \end{aligned}$$

Report the following:

1. $\langle v, w \rangle$
2. Are any pair of vectors orthogonal, and if so which ones?
3. $\|u\|_2$
4. $\|w\|_\infty$

Q3.4: Consider the following 3 matrices:

$$A = \begin{bmatrix} 2 & -2 \\ -3 & 1 \\ 5 & -3 \end{bmatrix} \quad B = \begin{bmatrix} 4 & 4 & 4 \\ -2 & 3 & -7 \\ 2 & 5 & -7 \end{bmatrix} \quad C = \begin{bmatrix} 4 & -1 & 2 \\ -8 & 2 & -4 \\ 2 & 1 & -4 \end{bmatrix}$$

Report the following:

1. A^TB
2. $C + B$
3. Which matrices are full rank?
4. $\|C\|_F$
5. $\|A\|_2$
6. B^{-1}

4 Distances and Nearest Neighbors

At the core of most data analysis tasks and their formulations is a distance. This **choice** anchors the meaning and the modeling inherent in the patterns found and the algorithms used. However, there are an enormous number of distances to choose from. In this chapter provide an overview of the typically most important properties of distances (e.g., is it a metric?) and how it related to the dual notion of a similarities. We provide some common modeling dynamics which motivate some of the distances, and overview their direct uses in nearest neighbor approaches, and how to algorithmically deal with the challenges that arise.

4.1 Metrics

So what makes a good distance? There are two aspects to the answer to this question. The first is that it captures the "right" properties of the data, but this is a sometimes ambiguous modeling problem. The second is more well-defined; it is the properties which makes a distance a metric.

A distance $\mathbf{d} : \mathcal{X} \times \mathcal{X} \to \mathbb{R}^+$ is a bivariate operator (it takes in two arguments, say $a \in \mathcal{X}$ and $b \in \mathcal{X}$) that maps to $\mathbb{R}^+ = [0, \infty)$. It is a *metric* if

(M1) $\mathbf{d}(a, b) \geq 0$ (**non-negativity**)

(M2) $\mathbf{d}(a, b) = 0$ if and only if $a = b$ (**identity**)

(M3) $\mathbf{d}(a, b) = \mathbf{d}(b, a)$ (**symmetry**)

(M4) $\mathbf{d}(a, b) \leq \mathbf{d}(a, c) + \mathbf{d}(c, b)$ (**triangle inequality**)

A distance that satisfies (M1), (M3), and (M4) (but not necessarily (M2)) is called a *pseudometric*.

A distance that satisfies (M1), (M2), and (M4) (but not necessarily (M3)) is called a *quasimetric*.

In the next few sections we outline a variety of common distances used in data analysis, and provide examples of their uses cases.

4.2 L_p Distances and their Relatives

We next introduce a specific family of distances between vectors $a, b \in \mathbb{R}^d$. As they are defined between vectors, the most common ones are defined purely from notions of norms in linear algebra. But other variants will restrict vectors to model specific sorts of data like probability distribution, and then draw from more probabilistic elements.

4.2.1 L_p Distances

Consider two vectors $a = (a_1, a_2, \ldots, a_d)$ and $b = (b_1, b_2, \ldots, b_d)$ in $\mathbb{R}^d$. Now an L_p distances is defined as

$$\mathbf{d}_p(a, b) = \|a - b\|_p = \left(\sum_{i=1}^{d} (|a_i - b_i|)^p \right)^{1/p}.$$

1. The most common is the L_2 distance

$$\mathbf{d}_2(a, b) = \|a - b\| = \|a - b\|_2 = \sqrt{\sum_{i=1}^{d} (a_i - b_i)^2}.$$

It easy interpreted as the *Euclidean* or "straight-line" distance between two points or vectors, since if you draw a line between two points, its length measures the Euclidean distance.
It is also the only L_p distance that is invariant to the rotation of the coordinate system (which will often be useful, but sometimes restrictive).

2. Another common distance is the L_1 distance

$$\mathbf{d}_1(a,b) = \|a-b\|_1 = \sum_{i=1} |a_i - b_i|.$$

This is also known as the "Manhattan" distance since it is the sum of lengths on each coordinate axis; the distance you would need to walk in a city like Manhattan since you must stay on the streets and can't cut through buildings.

3. A common modeling goal is the L_0 distance

$$\mathbf{d}_0(a,b) = \|a-b\|_0 = d - \sum_{i=1}^{d} \mathbb{1}(a=b),$$

where $\mathbb{1}(a=b) = \begin{cases} 1 & \textbf{if } a=b \\ 0 & \textbf{if } a \neq b. \end{cases}$ Unfortunately, $\mathbf{d}_0$ is not convex.
When each coordinate a_i is either 0 or 1, then this is known as the *Hamming distance*.

4. Finally, another useful variation is the L_∞ distance

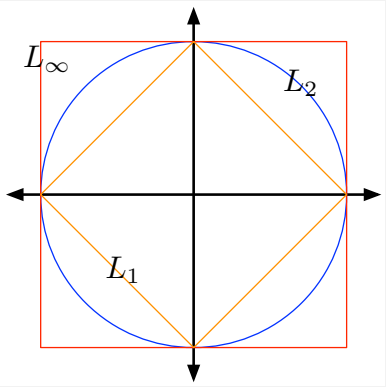
$$\mathbf{d}_\infty(a,b) = \|a-b\|_\infty = \max_{i=1\ldots d} |a_i - b_i|.$$

It is the maximum deviation along any one coordinate. Geometrically in $\mathbb{R}^2$, it is a rotation of the L_1 distance, so many algorithms designed for L_1 can be adapted to L_∞ in $\mathbb{R}^2$. However, in high dimensions they can act in surprisingly different ways. $\mathbf{d}_\infty$ is not technically an L_p distance, but is the limit of such distances as p goes to ∞.

Geometry of L_p Unit Balls

A useful way to imagine the geometry of these various L_p distances is by considering their unit balls. Much of the relevant information is conveyed even in $\mathbb{R}^2$. A *unit ball* is the set of points $a \in \mathbb{R}^2$ so that $\mathbf{d}(\mathbf{0}, a) = 1$ for our choice of distance $\mathbf{d}$.

The unit balls for L_1, L_2, and L_∞ distance are shown in $\mathbb{R}^2$. The most important points to notice are that L_1 is never larger than L_2 which is never larger than L_∞. Moreover, they are always equal along the coordinate axis, and these are the only places they are the same. Indeed these principles hold for any L_p distance where the L_p ball is never greater than the $L_{p'}$ ball for $p < p'$, and they are only and always equal along the coordinate axis.



The L_2 ball is the only distance invariant to the choice of axis. This means, for instance, that if it is rotated it stays the same shape. This is not true for any other L_p balls.

It is also possible to draw L_p balls for $p < 1$. However, these balls are not convex (see T7), and they "curve in" between the coordinate axis. Algorithmically, this makes them difficult to work with. It also, in effect, is the reason the associated L_p distances are not a metric, it is not hard to construct example where they violate the triangle inequality due to this non-convexity.

Metric properties. All of these distances are metrics, and in general for L_p for $p \in [1, \infty)$. (M1) and (M2) hold since the distances are at the core a sum of non-negative terms, and are only all 0 if all coordinates are identical. (M3) holds since $|a_i - b_i| = |b_i - a_i|$, vector subtraction is symmetric. (M4 - triangle inequality) is a bit trickier to show, but follows by drawing a picture $\ddot{\smile}$.

Warning about L_p Distance: These should *not* be used to model data when the units and meaning on each coordinate are not the same. For instance, consider representing two people p_1 and p_2 as points in $\mathbb{R}^3$ where the x-coordinate represents height in inches, the y-coordinate represents weight in pounds, and the z-coordinate represents income in dollars per year. Then most likely this distance is dominated by the z-coordinate income which might vary on the order of $10{,}000$ while the others vary on the order of 10.

Also, for the same data we could change the units, so the x-coordinate represents height in meters, the y-coordinate represents weight in centigrams, and the z-coordinate represents income in dollars per hour. The information may be exactly the same, only the unit changed. It is now likely dominated by the y-coordinate representing weight.

These sorts of issues can hold for distance other than L_p as well. A safe way is to avoid these issues is to use the L_0 metric – however this one can be crude and insensitive to small variations in data. Some heuristics to overcome this is: set hand-tuned scaling of each coordinate, "normalize" the distance so they all have the same min and max value (e.g., all in the range $[0, 1]$), or "normalize" the distance so they all have the same mean and variance. All of these are hacks and may have unintended consequences. For instance the $[0, 1]$ normalization is at the mercy of outliers, and mean-variance normalization can have strange effects in multi-modal distributions. *These are not solutions, they are hacks!*

With some additional information about which points are "close" or "far" one may be able to use the field of *distance metric learning* to address some of these problems. A simple solution can the be derived over all Mahalanobis distances (see below), using some linear algebra and gradient descent. But without this information, there is no one right answer. If you axes are the numbers of apples (x-axis) and number of oranges (y-axis), then its *literally comparing apples to oranges!*

4.2.2 Mahalanobis Distance

An extension to the L_2 distance is the *Mahalanobis* distance defined for two vectors $a, b \in \mathbb{R}^d$ and a $d \times d$ matrix M as

$$\mathbf{d}_M(a,b) = \sqrt{(a-b)^T M (a-b)}.$$

When $M = I$ (the identity matrix, so $I_{j,j} = 1$ and $I_{j,j'} = 0$ for $j \neq j'$), then $\mathbf{d}_M = \mathbf{d}_2$. When M is a diagonal matrix (so $M_{j,j'} = 0$ for $j \neq j'$) then $\mathbf{d}_M$ can be interpreted as skewing the Euclidean space (shrink some coordinates, and expanding others) based on the matrix M. When M is not diagonal, the skewing of Euclidean space still holds, but the skew is not aligned with the coordinate axis; instead it is defined through the eigenvectors and by the eigenvalues of M (see T8). As long as all eigenvalues are positive and real (implying M is positive definite) then $\mathbf{d}_M$ is a metric; since then the skew is well-defined and full-dimensional.

4.2.3 Cosine and Angular Distance

The *cosine distance* measures the 1 minus the cosine of the "angle" between vectors $a = (a_1, a_2, \ldots, a_d)$ and $b = (b_1, b_2, \ldots, b_d)$ in $\mathbb{R}^d$

$$\mathbf{d}_{\cos}(a,b) = 1 - \frac{\langle a, b\rangle}{\|a\|\|b\|} = 1 - \frac{\sum_{i=1}^d a_i b_i}{\|a\|\|b\|}.$$

Recall that if $\theta_{a,b}$ is the angle between vectors a and b then $\cos(\theta_{a,b}) = \frac{\langle a,b\rangle}{\|a\|\|b\|}$ Hence $\mathbf{d}_{\cos}(a,b) = 1 - \cos(\theta_{a,b})$.

Note that $\mathbf{d}(A,B) \in [0,\pi]$ and it does not depend on the magnitude $\|a\|$ of the vectors since this is normalized out. It only cares about their directions. This is useful when a vector of objects represent data sets of different sizes and we want to compare how similar are those distributions, but not their size. This makes $\mathbf{d}_{\cos}$ a psuedo-distance since for two vectors a and $a' = (2a_1, 2a_2, \ldots, 2a_d)$ where $\|a'\| = 2\|a\|$ have $\mathbf{d}_{\cos}(a,a') = 0$, but they are not equal.

Sometimes $\mathbf{d}_{\cos}$ is defined only with respect to normalized vectors $a, b \in \mathbb{S}^{d-1}$, where

$$\mathbb{S}^{d-1} = \left\{ x \in \mathbb{R}^d \mid \|x\| = 1 \right\}.$$

In this case, then more simply

$$\mathbf{d}_{\cos}(a,b) = 1 - \langle a, b\rangle.$$

Restricted to vectors in $\mathbb{S}^{d-1}$, then $\mathbf{d}_{\cos}$ does not have the issue of two vectors $a \neq b$ such that $\mathbf{d}_{\cos}(a,b) = 0$. However, it is not yet a metric since it can be negative (e.g., $b = -a$, then $\mathbf{d}_{\cos}(a,b) = -1$).

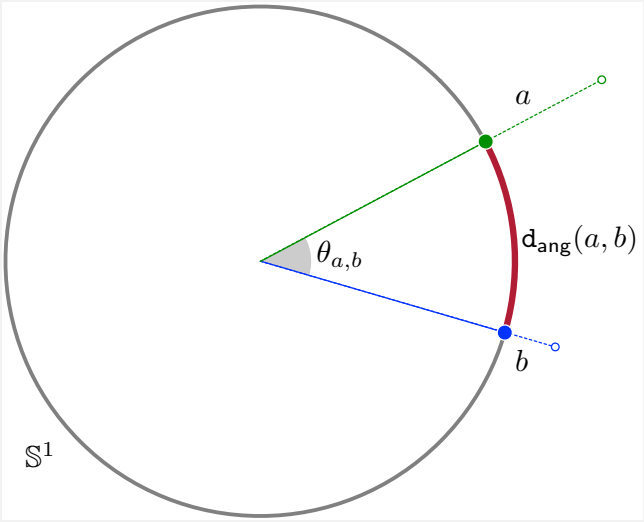
A simple isometric transformation of the cosine distance (this means the ordering between any pairs of points are the same) is called the *angular distance* $\mathbf{d}_{\mathsf{ang}}$; that is for $a, b, c, d \in \mathbb{S}^{d-1}$ if $\mathbf{d}_{\cos}(a,b) < \mathbf{d}_{\cos}(c,d)$ then $\mathbf{d}_{\mathsf{ang}}(a,b) < \mathbf{d}_{\mathsf{ang}}(c,d)$. Specifically we define for any $a, b \in \mathbb{S}^{d-1}$

$$\mathbf{d}_{\mathsf{ang}}(a,b) = \cos^{-1}(\langle a, b\rangle) = \arccos(\langle a, b\rangle).$$

That is, this undoes the cosine-interpretation of the cosine distance, and only measures the angle. Since, the inner product for any $a, b \in \mathbb{S}^{-1}$ is in the range $[-1, 1]$, then the value of $\mathbf{d}_{\cos}$ is in the range $[-1, 1]$, but the value of $\mathbf{d}_{\mathsf{ang}}(a,b) \in [0,\pi]$. Moreover, $\mathbf{d}_{\mathsf{ang}}$ is a metric over $\mathbb{S}^{d-1}$.

Geometry of Angular Distance

The angular interpretation of the cosine distance $\mathbf{d}_{\cos}(a,b) = 1 - \cos(a,b)$ and angular distance $\mathbf{d}_{\mathsf{ang}}(a,b) = \theta_{a,b}$ is convenient to think of for points on the sphere. To understand the geometry of the angular distance between two points, it is sufficient to consider them as lying on the unit sphere $\mathbb{S}^1 \subset \mathbb{R}^2$. Now $\mathbf{d}_{\mathsf{ang}}(a,b)$ is the radians of the angle $\theta_{a,b}$, or equivalently the arclength traveled between a and b if walking on the sphere.



Now we can see why $\mathbf{d}_{\mathsf{ang}}$ is a metric if restricted to vectors on $\mathbb{S}^{d-1}$. (M1) and (M3) hold by definition, and (M2) on $\mathbb{S}^{d-1}$ holds because no two distinct unit vectors have and angle of 0 radians between them. To show the triangle inequality (M4) (here we need to think in $\mathbb{S}^2 \subset \mathbb{R}^3$), observe that since $\mathbf{d}_{\mathsf{ang}}$ measures the shortest distance restricted to the sphere, there is no way that $\mathbf{d}_{\mathsf{ang}}(a,b)$ can be longer than $\mathbf{d}_{\mathsf{ang}}(a,c) + \mathbf{d}_{\mathsf{ang}}(c,b)$ since that would imply going through point $c \in \mathbb{S}^{d-1}$ makes the path from a to b shorter – which is not possible.

4.2.4 KL Divergence

The Kullback-Liebler Divergence (or KL Divergence) is a distance that is *not* a metric. Somewhat similar to the cosine distance, it considers as input discrete distributions X_a and X_b; they are d-variate random variables which can be instantiated at one of d possible values. Equivalently, it is useful to think of these objects as vectors $a, b \in \Delta_\circ^{d-1}$. Like the $(d-1)$-dimensional sphere, the $(d-1)$-dimensional open simplex $\Delta_\circ^{d-1}$ is a bounded subset of $\mathbb{R}^d$. Specifically

$$\Delta_\circ^{d-1} = \left\{ x = (x_1, x_2, \ldots x_d) \in \mathbb{R}^d \mid \|x\|_1 = 1 \text{ and } x_i > 0 \text{ for all } i \right\}.$$

That is $\Delta_\circ^{d-1}$ defines the set of d-dimensional discrete probability distributions where for $a \in \Delta^{d-1}$, the coordinate a_i is the probability that X_a takes the ith value. The $(d-1)$-dimensional (closed) simplex Δ^{d-1} differs in that it also allows values a_i to be 0, i.e., to have 0 probability to take the ith value. But the KL Divergence is only defined over vectors on $\Delta_\circ^{d-1}$.

Then we can define (often written $\mathbf{d}_{KL}(X_a \| X_b)$)

$$\mathbf{d}_{KL}(X_a, X_b) = \mathbf{d}_{KL}(a,b) = \sum_{i=1}^{d} a_i \ln(a_i/b_i).$$

It is reminiscent of entropy, and can be written as $H(X_a, X_b) - H(X_a)$ where $H(X)$ is the entropy of X, and $H(X_a, X_b)$ is the cross entropy. It roughly describes the extra bits needed to express a distribution X_a, given the knowledge of distribution X_b.

Note that $\mathbf{d}_{KL}$ is *not* a metric, violating (M3) since it is not symmetric. It also violates the triangle inequality (M4).

4.3 Distances for Sets and Strings

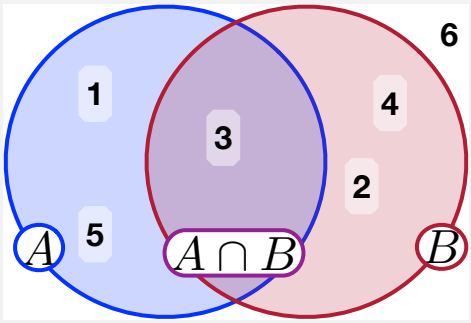
We now introduce some more general notions of distance, in particular ones that are heavily used to understand the relationship between text data: strings of words or characters. There are other techniques which draw more heavily on the semantic and fine-grained structural properties of text. We focus here on the ones which have simple mathematical connections and as a result are often more scalable and flexible.

Quick review of sets. A set A is a collection of elements $\{a_1, a_2, \ldots, a_k\}$. It is standard to use curly brackets $\{\ldots\}$ to denote a set. The ordering of elements does not matter, e.g., $\{a_1, a_2\} = \{a_2, a_1\}$. Multiplicity is not allowed or is ignored, e.g., $\{a_1, a_1, a_2\} = \{a_1, a_2\}$; if it is considered, then it is called a *multiset* (these more naturally models counts, or after normalization probability distributions (see cosine distance and KL divergence).

Given a set A, the *cardinality* of A denoted $|A|$ counts how many elements are in A. The *intersection* between two sets A and B is denoted $A \cap B$ and reveals all items which are in *both* sets. The *union* between two sets A and B is denoted $A \cup B$ and reveals all items which are in *either* set. The *set difference* of two sets A and B is denoted $A \setminus B$ and is all elements in A which are not in B. The *symmetric distance* between two sets A and B is denoted $A \triangle B$ and is the union of $A \setminus B$ and $B \setminus A$. Given a domain Ω (which contains all sets of interest, i.e., $A \subset \Omega$), the *complement* of a set A is defined as $\bar{A} = \Omega \setminus A$.

Example: Set operations

Observe the example with $A = \{1, 3, 5\}$ and $B = \{2, 3, 4\}$. These are represented as a the Venn diagram with a blue region for A and a red one for B. Element 6 is in neither set. Then the cardinality of A is $|A| = 3$, and it is the same for B. The intersection $A \cap B = \{3\}$ since it is the only object in both sets, and is visually represented as the purple region. The union $A \cup B = \{1, 2, 3, 4, 5\}$. The set difference $A \setminus B = \{1, 5\}$ and $B \setminus A = \{2, 4\}$. The complement $\bar{A} = \{2, 4, 6\}$ and the complement $\overline{A \cup B} = \{6\}$. Finally, the symmetric difference is $A \triangle B = \{1, 2, 4, 5\}$.



4.3.1 Jaccard Distance

The Jaccard distance between two sets A and B is defined

$$\mathbf{d}_{\mathbf{J}}(A, B) = 1 - \frac{|A \cap B|}{|A \cup B|}.$$

Since the union is always larger than the intersection, the fractional term is always between 0 and 1, and this the distance always takes value in $[0, 1]$. It is only 1 if the sets have nothing in common, and is only 0 if the intersection equals the union and the sets are exactly the same.

Example: Jaccard Distance

Consider two sets $A = \{0, 1, 2, 5, 6\}$ and $B = \{0, 2, 3, 5, 7, 9\}$. The *Jaccard distance* between A and B is

$$\begin{aligned} \mathbf{d}_{\mathbf{J}}(A, B) &= 1 - \frac{|A \cap B|}{|A \cup B|} \\ &= 1 - \frac{|\{0, 2, 5\}|}{|\{0, 1, 2, 3, 5, 6, 7, 9\}|} = 1 - \frac{3}{8} = 0.625 \end{aligned}$$

Notice that if we add an element 7 to A (call this set A') that is already in B, then the numerator increases, but the denominator stays the same. So then $\mathbf{d}_J(A', B) = 1 - \frac{4}{8} = 0.5$ and the distance is smaller – they are closer to each other.
On the other hand, if we add an element 4 to A (call this set A'') which is in neither A or B, then the numerator stays the same, but the denominator increases. So then $\mathbf{d}_J(A'', B) = 1 - \frac{3}{9} \approx 0.666$ and then distance is larger – the sets are further from each other.

The Jaccard distance is a popular distance between sets since it is a metric, and it is invariant to size of the sets. It only depends on the fraction of the items among both sets which are the same. It also does not require knowledge of some larger universe of elements that the sets may be from. For instance, as in the example, we can implicitly require that the sets contain only positive integers, but do not need to know an upper bound on the largest positive integer allowed.

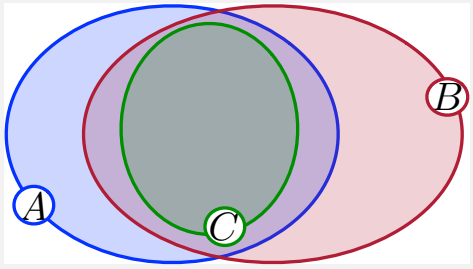
Geometry of Metric Property of Jaccard Distance

To show that $\mathbf{d}_\mathsf{J}$ is a metric, we need to show that the 4 properties each hold. The first three are direct. In particular (M1) and (M2) follow from $\mathbf{d}_\mathsf{J}(A, B) \in [0, 1]$ and only being 0 if $A = B$. Property (M3) holds by the symmetry of set operations $\cap$ and $\cup$.

The triangle inequality (M4) requires a bit more effort to show, namely for any sets A, B, C that $\mathbf{d}_\mathsf{J}(A, C) + \mathbf{d}_\mathsf{J}(C, B) \geq \mathbf{d}_\mathsf{J}(A, B)$. We will use the notation that

$$\mathbf{d}_\mathsf{J}(A, B) = 1 - \frac{|A \cap B|}{|A \cup B|} = \frac{|A \triangle B|}{|A \cup B|}.$$

We first rule out that there are elements $c \in C$ which are not in A or not in B. Removing these elements from C will only decrease the left-hand-side of the triangle inequality while not affecting the right-hand-side. So if C violates this inequality, we can assume there are no such $c \in C$ and it will still violate it. So now we assume $C \subseteq A$ and $C \subseteq B$.



Now we have

$$\begin{aligned}
\mathbf{d}_\mathsf{J}(A, C) + \mathbf{d}_\mathsf{J}(C, B) &= \frac{|A \setminus C|}{|A|} + \frac{|B \setminus C|}{|B|} \\
&\geq \frac{|A \setminus C| + |B \setminus C|}{|A \cup B|} \\
&\geq \frac{|A \triangle B|}{|A \cup B|} = \mathbf{d}_\mathsf{J}(A, B).
\end{aligned}$$

The first inequality follows since $|A|, |B| \leq |A \cup B|$. The second inequality holds since anything taken out from A or B would be in $A \cup B$ and thus would not affect $A \triangle B$; it is only equal if $C = A \cup B$, and $A \triangle B = \emptyset$.

4.3.2 Edit Distance

Let Σ be a set, in this case an alphabet of possible characters (e.g., all ASCII characters, or all lowercase letters so $\Sigma = \{a, b, \ldots, z\}$). Then we can say a string a of length d is an element in Σ^d; that is an ordered sequence of characters from the alphabet Σ. The *edit distance* considers two strings $a, b \in \Sigma^d$, and

$$\mathbf{d}_\mathsf{ed}(a, b) = \text{\# operations to make } a = b,$$

where an operation can delete a letter or insert a letter. In fact, the strings are not required to have the same length, since we can insert items in the shorter one to make up the difference.

Example: Edit Distance

Consider two strings $a =$ `mines` and $b =$ `smiles`. Here $\mathbf{d}_{\text{ed}}(a, b) = 3$.

`mines`
1 : `minles` insert `l`
2 : `miles` delete `n`
3 : `smiles` insert `s`

There are many alternative variations of operations. The insert operation may cost more than the delete operation. Or we could allow a replace operation at the same unit cost as either insert or delete; in this case the edit distance of `mines` and `smiles` is only 2.

Edit distance is a metric. (M1) holds since the number of edits is always non-negative. (M2) There are no edits only if they are the same. (M3) the operations can be reversed. (M4) If c is an intermediate "word" then the $\mathbf{d}_{\text{ed}}(a, c) + \mathbf{d}_{\text{ed}}(c, b) = \mathbf{d}_{\text{ed}}(a, b)$, otherwise it requires more edits.

Is this good for large text documents? Not really. It is slow to compute – basically requiring quadratic time dynamic programing in th worst case to find the smallest set of edits. And removing one sentence can cause a large edit distance without changing meaning. But this *is* good for small strings. Some version is used in most spelling recommendation systems (e.g., a search engine's auto-correct). Its a good guide that usually $\mathbf{d}_{\text{ed}}(a, b) > 3$ is pretty large since, e.g., with replace $\mathbf{d}_{\text{ed}}(\texttt{cart}, \texttt{score}) = 4$.

4.4 Modeling Text with Distances

There are many many choices of distances. Which one to *choose* (it is definitely a choice) is based on (a) computational efficiency and (b) modeling effectiveness. The Euclidean distance $\mathbf{d}_2$ and sometimes the Jaccard distance $\mathbf{d}_J$ are often chosen because various algorithmic and computational benefits area available for these – as we will see, this efficiency comes not just from time to compute the distance once, but how it can be used within more complex operations. However, each have other benefits due to modeling factors. Sometimes this modeling is just based on mathematical properties (is it a metric, are my vectors normalized), sometimes it is intuitive, and sometimes it can be empirically validated by measuring performance on downstream applications. In this section we show how to arrive at various of the distances as the logical choice in an example case in modeling text.

As mentioned, edit distance $\mathbf{d}_{\text{ed}}$ is useful for shorter strings. The other variants will all be more useful when dealing with much larger texts.

Example: Running text example

In this section we will use as a running example, the text from the following 4 short documents. In practice, these approaches are typically applied to much longer documents (e.g., text on a webpage, a newspaper article, a persons bio).

D_1 : `I am Sam.`
D_2 : `Sam I am.`
D_3 : `I do not like jelly and ham.`
D_4 : `I do not, do not, like them, Sam I am.`

How can we measure the distance among these 4 documents?

4.4.1 Bag-of-Words Vectors

The simplest model for converting text into an abstract representation to applying a distance is the bag-of-words approach. Intuitively, each document creates a "bag" and throws each word in that "bag" (a multi-set data structure), and maintains only the count of each word. This transforms each document into a multi-set. However it is convenient to think of it as a (very sparse, meaning mostly 0s) vector.

That is, consider a vector $v \in \mathbb{R}^D$ for a very large D, where D is the number of all possible words. Each coordinate of such a vector corresponds to one word, and records the count of that word.

These vector representation naturally suggests that one could use an L_p distance, most commonly $\mathbf{d}_2$, to measure their distance. However, it is more common to use the cosine distance $\mathbf{d}_{\cos}$. This has the advantage of not penalizing certain documents for their length, in principle focusing more on its content. For instance, a document with a simple phrase would be identical under $\mathbf{d}_{\cos}$ to another document that repeated that phrase multiple times. Or two documents about, say, baseball would typically draw from a similar set of words (e.g., {`bat`, `ball`, `hit`, `run`, `batter`, `inning`}) and likely be close even if their lengths differs.

Example: Bag-of-Words

For the running example, consider D-dimensional space with $D = 11$; it could be much higher. For each coordinate, we list the corresponding word as

(`am`, `and`, `do`, `ham`, `I`, `jelly`, `like`, `not`, `Sam`, `them`, `zebra`).

Now each of the documents have the following representative vectors

$$v_1 = (1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0)$$
$$v_2 = (1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0)$$
$$v_3 = (0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0)$$
$$v_4 = (1, 0, 2, 0, 2, 0, 1, 2, 1, 1, 0).$$

We can now use the Euclidean distance $\mathbf{d}_2$ between these vectors (or any other L_2 distance). We notice that $\mathbf{d}_2(v_1, v_2) = 0$, even though the text is different. However, since the bag-of-words only measures which words are present, not how they are used, it can not distinguish between these cases.

Also notice that the 11th coordinate for the work `zebra` is never used, and is 0 in all coordinates of the vectors. If this coordinate was omitted and only a 10-dimensional representation was used it would not change any of the distances. On the other hand, these vectors can be much larger and represent many other words, and this will not affect the distance.

	(v_1, v_2)	(v_1, v_3)	(v_1, v_4)	(v_2, v_3)	(v_2, v_4)	(v_3, v_4)
$\mathbf{d}_2$	0	2.83	3.32	2.83	3.32	3
$\mathbf{d}_{\cos}$	0	0.781	0.423	0.781	0.423	0.339

Alternatively, we can use the cosine distance $\mathbf{d}_{\cos}$. This models the text differently, the main difference is that it normalizes the vectors. So for instance D_3 and D_4 are not likely to be further apart simply because they contain more words, as is the case with $\mathbf{d}_2$. This metric still treats D_1 and D_2 as identical.

Another potential use of the bag-of-words representation is as a probability distribution over the words in a document. That is, again consider a D-dimensional space where D is the number of possible words. But now for each word w_i, its representative coordinate (the ith coordinate) denotes the probability that a

randomly drawn word from the document is that word. This is equivalent to normalizing a vector v by its L_1 norm so $\bar{v} = v/\|v\|_1$, and this ensures that $v \in \Delta^{D-1}$. This representation inclines a modeler to use a distance designed for probability distributions, such as the KL Divergence $\mathbf{d}_{KL}$.

Example: KL Divergence on Probability Measures

Using the same 11-dimensional representation of our running example documents, we can normalize by L_1 the vectors v_1, v_2, v_3, and v_4 so they lie on $\Delta_\circ^{10}$. However, this does not quite work, since the vectors have 0 values, which are not allowed in $\Delta_\circ^{10}$. The right way to address this (if we insist on using KL Divergence) is to add a so-called regularization term α to each vector value before L_1 normalization. Using $\alpha = 0.01$ we obtain

$$\bar{v}_1 = (0.325, 0.003, 0.003, 0.003, 0.325, 0.003, 0.003, 0.003, 0.325, 0.003, 0.003)$$
$$\bar{v}_2 = (0.325, 0.003, 0.003, 0.003, 0.325, 0.003, 0.003, 0.003, 0.325, 0.003, 0.003)$$
$$\bar{v}_3 = (0.001, 0.142, 0.142, 0.142, 0.142, 0.142, 0.142, 0.142, 0.001, 0.001, 0.001)$$
$$\bar{v}_4 = (0.100, 0.001, 0.199, 0.001, 0.199, 0.001, 0.100, 0.199, 0.100, 0.100, 0.001)\,.$$

The KL divergence is not symmetric, so we report the distance in both directions for all pairs of vectors, where the row label is the first term and the column label is the second term (e.g., $\mathbf{d}_{KL}(\bar{v}_2||\bar{v}_4) = 0.89$).

$\mathbf{d}_{KL}$	$\bar{v}_1$	$\bar{v}_2$	$\bar{v}_3$	$\bar{v}_4$
$\bar{v}_1$	0	0	3.74	0.89
$\bar{v}_2$	0	0	3.74	0.89
$\bar{v}_3$	3.09	3.09	0	2.01
$\bar{v}_4$	1.99	1.99	1.43	0

```
import numpy as np
from numpy import linalg as LA
import scipy as sp
from scipy import stats

#create representative vectors
v1 = np.array([1,0,0,0,1,0,0,0,1,0,0])
v2 = np.array([1,0,0,0,1,0,0,0,1,0,0])
v3 = np.array([0,1,1,1,1,1,1,1,0,0,0])
v4 = np.array([1,0,2,0,2,0,1,2,1,1,0])

#Euclidean distance
print [LA.norm(v1-v2), LA.norm(v1-v3), LA.norm(v1-v4), LA.norm(v2-v3),
        LA.norm(v2-v4), LA.norm(v3-v4)]
# [0.0, 2.8284271247461903, 3.3166247903553998, 2.8284271247461903,
#         3.3166247903553998, 3.0]

#normalized vectors
v1n = v1/LA.norm(v1)
v2n = v2/LA.norm(v2)
v3n = v3/LA.norm(v3)
v4n = v4/LA.norm(v4)

#Cosine distance
print [1 - v1n.dot(v2n), 1-v1n.dot(v3n), 1-v1n.dot(v4n), 1-v2n.dot(v3n),
        1-v2n.dot(v4n), 1-v3n.dot(v4n)]
```

```
# [-2.2204460492503131e-16, 0.78178210976400764, 0.42264973081037416,
#       0.78178210976400764, 0.42264973081037416, 0.33856217223385243]

#regularizer requied for KL (it cannot handle 0 terms)
reg = 0.01
v1r = v1+reg
v2r = v2+reg
v3r = v3+reg
v4r = v4+reg

# KL-divergence (the entropy function L1-normalizes vectors internally)
print [stats.entropy(v1r,v1r), stats.entropy(v1r,v2r), stats.entropy(v1r,v3r),
       stats.entropy(v1r,v4r)]
print [stats.entropy(v2r,v1r), stats.entropy(v2r,v2r), stats.entropy(v2r,v3r),
       stats.entropy(v2r,v4r)]
print [stats.entropy(v3r,v1r), stats.entropy(v3r,v2r), stats.entropy(v3r,v3r),
        stats.entropy(v3r,v4r)]
print [stats.entropy(v4r,v1r), stats.entropy(v4r,v2r), stats.entropy(v4r,v3r),
       stats.entropy(v4r,v4r)]
# [0.0, 0.0, 3.735444216340686, 0.89162425422773228]
# [0.0, 0.0, 3.735444216340686, 0.89162425422773228]
# [3.0937010620836185, 3.0937010620836185, 0.0, 2.0060541317114353]
# [1.9887613727596174, 1.9887613727596174, 1.4279093007498529, 0.0]
```

4.4.2 k-Grams

As an alternative to bag-of-word vectors, k-grams can provide a richer context for text. These convert a document into a set (not a multi-set), but does not use just the words.

There are a few variants of how to apply k-grams, and we will focus on two common and simple versions over words and characters. The *word k-grams* over a document is the set of all witnessed consecutive sets of k words.

Ethical Questions with Modeling Text

After a mechanism to transform a text document or a webpage into an abstract data type (e.g., a vector or a set) os implemented and a choice of distance is made, then many downstream algorithms and analysis can be applied. And these applications can be fairly automatically invoked and with little or no modification based on the modeling decisions. Then these can be used to make recommendations about jobs, loans, and education opportunities.

Consider now you are processing essays to aid in college admission decisions, and you change a modeling choice from bag-of-words and cosine distance to k-grams and Jaccard distance. Then you realize that this change will sort admission applications differently, and a class of applicants will likely have their admission decisions changed because of this modeling. Do you have an obligation to those applicants to keep the decision the same? How could you alert decision makers to this effect, and what resolutions could you make knowing that this happens?

Example: Word 2-Grams

Using the running example, we show the 2-word-grams of each of the documents. Each gram is shown as a set of two words in square brackets.

D_1: $G_1 = \{$`[I am], [am Sam]`$\}$
D_2: $G_2 = \{$`[Sam I], [I am]`$\}$
D_3: $G_3 = \{$`[I do], [do not], [not like], [like jelly], [jelly and] [and ham]`$\}$
D_4: $G_4 = \{$`[I do], [do not], [not do], [not like], [like them], [them Sam], [Sam I], [I am]`$\}$

In particular, note that in D_4 that `[do not]` $\neq$ `[not do]` so both appear in G_4; however, that even though the two-word sequence `[do not]` appears twice in D_4, it only appears once in the set G_4.

We can then compute the Jaccard distance between these sets as representations of the distances between documents.

	(D_1, D_2)	(D_1, D_3)	(D_1, D_4)	(D_2, D_3)	(D_2, D_4)	(D_3, D_4)
$\mathbf{d_J}$	$1 - \frac{1}{3}$ ≈ 0.667	$1 - \frac{0}{8}$ $= 1$	$1 - \frac{1}{9}$ ≈ 0.889	$1 - \frac{0}{8}$ $= 1$	$1 - \frac{2}{8}$ $= 0.75$	$1 - \frac{3}{11}$ ≈ 0.727

Variants and modeling choices. There are many variants of how to constrict k-grams for words. The most prominent of which is to use k consecutive characters instead of k consecutive words; we call these *character k-grams*.

Many other modeling decisions go into constructing a k-gram. Should punctuation be included? Should whitespace be used as a character, or should sentence breaks be used as a word in word grams? Should differently capitalization characters or words represent distinct objects? And most notoriously, how large should k be?

Example: Character k-Grams

Characters k grams for sample document D_4, ignoring whitespace, punctuation, and capitalization.
Characters $k = 3$:
`{[ido], [don], [ono], [not], [otd], [tdo], [otl], [tli], [lik], [ike], [ket], [eth], [the], [hem], [ems], [msa], [sam], [ami], [mia], [iam]}`

Characters $k = 4$:
`{[idon], [dono], [onot], [notd], [otdo], [tdon], [notl], [otli], [tlik], [like], [iket], [keth], [ethe], [them], [hems], [emsa], [msam], [sami], [amia], [miam]}`

Through all of these variants, a few common rules apply. First, the more expressive the k-grams (e.g., keeping track of punctuation, capitalization, and whitespace), the large the quantity of data that is required to get meaningful results – otherwise most documents will have a Jaccard distance 1 or very close to it unless large blocks of text are verbatim repeated (for instance, plagiarism). But for long and structured articles

(e.g., newspaper articles with rigid style guide), some of these more expressive choices can be effective.

Second, for longer articles it is better to use words and larger values of k, while for shorter articles (like tweets) it is better to use smaller k and possibly character k-grams. The values used for k are often perhaps-surprisingly short, such as $k = 3$ or $k = 4$ for both characters and words.

Finally, there are a few structured tricks which come up. It can be useful to particularly keep track of or emphasize starts of sentences, capitalized words, or word k-grams which start with "stop words" (that is, very common words like {`a`, `for`, `the`, `to`, `and`, `that`, `it`, ...} that often signal starts of expressions).

Continuous Bag-of-Words. As an aside, we mention a more recent trend. Instead of creating abstract representations of entire documents, we may use many such documents to create similar representations for each word, based on how it appears in the union of the documents. These methods typically aim for a Euclidian representation for words. For a set of words, these are called *word vector embeddings*. Then the distance between the abstract representations of these words can be shown to carry a variety of information.

There are numerous of these techniques, but at a high-level, they start by creating a bag-of-word representation for each instance of a word. That is within some sequential radius of r words in a text, for each instance of a word, it creates an D-dimensional vector (where we maintain counts over a size D vocabulary). This is like the $D = 11$ words in the running example, but typically D is tens or hundreds of thousands. This vector representation of each word is sparse, and counts the multiplicity of each other word in its neighbor. Then these vectors are averaged over all instances of each word. This is called the *continuous bag-of-words* or CBOW model. Usually these representations are non-linearly compressed to a lower-dimensional representation, often using 200 or 300 dimensions.

Ethical Questions with using Abstract Language Models

These word vectors embedding have led to dramatic improvements for various natural language processing tasks such as translation between languages and comparing meaning at sentence level structures. As a result, they are now commonly incorporated into state-of-the-art text models. However, they have also been shown to implicitly encode bias into the embeddings, coming from the text corpuses on which they were built. For instance, the work `man` is significantly closer to the word `engineer` than is the word `woman`. Such biases, even if unintentional, may affect automated hiring decisions based on gender. As a data analyst, should you use these models to help make predictions if they are known to include biases, even if they actual lead to better prediction and generalization?

4.5 Similarities

A notion that is dual to a distance $\mathbf{d}$, is a similarity $\mathbf{s}$. It is still a bivariate function, but when the arguments are close, then the value should be large. Often, but not always, the definitions are normalized so the similarity of an object with itself is $\mathbf{s}(A, A) = 1$, and so the range of the function is in $[0, 1]$, indicating that pairs of values A, B which are totally different have value $\mathbf{s}(A, B) = 0$.

There are two standard ways to convert from a similarity to a distance. For set-based similarities, it is common to obtain a distance as $\mathbf{d}(A, B) = 1 - \mathbf{s}(A, B)$. In vectorized and norm-based similarities, it is common to obtain a distance as $\mathbf{d}(A, B) = \sqrt{\mathbf{s}(A, A) + \mathbf{s}(B, B) - 2\mathbf{s}(A, B)}$.

4.5.1 Normed Similarities

The most basic normed similarity is the Euclidian dot product. For two vectors $p, q \in \mathbb{R}^d$, then we can define the *dot product similarity* as

$$\mathbf{s}_{\mathsf{dot}}(p, q) = \langle p, q \rangle;$$

that is, just the dot product. Any indeed this converts to the Euclidian distance as

$$\begin{aligned}\mathbf{d}_2(p,q) = \|p-q\|_2 &= \sqrt{\mathbf{s}_{\mathsf{dot}}(p,p) + \mathbf{s}_{\mathsf{dot}}(q,q) - 2\mathbf{s}_{\mathsf{dot}}(p,q)} \\ &= \sqrt{\langle p,p\rangle + \langle q,q\rangle - 2\langle p,q\rangle} \\ &= \sqrt{\|p\|_2^2 + \|q\|_2^2 - 2\langle p,q\rangle}.\end{aligned}$$

However, in this case the similarity could be arbitrarily large, and indeed the similarity of a vector p with itself $\mathbf{s}_{\mathsf{dot}}(p,p) = \langle p,p\rangle = \|p\|^2$ is the squared Euclidean norm. To enforce that the similarity is at most 1, we can normalize the vectors first, and we obtain the *cosine similarity*

$$\mathbf{s}_{\cos}(p,q) = \langle \frac{p}{\|p\|}, \frac{q}{\|q\|}\rangle = \frac{\langle p,q\rangle}{\|p\|\|q\|}.$$

Converting to a distance via the normed transformation:

$$\sqrt{\mathbf{s}_{\cos}(p,p) + \mathbf{s}_{\cos}(q,q) - 2\mathbf{s}_{\cos}(p,q)} = \mathbf{d}_2\Big(\frac{p}{\|p\|}, \frac{q}{\|q\|}\Big)$$

is still the Euclidian distance, but now between the normalized vectors $\frac{p}{\|p\|}$ and $\frac{q}{\|q\|}$. However, for this similarity, it is more common to instead use the set transformation to obtain cosine distance:

$$\mathbf{d}_{\cos}(p,q) = 1 - \frac{\langle p,q\rangle}{\|p\|\|q\|} = 1 - \mathbf{s}_{\cos}(p,q).$$

Another family of similarities in this class are from kernels. The most common of which is the Gaussian kernel $K(p,q) = \exp(-\|p-q\|^2)$. Then the *kernel distance* is defined

$$\mathbf{d}_K(p,q) = \sqrt{K(p,p) + K(q,q) - 2K(p,q)}.$$

For Gaussian kernels, and a larger class called *characteristic kernels* (a subset of positive definite kernels), this distance is a metric.

4.5.2 Set Similarities

Given two sets A and B, the *Jaccard similarity* is defined

$$\mathbf{s}_{\mathsf{J}}(A,B) = \frac{|A \cap B|}{|A \cup B|}.$$

Indeed the Jaccard distance is defined $\mathbf{d}_{\mathsf{J}}(A,B) = 1 - \mathbf{s}_{\mathsf{J}}(A,B)$.

To generalize set similarities (at least those that are amenable to large-scale techniques) we consider a class of similarities which can be written in the following form based on parameters x, y, z, z':

$$\mathbf{s}_{x,y,z,z'}(A,B) = \frac{x|A \cap B| + y|\overline{A \cup B}| + z|A \triangle B|}{x|A \cap B| + y|\overline{A \cup B}| + z'|A \triangle B|}.$$

For instance $\mathbf{s}_{\mathsf{J}} = \mathbf{s}_{1,0,0,1}$. Note that this family includes a complement operation of $\overline{A \cup B}$ and thus seems to requires to know the size of the entire domain Ω from which A and B are subsets. However, in the case of Jaccard similarity and others when $y = 0$, this term is not required, and thus the domain Ω is not required to be defined.

Other common set similarities in this family include the following.

$$\text{Hamming:} \quad \mathbf{s}_{\mathsf{Ham}}(A,B) = \mathbf{s}_{1,1,0,1}(A,B) = \frac{|A \cap B| + |\overline{A \cup B}|}{|A \cap B| + |\overline{A \cup B}| + |A \triangle B|} = 1 - \frac{|A \triangle B|}{|\Omega|}$$

$$\text{Andberg:} \quad \mathbf{s}_{\mathsf{Andb}}(A,B) = \mathbf{s}_{1,0,0,2}(A,B) = \frac{|A \cap B|}{|A \cap B| + 2|A \triangle B|} = \frac{|A \cap B|}{|A \cup B| + |A \triangle B|}$$

$$\text{Rogers-Tanimoto:} \quad \mathbf{s}_{\mathsf{RT}}(A,B) = \mathbf{s}_{1,1,0,2}(A,B) = \frac{|A \cap B| + |\overline{A \cup B}|}{|A \cap B| + |\overline{A \cup B}| + 2|A \triangle B|} = \frac{|\Omega| - |A \triangle B|}{|\Omega| + |A \triangle B|}$$

$$\text{Søensen-Dice:} \quad \mathbf{s}_{\mathsf{Dice}}(A,B) = \mathbf{s}_{2,0,0,1}(A,B) = \frac{2|A \cap B|}{2|A \cap B| + |A \triangle B|} = \frac{2|A \cap B|}{|A| + |B|}$$

For $\mathbf{s}_{\mathsf{J}}$, $\mathbf{s}_{\mathsf{Ham}}$, $\mathbf{s}_{\mathsf{Andb}}$, and $\mathbf{s}_{\mathsf{RT}}$, then $\mathbf{d}(A,B) = 1 - \mathbf{s}(A,B)$ is a metric. In particular, $\mathbf{d}_{\mathsf{Ham}}(A,B) = |\Omega|(1 - \mathbf{s}_{\mathsf{Ham}}(A,B))$ is known as the *Hamming distance*; it is typically applied between bit vectors. In the bit vector setting it counts the number of bits the vectors differ on. Indeed, if we represent each object $i \in \Omega$ by a coordinate b_i in $|\Omega|$-dimension vector $b = (b_1, b_2, \ldots, b_{|\Omega|})$ then these notions are equivalent.

Example: Set Similarities

Consider two sets $A = \{0,1,2,5,6\}$ and $B = \{0,2,3,5,7,9\}$ in a domain $\Omega = \{0,1,2,3,4,5,6,7,8,9\}$. We can compute each of our defined set similarities between these sets:

Jaccard:	$\mathbf{s}_{\mathsf{J}}(A,B) = \frac{3}{8} = 0.375$
Hamming:	$\mathbf{s}_{\mathsf{Ham}}(A,B) = 1 - \frac{5}{10} = 0.5$
Andberg:	$\mathbf{s}_{\mathsf{Andb}}(A,B) = \frac{3}{13} \approx 0.231$
Rogers-Tanimoto:	$\mathbf{s}_{\mathsf{RT}}(A,B) = \frac{10-5}{10+5} \approx 0.333$
Sørensen-Dice:	$\mathbf{s}_{\mathsf{Dice}}(A,B) = \frac{2(3)}{5+6} \approx 0.545$

4.6 Locality Sensitive Hashing

This chapter has surveyed numerous different distances and similarities, which are themselves a small subset of all distances and similarities one may consider when modeling data and an analysis task. While metric properties and other nice mathematical properties are useful, another key concern is the computational cost associated with using the distances and similarities. This is also not just the case of a single evaluation, but when comparing a very large set of objects (say of $n = 100$ million objects). Then how can one quickly determine which ones are close, or given a query object q, which ones in the large set are close to this query?

In 1-dimensional Euclidean data, these problems are relatively simple to address. Start by sorting all objects, and storing them in sorted order. Then the nearby objects are the adjacent ones in the sorted order. And if this ordering induces a balanced binary tree, then on a query, nearby objects can be found in logarithmic running time (i.e., time proportional to $\log n$).

Next we introduce "locality sensitive hashing" to address these question for higher-dimensional vectorized objects and for metrics on set-based representations. The main idea is for a family of objects $\mathcal{B}$, and a similarity $\mathbf{s} : \mathcal{B} \times \mathcal{B} \to [0,1]$, it defines a family of random hash functions $\mathcal{H}$ with roughly the following property, defined more precisely soonafter.

Locality Sensitive Hash Family: For a similarity function $\mathbf{s}$, a locality sensitive hash family $\mathcal{H}$ is a set of hash functions so for any two objects $p, q \in \mathcal{B}$ that

$$\mathbf{Pr}_{h \in \mathcal{H}}[h(p) = h(q)] \approx \mathbf{s}(p, q).$$

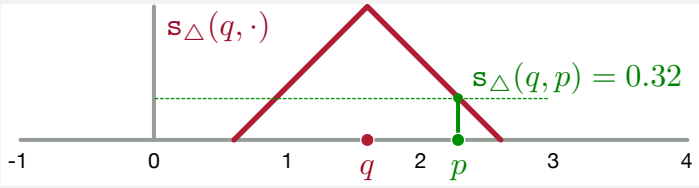
That is for any $p, q \in \mathcal{B}$, a randomly chosen hash function $h \in \mathcal{H}$ will cause those two objects to collide with probability roughly the same as their similarity.

Once a family is defined, it will be useful to randomly select a specific hash function from that family (e.g., the $h \in \mathcal{H}$ this choice is the random variable for the probability). However, once some function h is chosen, then it is a deterministic function.

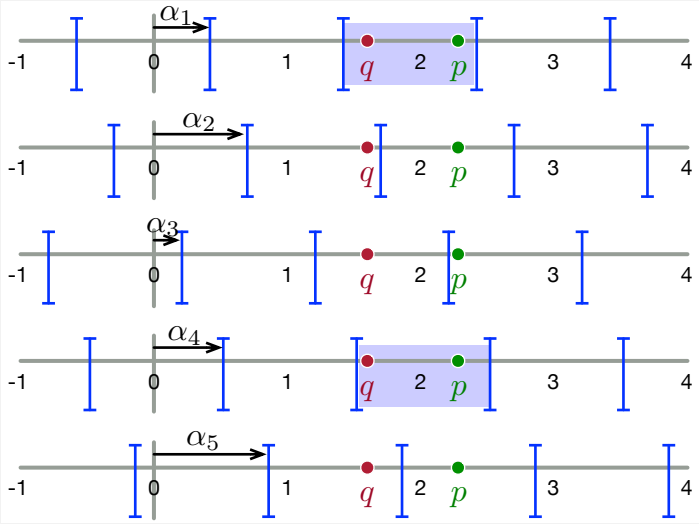
Example: Simple Gridding LSH for Triangle Similarity

Consider a 1-dimensional dataset $X \subset \mathbb{R}^1$, and the triangle similarity

$$\mathbf{s}_\triangle(p, q) = \max\{0, 1 - |p - q|\}.$$



A simple hash family for this is a set of randomly shifted grids, where items in the same grid cell hash together. Define $\mathcal{H}_\triangle = \{h_\eta \mid \eta \in [0, 1)\}$ where $h_\eta(x) = \lceil x + \eta \rceil$. That is each h_η maps the input to an integer (representing the index of a hash bucket) where a grid defines consecutive sets (intervals) of numbers of length 1 which are all mapped to the same hash bucket. The parameter α randomly shifts these grid boundaries. So whether or not a pair of points are in the same grid cell depends on the random choice of η, and more similar points are more likely to be in the same grid cell, and hence the same hash bucket.



The example above shows 5 hash functions $h_{\eta_1}, h_{\eta_2}, h_{\eta_3}, h_{\eta_4}, h_{\eta_5} \in \mathcal{H}_\triangle$. In this example p and q are in the same hash bucket for 2 of the 5 hash functions (for h_{η_1} and h_{η_4}), so we would estimate the similarity between p and q as $2/5 = 0.4$.

Indeed, we can verify that for any $p, q \in \mathbb{R}$ that

$$\mathbf{Pr}_{h_\eta \in \mathcal{H}_\triangle}[h_\eta(p) = h_\eta(q)] = \mathbf{s}_\triangle(p, q).$$

For any p, q with $|p - q| > 1$, both the probability and the similarity is 0; and if $p = q$, then both the probability and similarity is 1. The probability that p and q are *not* hashed together is the probability a randomly shifted grid boundary falls between them, which is precisely $|p-q|$ given that $|p-q| \leq 1$. Hence the probability and similarity in this case are both $1 - |p - q|$, as desired.

A common and important data structure is a *hash table* that relies on a different type of hash func-

tion, which to distinguish it from an LSH, we call a *separating hash function*. These hash functions $h : \mathcal{B} \to \mathcal{U}$ maps an object $b \in \mathcal{B}$ into a fixed universe $\mathcal{U}$; typically $\mathcal{U}$ can be represented as a set of integers $\{0, 1, 2, \ldots, u-1\}$, representing indices of an array of size u. Hash tables are again defined with respect to a family $\mathcal{H}$, and we consider a random choice $h \in \mathcal{H}$. Given this random choice, then a *perfect* separating hash function guarantees that for any two $b, b' \in \mathcal{B}$ that $\mathbf{Pr}_{h\in\mathcal{H}}[h(b) = h(b')] = 1/u$.

It is important to distinguish these two data structures and types of hash functions. Separating hash functions are powerful and useful, but are *not* locality sensitive hash functions.

4.6.1 Properties of Locality Sensitive Hashing

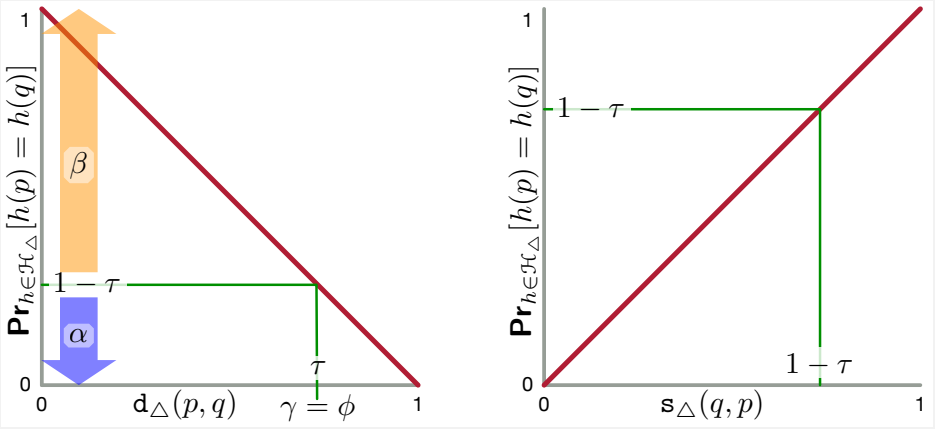
More generally, locality sensitive hash families are defined with respect to distance $\mathbf{d}$. A hash family $\mathcal{H}$ is $(\gamma, \phi, \alpha, \beta)$-*sensitive* with respect to $\mathbf{d}$ when it has the following properties:

- $\mathbf{Pr}_{h\in\mathcal{H}}[h(p) = h(q)] > \alpha$ if $\mathbf{d}(p, q) < \gamma$
- $\mathbf{Pr}_{h\in\mathcal{H}}[h(p) = h(q)] < \beta$ if $\mathbf{d}(p, q) > \phi$

For this to make sense we need $\alpha > 0$ and $\beta < 1$ for $\gamma \leq \phi$. Ideally we want α to be large, β to be small, and $\phi - \gamma$ to be small. Then we can repeat this with more random hash functions to *amplify* the effect using concentration of measure bounds; this can be used to make α larger and β smaller while fixing γ and ϕ. Ultimately, if α becomes close to 1, then we almost always hash items closer than γ in the same bucket, and as β becomes close to 0, then we almost never has items further than ϕ in the same bucket. Thus to define similar items (within a $\phi - \gamma$ tolerance) we simply need to look at which ones are hashed together.

Example: Sensitivity for Triangle Similarity

Revisit the triangle similarity $\mathbf{s}_\triangle$ and its associated grid-based hash family $\mathcal{H}_\triangle$. Recall for any objects p, q we have $\mathbf{Pr}_{h\in\mathcal{H}_\triangle}[h(p) = h(q)] = \mathbf{s}_\triangle(p, q)$. Then for some similarity threshold $\tau = \mathbf{d}_\triangle(p, q) = 1 - \mathbf{s}_\triangle(p, q)$ we can set $\tau = \gamma = \phi$ and have $\alpha = 1 - \tau$ and $\beta = \tau$.



This same setting (and picture) can be used for any threshold $\tau \in (0, 1)$ where the similarity, hash family pair $(\mathbf{s}, \mathcal{H})$ satisfies $\mathbf{Pr}_{h\in\mathcal{H}}[h(p) = h(q)] = \mathbf{s}(p, q)$.

This $(\gamma, \phi, \alpha, \beta)$-sensitivity condition is indeed more complicated than the $\mathbf{Pr}[h(a) = h(b)] = \mathbf{s}(a, b)$ condition. But it is more general, and any such separation will be all that is required to allow the LSH data structure to work, as elaborated on next. Moreover, this flexibility is required for certain important distances like Euclidean distance $\mathbf{d}_2$.

4.6.2 Prototypical Tasks for LSH

We want to do this to answer three types of tasks given a large dataset P (which could have size $|P|$ in the hundreds of millions):

- For a threshold τ, determine all pairs $p, p' \in P$ so that $\mathbf{d}(p, p') \leq \tau$ (or symmetrically, so that $\mathbf{s}(p, p') \geq \tau'$, e.g., with $\tau' = 1 - \tau$).
- For a threshold τ and a query q, return all $p \in P$ so that $\mathbf{d}(p, q) \leq \tau$.
- For a query q, return point $\tilde{p} \in P$ so that it is approximately $\text{argmin}_{p \in P} \mathbf{d}(p, q)$.

The first two tasks are important in deduplication and plagiarism detection. For instance, this task occurs when a search engine wants avoid returning two webpages which are very similar in content, or an instructor wants to quickly check if two students have turned in very similar assignments to each other, or very similar to one from an online solution repository.

In each case, we desire these tasks to take time roughly proportional to the number of items within the distance threshold τ. In very high-dimensional Euclidean space, or in the space of sets, a brute force checking of all distances would require time proportional to $|P|^2$ for the first task, and proportional to $|P|$ time for the second task. This can be untenable when for instance the size of $|P|$ is hundreds of millions.

The third task, as we will see in the classification section, is an essential tool for data analysis. If we know how the objects $p \in P$ in the database P behave, then we can guess that q will behave similarly to the closest known example $p^* = \text{argmin}_{p \in P} \mathbf{d}(p, q)$. We can solve for p^* by invoking the second task multiple times while doing a binary-like search on τ; if the threshold is too large, we short cut the operation of the second task as soon as we find more than a constant number of τ-close items.

The first task can also essentially be reduced to the second one, but just running the second task with each $p \in P$ as a query. So our description below will focus on the second task.

In all of these tasks, we will be able to solve, assuming we allow some approximation. We may not return the closest item, but one that is not too much further than the closest item. We may return some items slightly beyond our distance threshold, and many miss some which are barely closer than it. Given that the choice of distance or similarity used is a modeling choice (why not use a different one?), we should not take its exact value too seriously.

4.6.3 Banding to Amplify LSH

If we only use a single hash function $h \in \mathcal{H}$ where $h : \mathcal{B} \to U$, then we can design a hash table over the universe $U = \{0, 1, 2, \ldots, u - 1\}$. That is, if item $h(p) = j$, then we store it in a linked list L_j stored in location j in a length u array.

But to increase the probability that two similar items hash to the same location in the array, but non-similar items do not, we have two options: "Mama" bear's approach and "Papa" bear's approach. Papa bear will be too harsh, and Mama bear too generous, but together they can be used in an approach called banding – which is "Baby" bear's approach and it is just right.

Papa bear is harsh, and says items $p \in P$ are not close to q unless on multiple hash functions $h_1, h_2, \ldots, h_b$ *all* report that $h_i(p) = h_i(q)$. This can be handled efficiently with a hash table again by creating a banded-hash $H(p) : \mathcal{B} \to U^b$, that is it maps each object p to a b-dimensional vector, each element in U, where the ith coordinate is determined by the ith hash function. We can again create a hash table over these u^b possible entries (using standard tricks from data structures to avoid u^b space). The problem with Papa bear's approach is that it is too selective, and reduces the probability that any two items hash to the same bucket in the banded-hash table.

Mama bear is generous, and also considers multiple hash functions $h_1, h_2, \ldots, h_r$, and reports two items similar if they collide in *any* hash function. If we keep r separate hash tables which store the contents of P, then we can report the union of the items which collide with q. However, now the problem with Mama

bear's approach is that it is not selective enough, it increases the probability that any two items hash to the same bucket.

Banding (Baby bear's approach) puts these together (Baby bear takes after both parents after all). We create r banded hash functions H, with each composed of b hash functions. This requires a total of $r \cdot b$ hash functions, and a query to be hashed by all of these. Then if q intersects with *any* p in any banded hash table (which requires *all* hash functions to collide), we return it as similar.

Example: Banding

Consider $r = 3$ banded hash functions H_1, H_2, and H_3, where each banded hash function H_i has $b = 2$ hash functions $(h_{i,1}, h_{i,2})$; each makes to a universe $U = \{0, 1, 2, \ldots 5\}$. We apply these hash functions to 5 points in a set P and a query point q, the results are shown in the table below. We return objects close to q if and only if they have an exact match in any banded hash function.

hash	$h_{1,1}$	$h_{1,2}$	$h_{2,1}$	$h_{2,2}$	$h_{3,1}$	$h_{3,2}$	close?
q	3	5	1	2	3	2	
p_1	**[3,**	**5]**	0	**2**	4	3	yes
p_2	**3**	4	**1**	0	4	**2**	no
p_3	0	2	4	3	5	1	no
p_4	0	1	**[1,**	**2]**	0	**2**	yes
p_5	4	**5**	3	5	4	1	no

We observe that only p_1 and p_4 are designated as close. Object p_1 collides on H_1 since both p_1 and q have $h_{1,1} = 3$ and $h_{1,2} = 5$. Object p_4 collides on H_2 since both p_4 and q have $h_{2,1} = 1$ and $h_{2,2} = 2$. Note that although p_2 collides with q on 3 individual hash functions ($h_{1,1} = 3$, $h_{2,1} = 1$, and $h_{3,2} = 2$) it never has an entire banded hash function collide, so it is not marked as close.

Analysis of banding. We will analyze the simple case where there is a hash family $\mathcal{H}$ so that we have $\mathbf{Pr}_{h \in \mathcal{H}}[h(p) = h(q)] = \mathbf{s}(p, q)$. Let $s = \mathbf{s}(p, q)$, and we will analyze the case where we use r bands with b hash functions each.

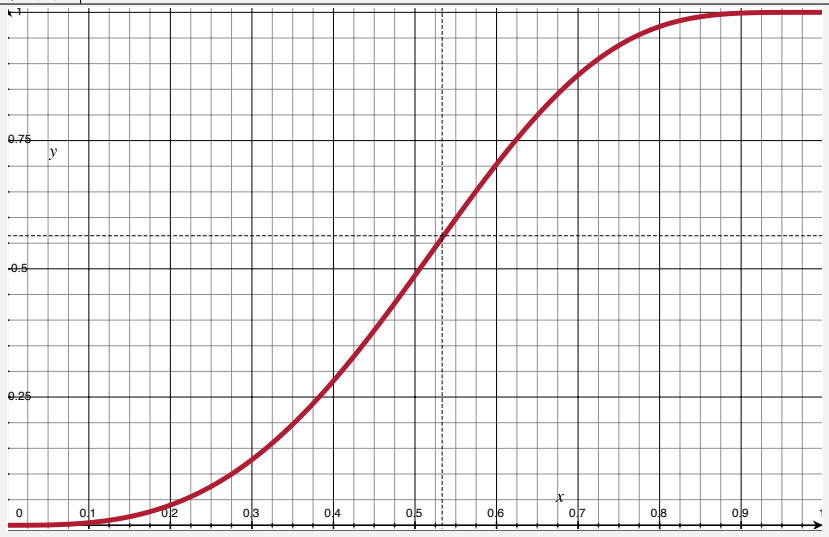
$$
\begin{aligned}
s &= \text{probability of collision on any one hash function} \\
s^b &= \text{probability all hash functions collide in 1 band} \\
(1 - s^b) &= \text{probability not all collide in 1 band} \\
(1 - s^b)^r &= \text{probability that in no band, do all hashes collide} \\
f_{b,r}(s) = 1 - (1 - s^b)^r &= \text{probability all hashes collide in at least 1 band}
\end{aligned}
$$

So this function $f_{b,r}(s) = 1 - (1 - s^b)^r$ describes the probability that two objects of similarity s are marked as similar in the banding approach. A similar, but slightly messier, analysis can be applied for any $(\gamma, \phi, \alpha, \beta)$-sensitive hash family.

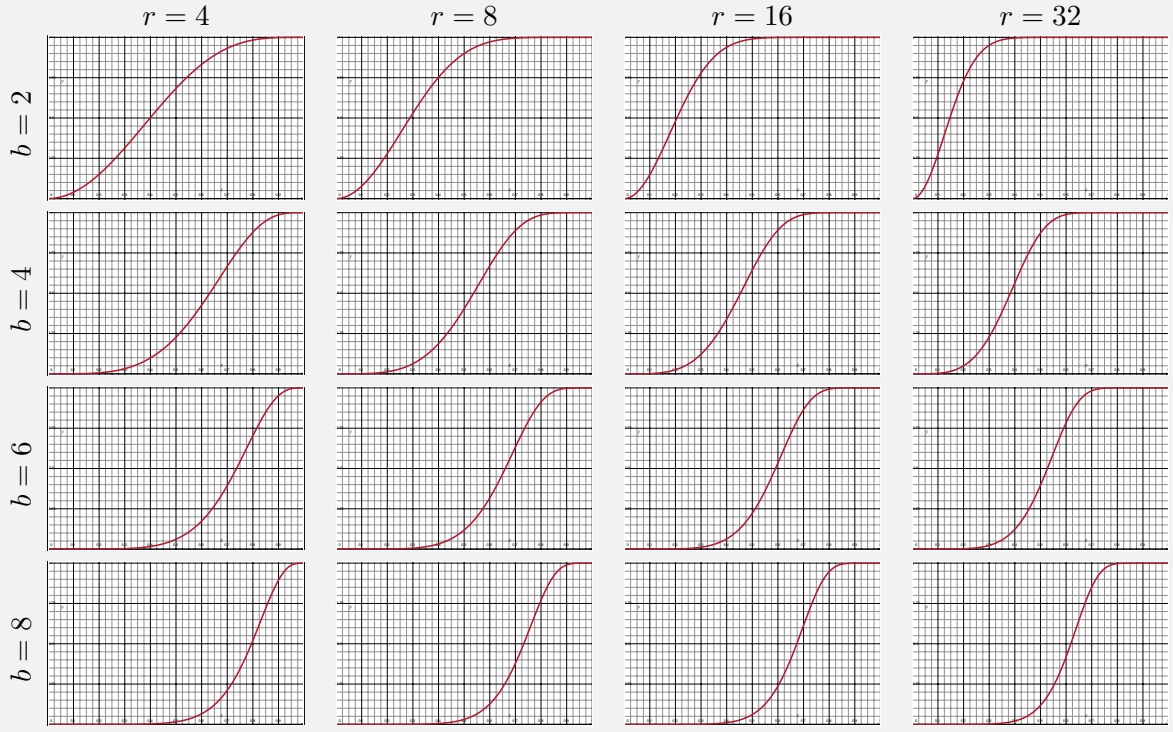
Example: Plotting the LSH similarity function

We can plot $f_{b,r}$ as an S-curve where on the x-axis $s = \mathbf{s}(p, q)$ and the y-axis represents the probability that the pair p, q is detected as close. In this example we have $t = 15$ and $r = 5$ and $b = 3$.

s	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
$f_{3,5}(s)$	0.005	0.04	0.13	0.28	0.48	0.70	0.88	0.97	0.998



Next we change the r and b parameters and observe what happens to the plot of the S-curve $f_{r,b}(s)$. We show combinations of values $r \in \{4, 8, 16, 32\}$ and $b \in \{2, 4, 6, 8\}$. The value b is fixed each row, and increases as the rows go down. The value r is fixed each column, and increases as the columns go from left to right.



We see that as $r \cdot b$ increases, the curve becomes steeper. As r increases (move to the right), the curve shifts to the left, that is a smaller similarity is likely designated as close. As b increases (move down), the curve shifts to the right, that is a larger similarity is likely designated as close.

Choice of r and b. Usually there is a budget of t hash function one is willing to use. Perhaps it is $t = (1/2\varepsilon^2)\ln(2/\delta)$; e.g., the number required by a Chernoff-Hoeffding bound to estimate the similarity within ε-error, with probability at least $1-\delta$. Then how does one divvy them up among r and b?

The threshold τ' where f has the steepest slope is about $\tau' \approx (1/r)^{1/b}$. So given a similarity value $s = \mathbf{s}(p,q)$, if we want to return objects more similar than τ' (e.g. $\tau' = s = \alpha = 1-\beta$) we can solve for $b = t/r$ in $\tau' = (b/t)^{1/b}$ to (very roughly) yield $b \approx -\log_{\tau'}(t)$. And set $r = t/b = t/(-\log_{\tau'}(t))$. This is only a rule-of-thumb. It is often best to plot the f function, and play with values of b and r to find your desired trade-off, but this gives a good place to start searching.

If there is no budget on r and b, as they increase, the S-curve produced by $f(s)$ gets sharper and sharper, and provides a more accurate retrieval of all and only the close data points.

4.6.4 LSH for Angular Distance

Locality sensitive hash families can be developed for more than just the simplistic 1-dimensional triangle similarity. Here we describe such a family for the angular distance $\mathbf{d}_{\mathsf{ang}}$ and similarity $\mathbf{s}_{\mathsf{ang}}$. Recall these functions operate on vectors $a, b \in \mathbb{S}^{d-1}$, which are unit vectors in $\mathbb{R}^d$.

A hash h_u from hash family $\mathcal{H}_{\mathsf{ang}}$ is defined

$$\mathcal{H}_{\mathsf{ang}} = \left\{ h_u(\cdot) = \mathsf{sign}(\langle \cdot, u \rangle) \mid u \sim \mathsf{Unif}(\mathbb{S}^{d-1}) \right\}$$

Lets unpack this. First choose a uniform random unit vector $u \in \mathbb{S}^{d-1}$, that is so any unit vector in $\mathbb{R}^d$ is equally likely to be chosen ($u \sim \mathsf{Unif}(\mathbb{S}^{d-1})$). Then $h_u(p) = \mathsf{sign}(\langle a, u \rangle)$, that is if the dot product of a and u is positive, it returns $+1$, and if it is negative, it returns -1.

We will show at $\mathbf{Pr}_{h_u \in \mathcal{H}_{\mathsf{ang}}}[h_u(a) = h_u(b)] = \mathbf{s}_{\mathsf{ang}}(a,b)/\pi$ for any $a, b \in \mathbb{S}^{d-1}$. Thus, after scaling by π, the amplification analysis for this similarity and hash family pair is exactly same as with the simple triangle similarity example.

Geometry of Angular LSH

Here we see why $\mathbf{Pr}_{h_u \in \mathcal{H}_{\mathsf{ang}}}[h_u(a) = h_u(b)] = \mathbf{s}_{\mathsf{ang}}(a,b)/\pi$.

First consider the case for unit vectors $u \in \mathbb{S}^1$, the circle. These vectors u can be generated from a random value $\eta \sim \mathsf{Unif}[0, 2\pi)$ and then walking on the the unit circle a distance u' from the point $(1,0)$. Let $F_{u,\perp}$ be the set of perpendicular vectors to u; it is 2 vectors which can be specifically defined as walking a distance η from $(0,1)$ and from $(0,-1)$, instead of starting from $(1,0)$. The set $F_{u,\perp}$ partitions the points $s \in \mathbb{S}^1$ with $h_u(s) = 1$ from those with $h_u(s) = -1$.

To analyze whether, any two points $a, b \in \mathbb{S}^1$ hash together $h_u(a) = h_u(b)$, we can always consider $a = (0,1)$ without loss of generality, by rotating the starting points to define u and correspondingly $F_{u,\perp}$. It is also sufficient to consider the case where b has a positive y-coordinate; otherwise swap roles of a, b, and rotate b to $(0,1)$. The key observation is that $h_u(a) \neq h_u(b)$ if and only if a vector $u' \in F_{u,\perp}$ lies between a and b (in the positive y-coordinate part of $\mathbb{S}^1$).

The probability of generating such a u is precisely $\mathbf{d}_{\mathsf{ang}}(a,b)/\pi$: it occurs if $\eta \in [0, \mathbf{d}_{\mathsf{ang}}]$ or $\eta \in [\pi, \pi + \mathbf{d}_{\mathsf{ang}}]$, which happens with probability of $2\mathbf{d}_{\mathsf{ang}}/(2\pi)$.

It turns out the general case where $p, q \in \mathbb{S}^{d-1}$ follows the exact same analysis. We only need to consider the great circle C on $p, q \in \mathbb{S}^{d-1}$ which contains p, q. This great circle is equivalent to $\mathbb{S}^1$. Moreover, for a random $u \in \mathbb{S}^{d-1}$, the set $F_{u,\perp}$ now also defines a set equivalent to $\mathbb{S}^{d-2}$. The sets C and $F_{u,\perp}$ will intersect in precisely 2 cases, with probability 1. The probability that $F_{u,\perp}$ intersects the short arc connecting p, q on C is again precisely $\mathbf{d}_{\mathsf{ang}}(p,q)/\pi$. Hence $\mathbf{Pr}_{h_u \in \mathcal{H}_{\mathsf{ang}}}[h_u(p) = h_u(q)] = 1 - \mathbf{d}_{\mathsf{ang}}(p,q)/\pi = \mathbf{s}_{\mathsf{ang}}(a,b)/\pi$.

Generating random unit vectors. A missing algorithmic element of this hash family is generating a random unit vector. While generating a random uniform scalar is assumed standard, random unit vectors require the following observations. For instance, generating d uniform vectors $u_1, u_2, \ldots, u_d \sim \mathsf{Unif}[-1, 1]$ and then normalizing $u = (u_1, u_2, \ldots, u_d) \to u/\|u\|$ does not give the desired result. It places too much probability near vector $(1/\sqrt{d}, 1/\sqrt{d}, \ldots, 1/\sqrt{d})$ and similar vectors with signs flipped.

The easiest way to generate a random unit vector is through Gaussian random variables. A d-dimensional Gaussian distribution $\mathcal{G}_d$ is defined:

$$\mathcal{G}_d(x) = \frac{1}{(2\pi)^{d/2}} e^{-\|x\|_2^2/2}.$$

If we have two uniform random numbers $u_1, u_2 \sim \mathsf{Unif}[0, 1]$ then we can generate two independent 1-dimensional Gaussian random variables as (using the Box-Muller transform):

$$y_1 = \sqrt{-2\ln(u_1)}\cos(2\pi u_2)$$
$$y_2 = \sqrt{-2\ln(u_1)}\sin(2\pi u_2).$$

A uniform Gaussian has the (*amazing!*) property that all coordinates (in any orthogonal basis) are independent of each other. Thus to generated a point $x \in \mathbb{R}^d$ from a d-dimensional Gaussian, for each coordinate i we assign it the value of an independent 1-dimensional Gaussian random variable.

Moreover, the amazing property of this d-dimensional Gaussian random variable $x \sum \mathcal{G}_d$, implies that projected to any one-dimensional subspace (through its mean, the origin) that it is a 1-dimensional Gaussian distribution. That is for any unit vector v and $x \sim \mathcal{G}_d$, then $\langle x, v\rangle \sim \mathcal{G}_1$. This means it does not favor any directions (as the d-variate uniform distribution $(\mathsf{Unif}[-1, 1])^d$ favors the "corners"), and hence if we normalize $u \leftarrow x/\|x\|_2$ for $x \sim \mathcal{G}_d$, then $u \sim \mathsf{Unif}(\mathbb{S}^{d-1})$, as desired.

4.6.5 LSH for Euclidean Distance

The LSH hash family for Euclidean distance $\mathbf{d}_2$ is a combination of the ideas for angular distance and for the triangle similarity. However, the probability does not work out quite as nicely so we require the more general notion of $(\gamma, \phi, \alpha, \beta)$-sensitivity.

The hash family $\mathcal{H}_{L_2,\tau}$ requires a desired similarity parameter τ (which was implicit in the scaling of the triangle similarity), but not present in the angular similarity. The set is defined

$$\mathcal{H}_{E,\tau} = \left\{ h_{u,\eta}(\cdot) = \lceil \langle \cdot, u\rangle + \eta \rceil \mid u \sim \mathsf{Unif}(\mathbb{S}^{d-1}), \eta \sim \mathsf{Unif}[0, \tau] \right\}.$$

That is, it relies on a random unit vector u, and a random offset η. The hash operation $h_{u,\eta}(x) = \lceil \langle x, u\rangle + \eta \rceil$ first projects onto direction u, and then offsets by η in this directions, and rounds up to an integer. For a large enough integer t (e.g., if there are at most n data points, then using $t = n^3$), then using instead $h_{u,\eta}(x) \mod t$ so it maps to a finite domain $0, 1, 2, \ldots, t-1$ is also common.

The hash family $\mathcal{H}_{L_2,\tau}$ is $(\tau/2, 2\tau, 1/2, 1/3)$-sensitive with respect to $\mathbf{d}_2$.

Geometry of Euclidean LSH Sensitivity

We show here that $\mathcal{H}_{L_2,\tau}$ is $(\tau/2 = \gamma, 2\tau = \phi, 1/2 = \alpha, 1/3 = \beta)$-sensitive with respect to $\mathbf{d}_2$.
To see that we do not miss too many real collisions, observe that the projection onto u is contractive. So for $a, b \in \mathbb{R}^d$, then $|\langle a, u\rangle - \langle b, u\rangle| \leq \|a - b\|$. Thus as with the 1-dimensional triangle similarity, if $\|a - b\| \leq \tau/2 = \gamma$ then $|\langle a, u\rangle - \langle b, u\rangle| \leq \tau/2$ and a, b fall in the same bin with probability at least $1/2 = \alpha$.
To see that we do not observe too many spurious collisions, we can use an argument similar to that for angular distance. If $\|a - b\| > 2\tau = \phi$ and they collide, then the vector u must be sufficiently orthogonal to $(a - b)$ so the norm is reduced by half; otherwise if $|\langle a, u\rangle - \langle b, u\rangle| \geq \tau$, they cannot be in the same bin. Formally

$$|\langle a, u\rangle - \langle b, u\rangle| = |\langle a - b, u\rangle| = \|a - b\|\|u\|| \cos \theta_{u,a-b}|,$$

where $\theta_{u,a-b}$ is the angle (in radians) between u and $a - b$. We know $|\cos(\theta_{u,a-b})| \leq 1/2$ only if $\theta_{u,a-b} \geq \pi/3$. Fixing the 2-dimensional subspace which includes 0, u, and $a - b$, then we can see that the probability $\theta_{u,a-b} \geq \pi/3$ is at most $1/3 = \beta$ (that is $(2\pi/3)$ options for u out of 2π).

p-Stable Random Variables and LSH for L_p-Norms. There exists a beautiful extension to create similar LSH hash families $\mathcal{H}_{L_p,\tau}$ for any L_p distance with $p \in (0, 2]$. The main difference is to replace the $u \sim \mathsf{Unif}(\mathbb{S}^{d-1})$ which is really some $x \sim \mathcal{G}_d$, with another d-dimensional random variable from what is known as a p-stable distribution.

A distribution μ over $\mathbb{R}$ is *p-stable* (for $p \geq 0$) if the following holds. Consider $d + 1$ random variables $X_0, X_1, \ldots, X_d \sim \mu$. Then considering any d real values $\{v_1, \ldots, v_d\}$, then the random variable $\sum_i v_i X_i$ has the same distribution as $(\sum_i |v_i|^p)^{1/p} X_0$. Such p-stable distributions exist for $p \in (0, 2]$. Special cases are

- The Gaussian distribution $\mathcal{G}(x) = \frac{1}{\sqrt{2\pi}} \exp(-x^2/2)$ is 2-stable.
- Then the Cauchy distribution $\mathcal{C}(x) = \frac{1}{\pi} \frac{1}{1+x^2}$ is 1-stable.

Intuitively, this allows us to replace a sum of random variables with a single random variables by adjusting coefficients carefully. But actually, it is the composition of the coefficients that is interesting.

In particular, for $p = 2$ and a vector $v = (v_1, v_2, \ldots, v_d)$ where we want to estimate $\|v\|_2 = (\sum_i v_i^2)^{1/2}$, we can consider each coordinate v_i individually as an estimate by using a p-stable random variable. That is we can estimate $\|v\|_2$ by choosing $d + 1$ random Gaussian vectors $g_0, g_1, \ldots, g_d \sim \mathcal{G}$, and calculating $(1/g_0) \sum_i g_i v_i = (1/g_0)\langle g, v\rangle$, where $g = (g_1, \ldots, g_d)$ Note that this by dividing by g_0, we are approximately correctly normalizing the d-dimensional Gaussian random variable. This turns out to be sufficient for Euclidean LSH and other similar methods for high-dimensional we will see later.

Using the Cauchy random variables $c_0, c_1, \ldots, c_d \sim \mathcal{C}$ in place of the Gaussian ones allows us to estimate $\|v\|_1 = \sum_i |v_i|$ as $(1/c_0)\langle c, v\rangle$ with $c = (c_1, \ldots c_d)$.

4.6.6 Minhashing as LSH for Jaccard Distance

The final example of LSH we will provide is for the Jaccard distance, which is defined on sets. This hash family is $\mathcal{H}_J$ again has the nice property that $\mathbf{Pr}_{h_\sigma \in \mathcal{H}_J}[h_\sigma(A) = h_\sigma(B)] = \mathbf{s}_J(A, B)$, so the previous amplification analysis applies directly. Consider sets defined over a domain Ω of size n, and define Π_n as the set of all permutations from Ω to distinct integers in $[n] = \{1, 2, 3, \ldots, n\}$. So for each element $\omega \in \Omega$

and $\sigma \in \Pi_n$ maps $\sigma(\omega) = i$ for some $i \in [n]$, and for a fixed $\sigma(\omega) \neq \sigma(\omega')$ for $\omega \neq \omega'$. In practice we can relax this $\neq$ restriction as we discuss below. Then we define

$$\mathcal{H}_J = \left\{ h_\sigma(A) = \min_{\omega \in A} \sigma(\omega) \mid \sigma \sim \mathsf{Unif}(\Pi_n) \right\}.$$

That is, the hash function h_σ applies σ to each ω in A, and then returns the smallest value out of all of them. Due to the return of the smallest value, this type of approach is often called a *min hash*.

Example: Min Hashing

Consider two sets $A = \{a, b, e, f, h\}$ and $B = \{b, c, e, g, j\}$ over the domain $\Omega = \{a, b, c, d, e, f, g, h, i, j\}$. We can now set choose two permutations functions $\sigma_1, \sigma_2 \in \mathcal{H}_J$:

	a	b	c	d	e	f	g	h	i	j
σ_1	4	6	3	1	2	8	9	5	10	7
σ_2	3	7	8	5	9	10	1	4	6	2

These permutations induce hash functions $h_{\sigma_1}, h_{\sigma_2} \in \mathcal{H}_J$. These can then be applied to sets A and B as

	A	B
h_{σ_1}	2	2
h_{σ_2}	3	1

For instance $h_{\sigma_1}(A) = \min\{\sigma_1(a), \sigma_1(b), \sigma_1(e), \sigma_1(f), \sigma_1(h)\} = \min\{4, 6, 2, 8, 5\} = 2$.
Observe that the Jaccard similarity $\mathbf{s}_\mathsf{J}(A, B) = \frac{2}{8}$. On the hash functions there is a collision for $h_{\sigma_1}(A) = h_{\sigma_1}(B)$, but not for h_{σ_2}, so they would estimate the similarity as $\frac{1}{2}$.

It is useful to understand why $\mathbf{Pr}_{h_\sigma \in \mathcal{H}_J}[h_\sigma(A) = h_\sigma(B)] = \mathbf{s}_J(A, B)$. We think of three types of elements from Ω: the objects $\Omega_{A \cap B}$ which are in both A and B; the objects $\Omega_{A \veebar B}$ which are in either A or B, but not both; and the objects $\Omega_{\notin A,B}$ which are in neither A or B. Note that the Jaccard similarity is precisely $\mathbf{s}_\mathsf{J}(A, B) = \frac{|\Omega_{A \cap B}|}{|\Omega_{A \cap B}| + |\Omega_{A \veebar B}|}$. Recall, the similarity has no dependence on the set $\Omega_{\notin A,B}$ hence we actually do not need to know all of Ω. But we can also see that the probability of a hash collision is precisely $\frac{|\Omega_{A \cap B}|}{|\Omega_{A \cap B}| + |\Omega_{A \veebar B}|}$. Any value returned by h_σ can only be an element from $\Omega_{A \cap B} \cup \Omega_{A \veebar B}$, these are equally likely, and it is only a collision if it is one from $\Omega_{A \cap B}$.

Fast min hash functions. So this definition of min hash functions is easy to analyze but it is suboptimal. It needs to define and store a hash function from Ω to $[|\Omega|]$. But we would like to use these ideas over domains such as IP addresses or words or strings where there is an enormous implicit domain Ω that is often too big to store. An alternative is to replace σ with another separating hash function $f : \Omega \to [N]$ where N is a sufficiently large value (such as the size of the largest set we expect to encounter to the power 3). There will now be some chance of collisions between objects in any set A, but it will be small enough to have insignificant effect on the analysis. Moreover, most programming languages include built in such hash functions that can operate on strings or large integers or double precisions values (e.g., using the SHA-1 hash with a salt). Then we can define our hash family as

$$\mathcal{H}'_J = \{h_f(A) = \min_{\omega \in A} f(\omega \oplus S) \mid f : \Omega \to [N], S \sim \text{random salt}\}.$$

Exercises

Q4.1: Consider two vectors $a = (1, 2, -4, 3, -6)$ and $b = (1, 2, 5, -2, 3) \in \mathbb{R}^5$.

1. Calculate the $\mathbf{d}_1(a, b)$, $\mathbf{d}_2(a, b)$, $\mathbf{d}_0(a, b)$, $\mathbf{d}_\infty(a, b)$, and sort the answers.
2. Without calculating $\mathbf{d}_3(a, b)$, explain where it will fall in the sorted order.
3. Does it make sense to normalize these vectors so they lie in Δ^4 (e.g., by dividing by $\|a\|_1$ and $\|b\|_2$, and use the Kullback-Liebler divergence $\mathbf{d}_{KL}$ on them? Why or why not?
4. Normalize the data sets to like on $\mathbb{S}^4$, and compute the cosine distance between them.

Q4.2: Consider sets $A = \{1, 2, 4, 8\}$ and $B = \{1, 2, 3\}$.

1. Calculate the Jaccard distance $\mathbf{d}_{\mathsf{J}}(A, B)$.
2. Compute the follow similarities if they are well-defined?
   - Jaccard similarity $\mathbf{s}_{\mathsf{J}}(A, B)$
   - Hamming similarity $\mathbf{s}_{\mathsf{Ham}}(A, B)$
   - Andberg similarity $\mathbf{s}_{\mathsf{Andb}}(A, B)$
   - Rogers-Tanimoto similarity $\mathbf{s}_{\mathsf{RT}}(A, B)$
   - Soerensen-Dice similarity $\mathbf{s}_{\mathsf{Dice}}(A, B)$

5 Linear Regression

We introduce the basic model of linear regression. It builds a linear model to predict one variable from one other variable or from a set of other variables. We will demonstrate how this simple technique can extend to building potentially much more complex polynomial models. Then we will introduce the central and extremely powerful idea of cross-validation. This method fundamentally changes the statistical goal of validating a model, to characterizing the data.

5.1 Simple Linear Regression

We will begin with the simplest form of linear regression. The input is a set of n 2-dimensional data points $(X, y) = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$. The ultimate goal will be to predict the y values using only the x-values. In this case x is the *explanatory variable* and y is the *dependent variable*.

The notation (X, y), with a uppercase X and lowercase y will become clear later since it will commonly generalize to multidimensional settings for the x-part, but stay 1-dimensional (or otherwise simple) for the y part.

In order to do this, we will "fit" a line through the data of the form

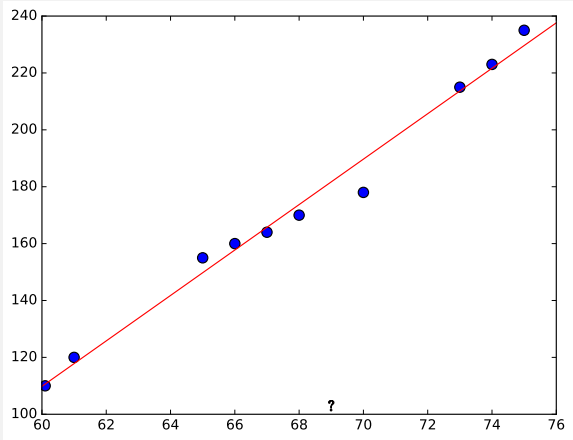
$$y = \ell(x) = ax + b,$$

where a (the slope) and b (the intercept) are parameters of this line. The line ℓ is our "model" for this input data.

Example: Fitting a line to height and weight

Consider the following data set that describes a set of heights and weights.

height (in)	weight (lbs)
66	160
68	170
60	110
70	178
65	155
61	120
74	223
73	215
75	235
67	164
69	?



Note that in the last entry, we have a height of 69, but we do not have a weight. If we were to guess the weight in the last row, how should we do this?

We can draw a line (the red one) through the data points. Then we can guess the weight for a data point with height 69, by the value of the line at height 69 inches: about 182 pounds.

Measuring error. The purpose of this line is not just to be close to all of the data (for this we will have to wait for PCA and dimensionality reduction). Rather, its goal is prediction; specifically, using the explanatory variable x to predict the dependent variable y.

In particular, for every value $x \in \mathbb{R}$, we can predict a value $\hat{y} = \ell(x)$. Then on our dataset, we can examine for each x_i how close $\hat{y}_i$ is to y_i. This difference is called a *residual*:

$$r_i = |y_i - \hat{y}_i| = |y_i - \ell(x_i)|.$$

Note that this residual is not the distance from y_i to the line ℓ, but the distance from y_i to the corresponding point with the same x-value. Again, this is because our only goal is prediction of y. And this will be important as it allows techniques to be immune to the choice of units (e.g., inches or feet, pounds or kilograms)

So the residual measures the error of a single data point, but how should we measure the overall error of the entire data set? The common approach is the sum of squared errors:

$$\mathsf{SSE}((X, y), \ell) = \sum_{i=1}^{n} r_i^2 = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 = \sum_{i=1}^{n} (y_i - \ell(x_i))^2.$$

Why is this the most common measure? Here are 3 explanations:

- The sum of squared errors was the optimal result for a single point estimator under Gaussian noise using Bayesian reasoning, when there was assumed Gaussian noise (See T2). In that case the answer was simply the mean of the data.
- If you treat the residuals as a vector $r = (r_1, r_2, \ldots, r_n)$, then the standard way to measure total length of a vector r is through its norm $\|r\|$, which is most commonly its 2-norm $\|r\| = \|r\|_2 = \sqrt{\sum_{i=1}^{n} r_i^2}$. The square root part is not so important (it does not change which line ℓ minimizes this error), so removing this square root, we are left with SSE.
- For this specific formulation, there is a simple closed form solution (which we will see next) for ℓ. And in fact, this solution will generalize to many more complex scenarios.

There are many other formulations of how best to measure error for the fit of a line (and other models), but we will not cover all them in this text.

Solving for ℓ. To solve for the line which minimizes $\mathsf{SSE}((X, y), \ell)$ there is a very simply solution, in two steps. Calculate averages $\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$ and $\bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i$, and create centered n-dimension vectors $\bar{X} = (x_1 - \bar{x}, x_2 - \bar{x}, \ldots, x_n - \bar{x})$ for all x-coordinates and $\bar{Y} = (y_1 - \bar{y}, y_2 - \bar{y}, \ldots, y_n - \bar{y})$ for all y-coordinates.

1. Set $a = \langle \bar{Y}, \bar{X} \rangle / \|\bar{X}\|^2$
2. Set $b = \bar{y} - a\bar{x}$

This defines $\ell(x) = ax + b$.

We will provide the proof for why this is the optimal solution for the high-dimensional case (in short, it can be shown by expanding out the SSE expression, taking the derivative, and solving for 0). We will only provide some intuition here.

First lets examine the intercept

$$b = \frac{1}{n}\sum_{i=1}^{n} (y_i - ax_i) = \bar{y} - a\bar{x}$$

This setting of b ensures that the line $y = \ell(x) = ax + b$ goes through the point $(\bar{x}, \bar{y})$ at the center of the data set since $\bar{y} = \ell(\bar{x}) = a\bar{x} + b$.

Second, to understand how the slope a is chosen, it is illustrative to reexamine the dot product as

$$a = \frac{\langle \bar{Y}, \bar{X} \rangle}{\|\bar{X}\|^2} = \frac{\|\bar{Y}\| \cdot \|\bar{X}\| \cdot \cos\theta}{\|\bar{X}\|^2} = \frac{\|\bar{Y}\|}{\|\bar{X}\|} \cos\theta,$$

where θ is the angle between the n-dimensional vectors $\bar{Y}$ and $\bar{X}$. Now in this expression, the $\|\bar{Y}\|/\|\bar{X}\|$ captures how much on (root-squared) average $\bar{Y}$ increases as $\bar{X}$ does (the rise-over-run interpretation of slope). However, we may want this to be negative if there is a negative correlation between $\bar{X}$ and $\bar{Y}$, or really this does not matter much if there is no correlation. So the $\cos\theta$ term captures the correlation after normalizing the units of $\bar{X}$ and $\bar{Y}$.

```
import numpy as np

x = np.array([66, 68, 65, 70, 65, 62, 74, 70, 71, 67])
y = np.array([160, 170, 159, 188, 150, 120, 233, 198, 201, 164])

ave_x = np.average(x)
ave_y = np.average(y)

#first center the data points
xc = x - ave_x
yc = y - ave_y

a = xc.dot(yc)/xc.dot(xc)
b = ave_y - a*ave_x
print a, b

#or with scipy
from scipy import polyfit
(a,b)=polyfit(x,y,1)
print a, b

#predict weight at x=69
w=a*69+b
```

5.2 Linear Regression with Multiple Explanatory Variables

Magically, using linear algebra, everything extends gracefully to using more than one explanatory variables. Now consider a data set $(X, y) = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$ where each data point has $x_i = (x_{i,1}, x_{i,2}, \ldots, x_{i,d}) \in \mathbb{R}^d$ and $y_i \in \mathbb{R}$. That is there are d explanatory variables, as the coordinates of x_i, and one dependent variable in y_i. We would now like to use all of these variables at once to make a single (linear) prediction about the variable y_i. That is, we would like to create a model

$$\begin{aligned}\hat{y}_i = M_\alpha(x_i) = M_\alpha(x_{i,1}, x_{i,2}, \ldots, x_{i,d}) &= \alpha_0 + \sum_{j=1}^{d} \alpha_j x_{i,j} \\ &= \alpha_0 + \alpha_1 x_{i,1} + \alpha_2 x_{i,2} + \ldots + \alpha_d x_{i,d}. \\ &= \langle \alpha, (1, x_{i,1}, x_{i,2}, \ldots, x_{i,d}) \rangle = \langle \alpha, (1, x_i) \rangle.\end{aligned}$$

In the above equivalent notations α_0 serves the purpose of the intercept b, and all of the α_js replace the single coefficient a in the simple linear regression. Indeed, we can write this model as a dot product between the $(d+1)$-dimensional vectors $\alpha = (\alpha_0, \alpha_1, \ldots, \alpha_d)$ and $(1, x_{i,1}, x_{i,2}, \ldots, x_{i,d}) = (1, x_i)$. As promised, the magic of linear algebra has allowed us to describe a more complex linear model M_α. Next we will see how to solve it.

Example: Predicting customer value

A website specializing in dongles (dongles-r-us.com) wants to predict the total dollar amount that visitors will spend on their site. It has installed some software that can track three variables:

- time (the amount of time on the page in seconds): X_1,
- jiggle (the amount of mouse movement in cm): X_2, and
- scroll (how far they scroll the page down in cm): X_3.

Also, for a set of past customers they have recorded the

- sales (how much they spend on dongles in cents): y.

We see a portion of their data set here with $n = 11$ customers:

time: X_1	jiggle: X_2	scroll: X_3	sales: y
232	33	402	2201
10	22	160	0
6437	343	231	7650
512	101	17	5599
441	212	55	8900
453	53	99	1742
2	2	10	0
332	79	154	1215
182	20	89	699
123	223	12	2101
424	32	15	8789

To build a model, we recast the data as an 11×4 matrix $X = [\mathbf{1}, X_1, X_2, X_3]$. We let y be the 11-dimensional column vector.

$$X = \begin{bmatrix} 1 & 232 & 33 & 402 \\ 1 & 10 & 22 & 160 \\ 1 & 6437 & 343 & 231 \\ 1 & 512 & 101 & 17 \\ 1 & 441 & 212 & 55 \\ 1 & 453 & 53 & 99 \\ 1 & 2 & 2 & 10 \\ 1 & 332 & 79 & 154 \\ 1 & 182 & 20 & 89 \\ 1 & 123 & 223 & 12 \\ 1 & 424 & 32 & 15 \end{bmatrix} \quad y = \begin{bmatrix} 2201 \\ 0 \\ 7650 \\ 5599 \\ 8900 \\ 1742 \\ 0 \\ 1215 \\ 699 \\ 2101 \\ 8789 \end{bmatrix}$$

The goal is to learn the 4-dimensional column vector $\alpha = [\alpha_0; \alpha_1; \alpha_2, \alpha_3]$ so

$$y \approx X\alpha.$$

Setting $\alpha = (X^T X)^{-1} X^T y$ obtains (roughly) $\alpha_0 = 2626$, $\alpha_1 = 0.42$, $\alpha_2 = 12.72$, and $\alpha_3 = -6.50$. This implies an average customer with no interaction on the site generates $\alpha_0 = \$2.62$. That time does not have a strong effect here (only a coefficient α_1 at only 0.42), but jiggle has a strong correlation (with coefficient $\alpha_2 = 12.72$, this indicates 12 cents for every centimeter of mouse movement). Meanwhile scroll has a negative effect (with coefficient $\alpha_3 = -6.5$); this means that the more they scroll, the less likely they are to spend (just browsing dongles!).

Given a data point $x_i = (x_{i,1}, x_{i,2}, \ldots, x_{i,d})$, we can again evaluate our prediction $\hat{y}_i = M(x_i)$ using the residual value $r_i = |y_i - \hat{y}_i| = |y_i = M(x_i)|$. And to evaluate a set of n data points, it is standard to consider the sum of squared error as

$$\mathsf{SSE}(X, y, M) = \sum_{i=1}^{n} r_i^2 = \sum_{i=1}^{n} (y_i - M(x_i))^2.$$

To obtain the coefficients which minimize this error, we can now do so with very simple linear algebra.

First we construct a $n \times (d+1)$ data matrix $\tilde{X} = [\mathbf{1}, X_1, X_2, \ldots, X_d]$, where the first column $\mathbf{1}$ is the n-dimensional all ones column vector $[1; 1; \ldots; 1]$. Each of the next d columns is a column vector X_j, where $x_{i,j} = X_{i,j}$ is the ith entry of X_j and represents the jth coordinate of data point x_i. Then we let y be a n-dimensional column vector containing all the dependent variables. Now we can simply calculate the $(d+1)$-dimensional column vector $\alpha = (\alpha_0, \alpha_1, \ldots, \alpha_d)$ as

$$\alpha = (\tilde{X}^T \tilde{X})^{-1} \tilde{X}^T y.$$

Let us compare to the simple case where we have 1 explanatory variable. The $(\tilde{X}^T \tilde{X})^{-1}$ term replaces the $\frac{1}{\|X\|^2}$ term. The $\tilde{X}^T y$ replaces the dot product $\langle \bar{Y}, \bar{X} \rangle$. And we do not need to separately solve for the intercept b, since we have created a new column in $\tilde{X}$ of all 1s. The α_0 replaces the intercept b; it is multiplied by a value 1 in $\tilde{X}$ equivalent to b just standing by itself.

Often the matrices X and $\tilde{X}$ are used interchangeably, and hence we drop the $\tilde{}$ from $\tilde{X}$ in most situations. We can either simply treat all data points x_i as one-dimension larger (with always a 1 in the first coordinate), or we can fit a model on the original matrix X and ignore the offset parameter α_0, which is then by default 0. The former approach, where each x_i is just assumed one dimension larger is more common since it automatically handles the offset parameter.

Geometry of the Normal Equations

Why does $\alpha = (X^T X)^{-1} X^T y$ minimize the sum of squared errors:

$$\mathsf{SSE}(X, y, M) = \sum_{i=1}^{n} r_i^2 = \sum_{i=1}^{n} (y_i - \langle \alpha, x_i \rangle)^2?$$

Fixing the data X and y, and representing M by its parameters α, we can consider a function $S(\alpha) = \mathsf{SSE}(X, y, M)$. Then we observe that $S(\alpha)$ is a quadratic function in α, and it is convex (see T6), so its minimum is when the gradient is 0. This will be true when each partial derivative is 0. Let $X_{i,j}$ be the jth coordinate of x_i so residual $r_i = (y_i - \langle \alpha, x_i \rangle)$ has partial derivative $\frac{\mathrm{d}r_i}{\mathrm{d}\alpha_j} = -X_{i,j}$. Then set to 0 each partial derivative:

$$0 = \frac{\mathrm{d}S(\alpha)}{\mathrm{d}\alpha_j} = 2 \sum_{i=1}^{n} r_i \frac{\mathrm{d}r_i}{\mathrm{d}\alpha_j} = 2 \sum_{i=1}^{n} r_i (-X_{i,j}) = 2 \sum_{i=1}^{n} (y_i - \langle \alpha, x_i \rangle)(-X_{i,j})$$

We can rearrange this into the *normal equations*

$$\sum_{i=1}^{n} X_{i,j} \langle x_i, \alpha \rangle = \sum_{i=1}^{n} X_{i,j} y_i \qquad \text{for all } j \in \{1, 2, \ldots, d\}$$

$$\text{equivalently} \quad (X^T X)\alpha = X^T y.$$

Multiplying by the inverse *gram matrix* $X^T X$ on both sides reveals the desired $\alpha = (X^T X)^{-1} X^T y$.
Geometry: To see why these are called the normal equations, consider a form

$$\mathbf{0} = (y - X\alpha)^T X,$$

where $\mathbf{0}$ is the all-zeros vector in $\mathbb{R}^n$. Thus for any vector $v \in \mathbb{R}^n$, then

$$0 = (y - X\alpha)^T X v = \langle y - X\alpha, Xv \rangle.$$

This includes when $v = \alpha$; under this setting then $Xv = X\alpha = \hat{y}$. Notice that $r = y - X\alpha$ is the vector in $\mathbb{R}^n$ that stores all residuals (so $y = \hat{y} + r$). Then the normal equations implies that

$$0 = \langle y - X\alpha, X\alpha \rangle = \langle r, \hat{y} \rangle;$$

that is, for the optimal α, the prediction $\hat{y}$ and the residual vector r are orthogonal. Since $\hat{y} = X\alpha$ is restricted to the $(d+1)$-dimensional span of the columns of X, and α minimizes $\|r\|^2$, then this orthogonality implies that r is the *normal* vector to this $(d+1)$-dimensional subspace.

```python
import numpy as np
from numpy import linalg as LA

# directly
alpha = np.dot(np.dot(LA.inv(np.dot(X.T,X)),X.T),y.T)

# or with LA.lstsq
alpha = LA.lstsq(X,y)[0]
```

5.3 Polynomial Regression

Sometimes linear relations are not sufficient to capture the true pattern going on in the data with even a single dependent variable x. Instead we would like to build a model of the form:

$$\hat{y} = M_2(x) = \alpha_0 + \alpha_1 x + \alpha_2 x^2$$

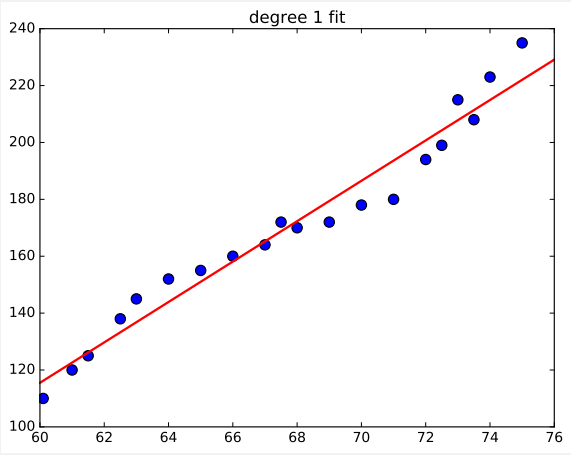
or more generally for some polynomial of degree p

$$\begin{aligned}\hat{y} = M_p(x) &= \alpha_0 + \alpha_1 x + \alpha_2 x^2 + \ldots + \alpha_p x^p \\ &= \alpha_0 + \sum_{i=1}^{p} \alpha_i x^i.\end{aligned}$$

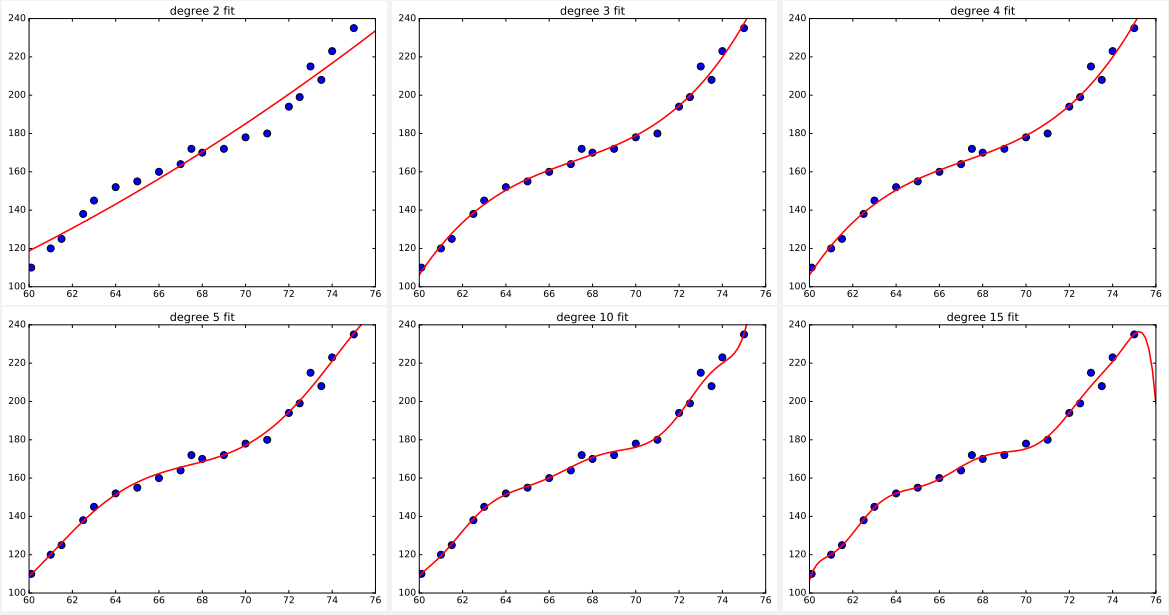
Example: Predicting Height and Weight with Polynomials

We found more height and weight data, in addition to the ones in the height-weight example above.

height (in)	weight (lbs)
61.5	125
73.5	208
62.5	138
63	145
64	152
71	180
69	172
72.5	199
72	194
67.5	172



But can we do better if we fit with a polynomial?



Again we can measure error for a single data point (x_i, y_i) as a residual as $r_i = |\hat{y} - y_i| = |M_\alpha(x_i) - y_i|$ and the error on n data points as the sum of squared residuals

$$\mathsf{SSE}(P, M_\alpha) = \sum_{i=1}^{n} r_i^2 = \sum_{i=1}^{n} (M_p(x_i) - y_i)^2.$$

Under this error measure, it turns out we can again find a simple solution for the residuals $\alpha = (\alpha_0, \alpha_1, \ldots, \alpha_p)$. For each dependent variable data value x we create a $(p+1)$-dimensional vector

$$v = (1, x, x^2, \ldots, x^p).$$

And then for n data points $(x_1, y_1), \ldots, (x_n, y_n)$ we can create an $n \times (p+1)$ data matrix

$$\tilde{X}_p = \begin{bmatrix} 1 & x_1 & x_1^2 & \ldots & x_1^p \\ 1 & x_2 & x_2^2 & \ldots & x_2^p \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \ldots & x_n^p \end{bmatrix} \quad y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ v_n \end{bmatrix}.$$

Then we can solve the same way as if each data value raised to a different power was a different dependent variable. That is we can solve for the coefficients $\alpha = (\alpha_0, \alpha_1, \alpha_2, \ldots, \alpha_p)$ as

$$\alpha = (\tilde{X}_p^T \tilde{X}_p)^{-1} \tilde{X}_p^T y.$$

5.4 Cross Validation

So it appears, as we increase p larger and larger, the data is fit better and better. The only downside appears to be the number of columns needed in the matrix $\tilde{X}_p$, right? Unfortunately, that is not right. Then how (and why?) do we choose the correct value of p, the degree of the polynomial fit?

A (very basic) statistical (hypothesis testing) approach may be choose a model of the data (the best fit curve for some polynomial degree p, and assume Gaussian noise), then calculate the probability that the data fell outside the error bounds of that model. But maybe many different polynomials are a good fit?

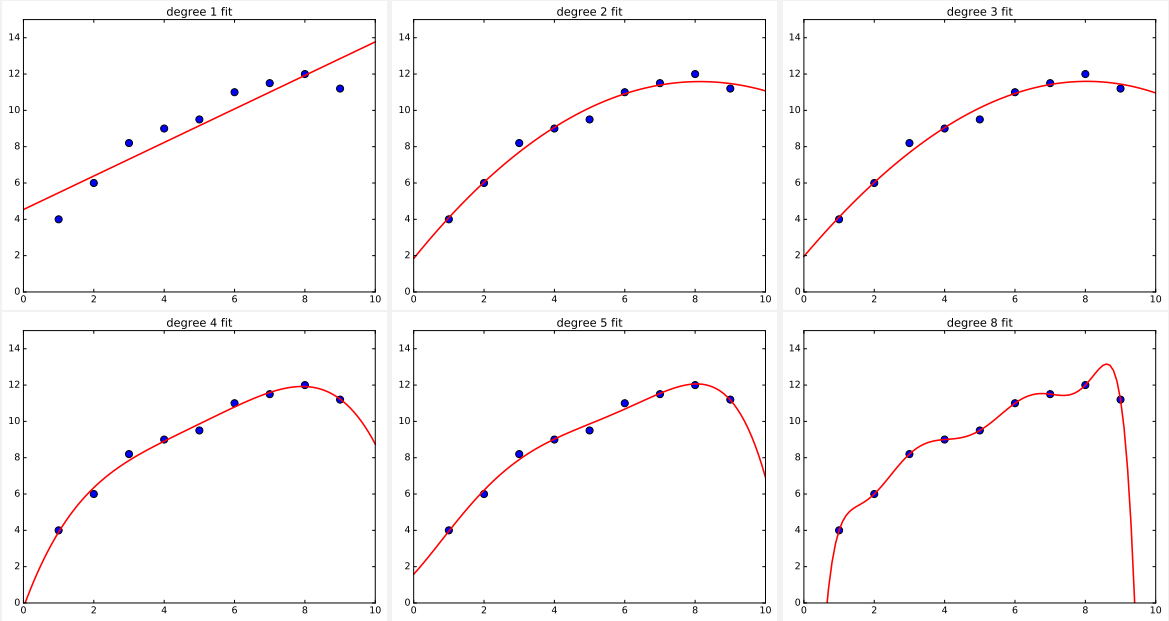
In fact, if we choose p as $n - 1$ or greater, then the curve will *polynomially interpolate* all of the points. That is, it will pass through all points, so all points have a residual of exactly 0 (up to numerical precision). This is the basis of a lot of geometric modeling (e.g., for CAD), but it turns out bad for data modeling.

Example: Simple polynomial example

Consider the simple data set of 9 points

x	1	2	3	4	5	6	7	8	9
y	4	6	8.2	9	9.5	11	11.5	12	11.2

With the following polynomial fits for $p = \{1, 2, 3, 4, 5, 8\}$. Believe your eyes, for $p = 8$, the curve actually passes through each and every point exactly.



Recall, our goal was for a new data point with only an x value to predict its y-value. Which do you think does the best job?

Generalization and Cross-Validation. Our ultimate goal in regression is *generalization* (how well do we predict on new data), not SSE! Using some error measure (SSE) to fit a line or curve, is a good proxy for what we want, but in many cases (as with polynomial regression), it can be abused. We want to know how our model will generalize to new data. How would we measure this without new data?

The solution is *cross-validation*. In the simplest form, we **randomly** split our data into training data (on which we build a model) and testing data (on which we evaluate our model). The testing serves to estimate how well we would do on future data which we do not have.

- *Why randomly?:* Because you do not want to bias the model to do better on some parts than other in how you choose the split. Also, since we assume the data elements come iid from some underlying distribution, then the test data is also iid if you chose it randomly.

- *How large should the test data be?:* It depends on the data set. Both 10% and 33% are common.

Let (X, y) be the full data set (with n rows of data), and we split it into data sets $(X_{\text{train}}, y_{\text{train}})$ and $(X_{\text{test}}, y_{\text{test}})$ with n_{train} and n_{test} rows, respectively. With $n = n_{\text{train}} + n_{\text{test}}$. Next we build a model with the training data, e.g.,

$$\alpha = (X_{\text{train}}^T X_{\text{train}})^{-1} X_{\text{train}}^T y_{\text{train}}.$$

Then we evaluate the model M_α on the test data X_{test}, often using $\mathsf{SSE}(X_{\text{test}}, y_{\text{test}}, M_\alpha)$ as

$$\mathsf{SSE}(X_{\text{test}}, y_{\text{test}}, M_\alpha) = \sum_{(x_i,y_i)\in(X_{\text{test}},y_{\text{test}})} (y_i - M_\alpha(x_i))^2 = \sum_{(x_i,y_i)\in(X_{\text{test}},y_{\text{test}})} (y_i - \langle (x_i;1), \alpha\rangle)^2.$$

We can use the testing data for two purposes:

- To estimate how well our model would perform on new data, yet unseen. That is the predicted residual of a new data point is precisely $\mathsf{SSE}(X_{\text{test}}, y_{\text{test}}, M_\alpha)/n_{\text{test}}$.
- To choose the correct parameter for a model (which p to use)?

Its important to not use the same $(X_{\text{test}}, y_{\text{test}})$ to do both tasks. If we choose a model with $(X_{\text{test}}, y_{\text{test}})$, then we should reserve even more data for predicting the generalization error. When using the test data to choose a model parameter, then it is being used to build the model; thus evaluating generalization with this same data can suffer the same fate as testing and training with the same data.

So how should we choose the best p? We calculate models M_{α_p} for each value p on the same training data. Then calculate the model error $\mathsf{SSE}(X_{\text{test}}, y_{\text{test}}, M_{\alpha_p})$ for each p, and see which has the smallest value. That is we train on $(X_{\text{train}}, y_{\text{train}})$ and test(evaluate) on $(X_{\text{test}}, y_{\text{test}})$.

Example: Simple polynomial example with Cross Validation

Now split our data sets into a train set and a test set:

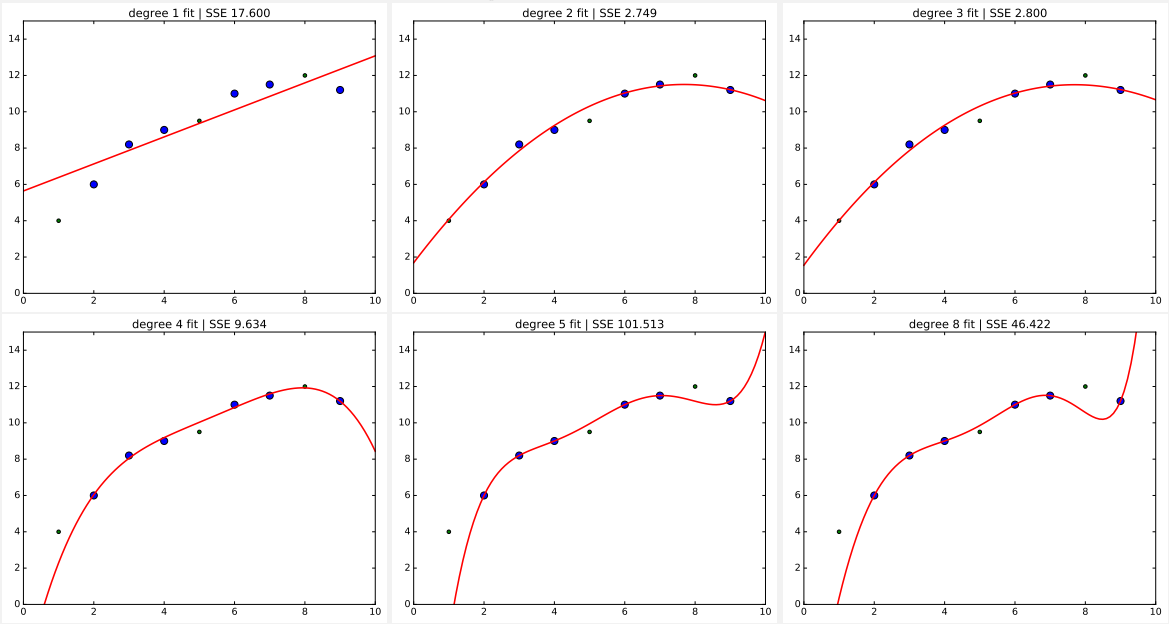
train:

x	2	3	4	6	7	8
y	6	8.2	9	11	11.5	12

test:

x	1	5	9
y	4	9.5	11.2

With the following polynomial fits for $p = \{1, 2, 3, 4, 5, 8\}$ generating model M_{α_p} on the test data. We then calculate the $\mathsf{SSE}(x_{\text{test}}, y_{\text{test}}, M_{\alpha_p})$ score for each (as shown):



And the polynomial model with degree $p = 2$ has the lowest SSE score of 2.749. It is also the simplest model that does a very good job by the "eye-ball" test. So we would choose this as our model.

Leave-one-out Cross Validation. But, not training on the test data means that you use less data, and your model is worse! If your data is limited, you may not want to "waste" this data!

If your data is very large, then leaving out 10% is not a big deal. But if you only have 9 data points it can be. The smallest the test set could be is 1 point. But then it is not a very good representation of the full data set.

The alternative is to create n different training sets, each of size $n-1$ ($X_{1,\text{train}}, X_{2,\text{train}}, \ldots, X_{n,\text{train}}$), where $X_{i,\text{train}}$ contains all points except for x_i, which is a one-point test set. Then we build n different models $M_1, M_2, \ldots, M_n$, evaluate each model M_i on the one test point x_i to get an error $E_i = (y_i - M_i(x_i))^2$, and average their errors $E = \frac{1}{n}\sum_{i=1}^{n} E_i$. Again, the parameter with the smallest associated average error E is deemed the best. This allows you to build a model on as much data as possible, while still using all of the data to test.

However, this requires roughly n times as long to compute as the other techniques, so is often too slow for really big data sets.

```
import matplotlib as mpl
mpl.use('PDF')
import matplotlib.pyplot as plt
import scipy as sp
import numpy as np
import math
from numpy import linalg as LA

def plot_poly(x,y,xE,yE,p):
  plt.scatter(x,y, s=80, c="blue")
  plt.scatter(xE,yE, s=20, c="green")
  plt.axis([0,10,0,15])

  s=sp.linspace(0,10,101)

  coefs=sp.polyfit(x,y,p)
  ffit = np.poly1d(coefs)
  plt.plot(s,ffit(s),'r-',linewidth=2.0)

  #evaluate on xE, yE
  resid = ffit(xE)
  RSSE = LA.norm(resid-yE)
  SSE = RMSE * RMSE

  title = "degree␣%s␣fit␣|␣SSE␣%0.3f" % (p, SSE)
  plt.title(title)
  file = "CVpolyReg%s.pdf" % p
  plt.savefig(file, bbox_inches='tight')
  plt.clf()
  plt.cla()



# train data
xT = np.array([2, 3,   4,  6,  7,     9])
yT = np.array([6, 8.2, 9,  11, 11.5,  11.2])

#test data
xE = np.array([1, 5,   8])
yE = np.array([4, 9.5, 12])
```



```
p_vals = [1,2,3,4,5,8]
for i in p_vals:
  plot_poly(xT,yT,xE,yE,i)
```

5.5 Regularized Regression

Returning to linear regression with multiple explanatory variables, cross-validation can again be useful to help tune our model. This at first may seem contradictory, as there is one optimal model which minimizes SSE; but we will see this is useful when there are more dimensions d (the explanatory variables) than needed to build a model from n data points. Indeed these methods can extend the least squares regression framework to when d is greater than n, which from a linear algebraic perspective seems impossible since then the SSE solution is under-defined.

Gauss-Markov Theorem Linear regression is in some sense optimal, as formalized by the Gauss-Markov Theorem. This states that, for data (X, y), the linear model M_α derived from $\alpha = (X^T X)^{-1} X^T y$ is the best possible, conditioned on three things:

- best is defined as lowest variance in residuals (the SSE cost we study)
- the solution has 0 expected error, and
- all errors $\varepsilon_i = x_i\alpha - y_i$ are not known to be correlated.

It is important to examine these benign seeming conditions. There are reasonable alternatives to the first condition, but studying the SSE is ultimately reasonable, it is the *minimum variance* solution. The third condition can in some cases be challenged as well, but requires more information than we typically assume and will involve ideas beyond the scope of this text. The second condition seems the most benign – why would we want to expect to have non-zero error? But this condition, it turns out, is the most ripe to exploit in ways that improve upon ordinary linear squares regression.

5.5.1 Tikhonov Regularization for Ridge Regression

We first introduce Tikhonov regularization, which results in a parameterized linear regression known as *ridge regression*. Consider again a data set (X, y) with n data points. Now instead of minimizing α over

$$\mathsf{SSE}(X, y, M_\alpha) = \sum_{i=1}^{n} (y_i - M_\alpha(x_i))^2 = \|X\alpha - y\|_2^2$$

we introduce a new cost function for a parameter $s > 0$ as

$$W_\circ(X, y, \alpha, s) = \frac{1}{n}\|X\alpha - y\|_2^2 + s\|\alpha\|_2^2.$$

This cost function is similar to before, except it adds a *regularization* term $+s\|\alpha\|_2^2$. This term is adding a bias towards α, and in particular, it is biasing α to be smaller; if there are two vectors α_1 and α_2 such that $\|X\alpha_1 - y\|_2^2 = \|X\alpha_2 - y\|_2^2$, then this cost function will prefer whichever has a smaller norm. In fact this will also prefer some vectors (representing models) α which do not correspond to the smallest values $\|X\alpha - y\|_2^2$, since they will make up for it with an even smaller term $s\|\alpha\|_2^2$. Indeed the larger that parameter s, the more a small norm of α is favored versus its fit to the data.

The normalization of the $\|X\alpha - y\|_2^2$ by $1/n$ makes it easier to understand the $s\|\alpha\|_2^2$ as the number of data points n change. This essentially compares the $s\|\alpha\|_2^2$ term with the effect of a single data point.

Geometry of Favoring a Small $\|\alpha\|$

Consider the simple example with one explanatory variable. In this case, the α_1 parameter corresponds to the slope of the line. Very large sloped models are inherently unstable. A small change in the x-value will lead to large changes in the y value (i.e., if $\hat{y} = \alpha_1 x$ then $\hat{y}_\delta = \alpha_1(x+\delta) = \hat{y} + \alpha_1\delta$. This picture only shows the affect of α with one explanatory variable, but this is most useful with many explanatory variables. In this setting, it is more common to *extrapolate*: make predictions on new x' which are outside the region occupied by the data X. In higher dimensions, it is hard to cover the entire domain with observations. And in these settings where predictions are made past the extent of data, it is even more dangerous to have parameters α_j which seem to fit the data locally, but change rapidly beyond the extent of the data.
Hence, smaller α are simpler models that are more cautious in extrapolated predictions, and so the predictions $\hat{y}$ are less sensitive to changes in x.

Moreover, it turns out that solving for the solution α_s° which minimizes $W_\circ(X, y, \alpha, s)$ is as simple as for the ordinary least squares where $s = 0$. In particular

$$\alpha_s^\circ = \arg\min_\alpha W_\circ(X, y, \alpha, s) = (X^T X + (s/n)I)^{-1} X^T y.$$

where I is the $d \times d$ identity matrix.

Recall that the matrix inverse is not defined if the matrix is not full rank. So in the case where X has fewer data points (n) than dimensions (d), then ordinary least squares (with $s = 0$) will not be solvable this way. Moreover, even if $n > d$, but if some of the eigenvectors of $X^T X$ are very small, then this matrix inverse operation will be numerically unstable. Adding the $(s/n)I$ term explicitly adds (s/n) to all eigenvalues, hence making the operation more stable. Explicitly, it makes the regression biased but more stable with respect to directions within $\mathbb{R}^d$ where X has small covariance.

Improved generalization. Indeed, it can be shown that appropriately setting some $s > 0$ can "get around" the Gauss-Markov Theorem. Formally, consider data drawn iid from a distribution $(X, y) \sim \mu_\alpha$ where given some choice of x, then $y = \langle x, \alpha \rangle + \varepsilon$, where ε is unbiased noise. Then unless always $|\varepsilon| = 0$, there exists some $s > 0$ such that predictions on new data from the same μ_α will have smaller SSE using the optimum from $W_\circ$ than from ordinary least squares.

This improved error argument is non-constructive since we do not explicitly know the distribution μ_α from which our data is drawn (we only observe our data (X, y)). However, we can use cross-validation to select the best choice of s on some testing data.

5.5.2 Lasso

A surprisingly effective variant of this regularization is known as the *lasso*. It replaces the $W_\circ$ cost function, again using a parameter $s > 0$ as

$$W_\diamond(X, y, \alpha, s) = \frac{1}{n}\|X\alpha - y\|_2^2 + s\|\alpha\|_1.$$

This has the same general form as ridge regression, but it makes the subtle change in the norm of the regularizer from $\|\cdot\|_2^2$ to $\|\cdot\|_1$.

This alternative form of regularization has all of the basic robustness and stronger generalization properties associated with biasing towards as a smaller normed choice of α in ridge regression. However, it also has two additional effects. First, the L_1 norm is less simple to optimize with respect to. Although there are algorithmic and combinatorial approaches, there is not a simple closed form using the matrix inverse

available anymore. We will discuss a couple ways to optimize this object soon, and then in the more general context of the following chapter. Second, this formulation has the additional benefit that it also biases towards sparser solutions. That is, for large enough values of s, the optimal choice of

$$\alpha_s^\diamond = \arg\min_\alpha W_\diamond(X, y, \alpha, s)$$

will have multiple coordinates $\alpha_{s,j}^\diamond = 0$. And, in general, as s increases the number of indexes j with $\alpha_{s,j}^\diamond = 0$ will increase.

Sparser models α, those with several coordinates $\alpha_j = 0$, are useful for various reasons. If the number of non-zero coordinates is small, it is simpler to understand and more efficient to use. That is, to make a prediction from a new data point x, only the coordinates which correspond to non-zero α_j values need to be considered. And while this does not definitively say which coordinates are actually meaningful towards understanding the dependent variable, it provides a small and viable set. The process of selecting such a subset is known *variable selection.* It is known that under some often reasonable assumptions, if the data is indeed drawn from a sparse model with noise, then using lasso to recover $\alpha_s^\diamond$ for the proper s can recover the true model exactly. This striking result is relatively recent, and already has many important implications in statistical inference, signal processing, and machine learning.

5.5.3 Dual Constrained Formulation

There is an equivalent formulation to both ridge regression and lasso that will make it easer to understand their optimization properties, and to describe a common approach used to solve for their solution. That is, instead of solving the modified objective functions $W_\circ$ and $W_\diamond$, we can solve the original least squares objective $\|X\alpha - y\|_2^2$, but instead provide a hard constraint on the allowable values of α.

Specifically, we can reformulate ridge regression for some parameter $t > 0$ as

$$\alpha_t^\circ = \arg\min_\alpha \|X\alpha - y\|_2^2 \quad \text{such that} \quad \|\alpha\|_2^2 \leq t$$

and lasso again using some $t > 0$, as

$$\alpha_t^\diamond = \arg\min_\alpha \|X\alpha - y\|_2^2 \quad \text{such that} \quad \|\alpha\|_1 \leq t.$$

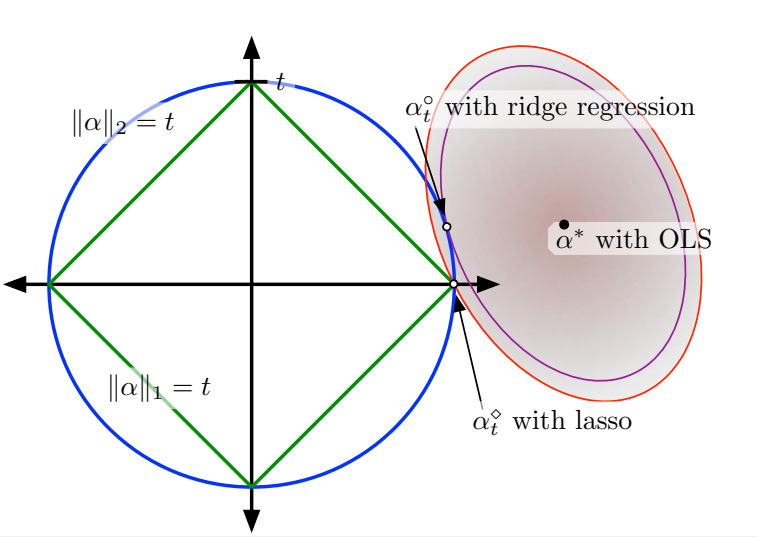
For ridge regression for each parameter $s > 0$ and solution α_s° there is a corresponding parameter $t > 0$ so $\alpha_t^\circ = \alpha_s^\circ$. And respectively, for lasso for each s there is a t so $\alpha_s^\diamond = \alpha_t^\diamond$. To see this, for a fixed s, find the solution α_s° (e.g., using the closed form solution), then set $t = \|\alpha_s^\circ\|$. Now using this value of t, the solution α_t° will match that of α_s° since it satisfies the hard norm constraint, and if there was another solution α satisfying that norm constraint with a smaller $\|X\alpha - y\|$ value, this would be smaller in both terms of $W_\circ$ than α_s°, a contradiction to the optimality of α_s°.

Hence, we can focus on solving either the soft norm formulations $W_\circ$ and $W_\diamond$ formulations or the hard norm formulations, for ridge regression or lasso. Ultimately, we will search over the choices of parameters s or t using cross-validation, so we do not need to know the correspondence ahead of time.

Geometry of Sparse Models with Lasso

Consider the hard constraint variants of lasso and ridge regression with some parameter t. We will visualize the geometry of this problem when there are $d = 2$ explanatory variables. The value of t constrains the allowable values of α to be within metric balls around the origin. These are shown below as an L_1-ball for lasso (in green) and an L_2-ball for ridge regression (in blue). Note that the constraint for ridge is actually $\|\alpha\|_2^2 \leq t$, where the norm is squared (this is the most common formulation, which we study), but we can always just consider the square root of that value t as the one we picture – the geometry is unchanged.

Then we can imagine the (unconstrained) ordinary least squares solution α^* as outside of both of these balls. The cost function $\|X\alpha - y\|_2^2$ is quadratic, so it must vary as a parabola as a function of α, with minimum at α^*. This is shown with red shading. Note that the values of α which have a fixed value in $\|X\alpha - y\|_2^2$ form concentric ellipses centered at α^*.



The key difference between ridge regression and lasso is now apparent in their solutions. Because the L_1-ball is "pointier," and specifically along the coordinate axes, it reaches out along these coordinate axes. Thus the innermost ellipse around α^* which the L_1-ball reaches tends to be along these coordinate axes – whereas there is no such property with the L_2-ball. These coordinate axes correspond with values of α which have 0 coordinates, thus the lasso solution with L_1 constraint explicitly biases towards sparse solutions.

The same phenomenon holds in higher dimensions (but which are harder to draw). However, in this setting, the L_1-ball is even pointier. The quadratic cost function now may not always first intersect the L_1-ball along a single coordinate axes (which has only a single non-zero coordinate), but it will typically intersect the L_1-ball along multi-dimensional ridges between coordinate axes, with only those coordinates as non-zero.

As t becomes smaller, it is more and more likely that the solution is found on a high-degree corner or ridge (with fewer non-zero coordinates), since it is harder for the convex ellipses to sneak past the pointy corners. As t becomes larger, the solution approaches the OLS solution a^*, and unless α^* itself has coordinates very close to 0, then it will tend to reach the L_1 ball away from those corners.

5.5.4 Orthogonal Matching Pursuit

As mentioned, there is not a simple closed form solution for lasso, but there is a common approach towards finding a good solution which retains its most important properties. And indeed running this procedure, *orthogonal matching pursuit* (OMP), does a good job of revealing these properties. When OMP is run with the lasso objective, then it is sometimes called *basis pursuit* or *forward subset selection* because it iteratively reveals a meaningful set of features or a basis within the data X which captures the trends with respect to the dependent variable.

Ethical Questions with Feature Selection

Lasso and Orthogonal Matching Pursuit can be used for feature selection, that is choosing a limited number of dimensions (the features) with which predictions from regression can generalize nearly as well, or in some cases better, than if all of the features are used. However, these approaches can be unstable when two features (or sets of features) are correlated. That is, there could be multiple subsets of features of the same size which provided nearly as good generalization.

Now consider you are working in a large business that pays for various sorts of indicators or features for customers or products it is trying to model. You notice that by small changes in the regularization parameter, two different subsets of features are selected, with both providing approximately the same generalization estimates when cross-validating. Changing which subsets you choose to purchase and use will dramatically affect the business of one of the companies trying to sell these features. Do you have an obligation to keep the financial benefits of this company in mind as you select features?

Alternatively, while the two subsets of features provide similar generalization predictions when averaged across the entire data set, on many individual data sets, it changes the predictions drastically. For instance, this may update the prediction for a product line to go from profitable to unprofitable. How should you investigate this prognostication before acting on it?

The OMP algorithm (sketched in Algorithm 5.5.1) uses a fixed value of s, initially sets $\alpha_j = 0$ in all coordinates, and iteratively chooses coordinates j which tend to minimize the objective $W_\diamond$ when made non-zero. It can also be run with $W_\circ$. As coordinates are chosen and assigned a value, a residual vector r of the part of y not yet captured is updated. Initially $r = y$, and as the algorithm is run and better fits are found $\|r\|_2^2$ decreases, as desired. In each successive round, the next best coordinate is selected using only r (not the original y); this is the key insight that makes this algorithm tractable. And in particular, this can be done by choosing the coordinate j which maximizes a linear operator (a dot product)

$$j^* = \arg\max_j |\langle r, X_j \rangle|$$

where X_j is the n-dimensional, jth column of the data matrix X. It is then possible to solve for α_{j^*} as

$$\alpha_{j^*} = \arg\min_\gamma \|r - X_j\gamma\|^2 + s|\gamma| = \frac{1}{\|X_j\|^2}\left(\langle r, X_j\rangle \pm \frac{s}{2}\right).$$

The choice of $\pm$ (either addition or subtraction of the $s/2$) term needs to be checked in the full expression.

This algorithm is greedy, and may not result in the true optimal solution. It may initially choose a coordinate which is not in the true optimum. And it may assign it a value α_j which is not the true optimum value. However, there are situations when the noise is small enough that this approach will still work. When using the $W_\circ$ objective for ridge regression, then at each step when solving for α_j, we can solve for all coordinates selected so far; and this is more robust to local minimum.

For either objective, this can be run until a fixed number k coordinates have been chosen (as in Algorithm 5.5.1), or until the residual's norm $\|r\|_2^2$ is below some threshold. For the feature selection goal, these

Algorithm 5.5.1 Orthogonal Matching Pursuit

Set $r = y$; and $\alpha_j = 0$ for all $j \in [d]$
for $i = 1$ **to** k **do**
  Set $X_j = \arg\max_{X_{j'} \in X} |\langle r, X_{j'} \rangle|$
  Set $\alpha_j = \arg\min_\gamma \|r - X_j\gamma\|^2 + s|\gamma|$
  Set $r = r - X_j\alpha_j$
Return α

coordinates are given an ordering where the more pertinent ones are deemed more relevant for the modeling problem.

Example: Orthogonal Matching Pursuit with $n < d$

Nevertheless, consider the data set (X, y) where X has $n = 5$ data points and $d = 7$ dimensions.

$$X = \begin{bmatrix} 1 & 8 & -3 & 5 & 4 & -9 & 4 \\ 1 & -2 & 4 & 8 & -2 & -3 & 2 \\ 1 & 9 & 6 & -7 & 4 & -5 & -5 \\ 1 & 6 & -14 & -5 & -3 & 9 & -2 \\ 1 & -2 & 11 & -6 & 3 & -5 & 1 \end{bmatrix} \quad y = \begin{bmatrix} -43.22 \\ -46.11 \\ -24.63 \\ 42.61 \\ -19.76 \end{bmatrix}$$

This was generated by applying a model $\alpha^T = [0, 0, 0, -5, 0, 2, 0]$ onto X and adding a small amount of noise to obtain y. Running ordinary least squares would fail since the system has more unknowns ($d = 7$) than equations ($n = 5$). However, we could run ridge regression; setting $s = 4.0$ fits a dense model $\alpha^\circ_{0.5} = [0.17, -0.23, -0.15, -4.76, 0.26, 1.86, -0.53]$.

Running OMP with the $W_\diamond$ objective, again using regularization parameter $s = 0.5$, does recover a sparse model. The first step identifies index $j = 4$ as having its column $X_4^T = [5, 8, -7, -5, -6]$ as being most correlated with $r = y$. Solving for $\alpha_4^* = -5.46$ which is large (in absolute value) than the optimal -5.0. We then update

$$r = r - X_4\alpha_4^* = [-15.91, -2.41, -13.60, 15.29, -13.01]$$

This suboptimal choice of α_4^* is still enough to reduce the norm of r from $\|r\| = 82.50$ to $\|r\| = 29.10$.
The next step again correctly selects $j = 5$ for column $X_6^T = [-9, -3, -5, 9, -5]$ having the most remaining alignment with r. It solves for $\alpha_6^* = 1.89$, less than the ideal 2.0. This updates

$$r = r - X_6\alpha_6^* = [1.16, 3.28, -4.12, -1.77, -3.52]$$

reducing the residual norm to $\|r\| = 6.68$.
If we run for another step, the algorithm will again choose $j = 4$, and now choose $\alpha_4^* = 0.44$. If we sum this with the previous choice, then the final value is -5.02, very close to the true model. After updating the residual norm goes down to $\|r\| = 1.77$.

```
import numpy as np
from numpy import linalg as LA

X = np.array([[1, 8, -3, 5, 4, -9, 4],[1,-2,4,8,-2,-3,2],[1,9,6,-7,4,-5,-5],
              [1,6,-14,-5,-3,9,-2],[1,-2,11,-6,3,-5,1]])
```



```
a = np.array([0,0,0,-5,0,2,0])
noise = np.random.normal(0,0.5,5)
y = np.dot(X,a) + noise
s = 4.0  # regularization parameter
k = 3    # number of iterations

print "norm: ", LA.norm(y), y

r = y
for i in range(k):

  # select column index most aligned with residual
  j = np.argmax(abs(np.dot(r,X)))

  # find best model parameter at j to fit residual
  ajp = (np.dot(r,X[:,j])+s/2) / (LA.norm(X[:,j])**2)
  ajm = (np.dot(r,X[:,j])-s/2) / (LA.norm(X[:,j])**2)
  if LA.norm(r-X[:,j]*ajp) + s*abs(ajp) < LA.norm(r-X[:,j]*ajm) + s*abs(ajm):
    aj = ajp
  else:
    aj = ajm

  # udpate residual
  r = r - X[:,j]*aj
  print "update: ", j, aj, LA.norm(r)

# Ordinary Least Squares
print "OLS: ", np.dot(np.dot(LA.inv(np.dot(X.T,X)),X.T),y.T)
# Ridge Regression
print "ridge: ", np.dot(np.dot(LA.inv(np.dot(X.T,X) + s*np.identity(7)),X.T),y.T)
```

An alternative to orthogonal matching pursuit is called least angle regression. Instead of explicitly adding one coordinate at a time for a fixed s, it follows the hard constraint objective. This method iteratively increases the constraint t while maintaining the optimal solution. With $t = 0$, then the solution $\alpha_t^\diamond = 0$. And as t increases, then initially a single coordinate is made non-zero in the optimal solution. Because the boundaries facets of the L_1-ball grow at a simple linear rate as t increases, then we can exactly describe the increases in the solution to $\alpha_t^\diamond$ as a linear equation of t. Moreover, solving another quadratic equation can be used to determine when the next coordinate is made non-zero, until eventually all coordinates are non-zero, and the OLS solution α^* is recovered. This also builds a model $M_{\alpha_t^\diamond}$ for every value t, so it can be cross-validated as it is built.

Compressed Sensing. A demonstration of what sort of data can be recovered exactly using lasso, is shown in the compressed sensing problem. There are variants of this problem that show up in computer vision, astronomy, and medical imagining; we will examine a simple form.

First consider an unknown signal vector $s \in \mathbb{R}^d$. It is d-dimensional, but is known to have most of its coordinates as 0. In our case, we consider with $m \ll d$ non-zero coordinates, and for simplicity assume these have value 1. For example, imagine a telescope scanning the sky. For most snapshots, the telescope sees nothing (a 0), but occasionally it sees a star (a 1). This maps to our model when it takes d snapshots, and only sees m stars. Many other examples exists: a customer will buy m out of d products a store sells; an earthquake registers on only m out of d days on record; or only m out of d people test positive for a rare genetic marker. We will use as an example s as

$$s^T = [0\,1\,0\,0\,0\,1\,0\,0\,0\,0\,0\,0\,0\,0\,0\,1\,0\,0\,0\,1\,1\,0\,1\,0\,0\,0\,1\,0\,0\,1\,0\,0].$$

But this problem is interesting because, as mentioned, s is unknown. Instead we "sense" s with a known

compressed vector $x_i \in \{-1, 0, +1\}^d$. For example, let

$$x_i^T = [\text{-}1\,0\,1\,0\,1\,1\,\text{-}1\,1\,0\,\text{-}1\,0\,0\,1\,\text{-}1\,\text{-}1\,1\,0\,1\,0\,1\,\text{-}1\,\text{-}1\,\text{-}1\,0\,1\,0\,0\,\text{-}1\,0\,1\,0\,0].$$

The sensing of s with x_i is recorded as $y_i = \langle x_i, s \rangle$, which is a single scalar value (in our setting an integer). In our example

$$y_i = \langle x_i, s \rangle = 0\text{+}0\text{+}0\text{+}0\text{+}1\text{+}0\text{+}0\text{+}0\text{+}0\text{+}0\text{+}0\text{+}0\text{+}0\text{+}0\text{+}0\text{+}1\text{+}0\text{+}0\text{+}0\text{+}1\text{-}1\text{+}0\text{-}1\text{+}0\text{+}0\text{+}0\text{+}0\text{+}0\text{+}0\text{+}1\text{+}0\text{+}0 = 2.$$

Moreover, in general we will have n measurements values $y_1, y_2, \ldots, y_n$ using n known measurement vectors $x_1, x_2, \ldots, x_n$. Stacking the measurement values $y = [y_1; y_2; \ldots; y_n]$ and vectors into a matrix $X = [x_1; x_2; \ldots; x_n]$ results in the familiar form

$$y = Xs.$$

And the goal is to recover s using only y and X. This appears to be precisely a regression problem! However, the we are interested in the case where $d \gg n$. In particular, this can be solvable when $n = C \cdot m \log(d/m)$, for the constant $C \in (4, 20)$ depending on the recovery approach. Given our assumption that $m \ll d$ (e.g., $m = 50$ and $d = 10{,}000$) this is a remarkable compression, since for instance just storing the location of each 1 bit uses $\log d$ bits (its index in binary), for a total of about $m \log d$.

The simplest recovery approach is with Orthogonal Matching Pursuit; this requires the measurement size constant C closer to 20. This modified version of OMP is sketched in Algorithm 5.5.2. The key step is modified to choose the dimension j where the column in the measurement matrix X_j has the largest dot product with r. We guess this has a 1 bit, and then factor out the effect of the measurement matrix witnessing a 1 bit at that location in the residual r. This is repeated either for a fixed (m) number of steps, or until the residual becomes all 0.

Algorithm 5.5.2 Orthogonal Matching Pursuit for Compressed Sensing

Set $r = y$; and $s = [0; 0; \ldots, 0]$.
for $i = 1$ **to** m **do**
  Set $j = \arg\max_{j'} \langle r, X_{j'} \rangle$.
  Set $s_j = 1$.
  Set $r = r - X_j$.
Return s.

Example: Recovery in Compressed Sensing

Consider a specific example for running Orthogonal Matching Pursuit, this has $d = 10$, $m = 3$ and $n = 6$. Let the (unknown) input signal be

$$s = [0, 0, 1, 0, 0, 1, 0, 0, 1, 0].$$

Let the known measurement matrix be

$$X = \begin{bmatrix} 0 & 1 & 1 & -1 & -1 & 0 & -1 & 0 & -1 & 0 \\ -1 & -1 & 0 & 1 & -1 & 0 & 0 & -1 & 0 & 1 \\ 1 & -1 & 1 & -1 & 0 & -1 & 1 & 1 & 0 & 0 \\ 1 & 0 & -1 & 0 & 0 & 1 & -1 & -1 & 1 & 1 \\ -1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & -1 \\ 0 & 0 & -1 & -1 & -1 & 0 & -1 & 1 & -1 & 0 \end{bmatrix}$$

so for instance the first row $x_1 = (0, 1, 1, -1, -1, 0, -1, 0, -1, 0)$ yields measurement

$$\langle x_1, s \rangle = 0 + 0 + 1 + 0 + 0 + 0 + 0 + 0 + (-1) + 0 = 0.$$

The observed measurement vector is

$$y = Xs^T = [0, 0, 0, 1, 1, -2]^T.$$

Columns 9 has the most explanatory power towards y, based on X. We let $j = 9$ so $X_j = X_9 = (-1, 0, 0, 1, 1, -1)^T$. Then $s_9 = 1$ and $r = y - X_9 = (1, 0, 0, 0, 0, -1)$.

Next, we observe that column 3 has the most explanatory power for the new r. We set $s_3 = 1$ and update $r = r - X_3 = (0, 0, -1, 1, 0, 0)$. *Note: This progress seemed sideways at best. It increased our non-zero s values, but did not decrease $\|r - y\|$.*

Finally, we observe column 6 has the most explanatory power of the new r. We set $s_6 = 1$ and update $r = r - X_6\gamma_3 = (0, 0, 0, 0, 0, 0)$. We have now completely explained y using only 3 data elements.

This will not always work so cleanly on a small example. Using OMP typically needs something like $n = 20m \log d$ measurements (instead of $n = 6$). Larger measurement sets act like concentrating random variables, and the larger the sets the more likely that at each step we chose the correct index j as most explanatory.

Exercises

We will use a dataset found here:
`http://www.cs.utah.edu/~jeffp/teaching/FoDA/D3.csv`

Q5.1: Let the first column of the data set be the explanatory variable `x`, and let the fourth column be the dependent variable `y`. [That is: ignore columns 2 and 3 for now]

1. Run simple linear regression to predict `y` from `x`. Report the linear model you found. Predict the value of `y` for new `x` values 0.3, for 0.5, and for 0.8.
2. Use cross-validation to predict generalization error, with error of a single data point (`x`, `y`) from a model M as $(M(\texttt{x}) - \texttt{y})^2$. Describe how you did this, and which data was used for what.
3. On the same data, run polynomial regression for $p = 2, 3, 4, 5$. Report polynomial models for each. With each of these models, predict the value of `y` for a new `x` values of 0.3, for 0.5, and for 0.8.
4. Cross-validate to choose the best model. Describe how you did this, and which data was used for what.

Q5.2: Now let the first three columns of the data set be separate explanatory variables `x1`, `x2`, `x3`. Again let the fourth column be the dependent variable `y`.

- Run linear regression simultaneously using all three explanatory variables. Report the linear model you found. Predict the value of `y` for new (`x1`,`x2`,`x3`) values $(0.3, 0.4, 0.1)$, for $(0.5, 0.2, 0.4)$, and for $(0.8, 0.2, 0.7)$.
- Use cross-validation to predict generalization error; as usual define the error of a single data point (`x1`, `x2`, `x3`, `y`) from a model M as $(M(\texttt{x1}, \texttt{x2}, \texttt{x3}) - \texttt{y})^2$. Describe how you did this, and which data was used for what.

Q5.3: Consider a data set (X, y) where $X \in \mathbb{R}^{n \times 3}$; and its decomposition into a test $(X_{\mathsf{test}}, y_{\mathsf{test}})$ and a training data set $(X_{\mathsf{train}}, y_{\mathsf{train}})$. Assume that X_{train} is not just a subset of X, but also prepends a columns of all 1s. We build a linear model

$$\alpha = (X_{\mathsf{train}}^T X_{\mathsf{train}})^{-1} X_{\mathsf{train}}^T y_{\mathsf{train}}.$$

where $\alpha \in \mathbb{R}^4$. The test data $(X_{\mathsf{test}}, y_{\mathsf{test}})$ consists of two data points: (x_1, y_1) and (x_2, y_2), where $x_1, x_2 \in \mathbb{R}^3$. Explain how to use (write a mathematical expression) this test data to estimate the generalization error. That is, if one new data point arrives x, how much squared error would we expect the model α to have compared to the unknown true value y?

6 Gradient Descent

In this topic we will discuss optimizing over general functions f. Typically the function is defined $f : \mathbb{R}^d \to \mathbb{R}$; that is its domain is multi-dimensional (in this case d-dimensional point α) and output is a real scalar ($\mathbb{R}$). This often arises to describe the "cost" of a model which has d parameters which describe the model (e.g., degree $(d-1)$-polynomial regression) and the goal is to find the parameters $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_d)$ with minimum cost. Although there are special cases where we can solve for these optimal parameters exactly, there are many cases where we cannot. What remains in these cases is to analyze the function f, and try to find its minimum point. The most common solution for this is gradient descent where we try to "walk" in a direction so the function decreases until we no longer can.

6.1 Functions

We review some basic properties of a function $f : \mathbb{R}^d \to \mathbb{R}$. Again, the goal will be to unveil abstract tools that are often easy to imagine in low dimensions, but automatically generalize to high-dimensional data. We will first provide definitions without any calculous.

Let $B_r(\alpha)$ define a Euclidean ball around a point $\alpha \in \mathbb{R}^d$ of radius r. That is, it includes all points $\{p \in \mathbb{R}^d \mid \|\alpha - p\| \leq r\}$, within a Euclidean distance of r from α. We will use $B_r(\alpha)$ to define a *local neighborhood* around a point α. The idea of "local" is quite flexible, and we can use *any* value of $r > 0$, basically it can be as small as we need it to be, as long as it is strictly greater than 0.

Minima and maxima. A *local maximum* of f is a point $\alpha \in \mathbb{R}^d$ so for some neighborhood $B_r(\alpha)$, all points $p \in B_r(\alpha)$ have smaller (or equal) function value than at α: $f(p) \leq f(\alpha)$. A *local minimum* of f is a point $\alpha \in \mathbb{R}^d$ so for some neighborhood $B_r(\alpha)$, all points $p \in B_r(\alpha)$ have larger (or equal) function value than at α: $f(p) \geq f(\alpha)$. If we remove the "or equal" condition for both definitions for $p \in B_r(\alpha), p \neq \alpha$, we say the maximum or minimum points are *strict*.

A point $\alpha \in \mathbb{R}^d$ is a *global maximum* of f if for all $p \in \mathbb{R}^d$, then $f(p) \leq f(\alpha)$. Likewise, a point $\alpha \in \mathbb{R}^d$ is a *global minimum* if for all $p \in \mathbb{R}^d$, then $f(p) \geq f(\alpha)$. There may be multiple global minimum and maximum. If there is exactly one point $\alpha \in \mathbb{R}^d$ that is a global minimum or global maximum, we again say it is *strict*.

When we just use the term *minimum* or *maximum* (without local or global) it implies a local minimum or maximum.

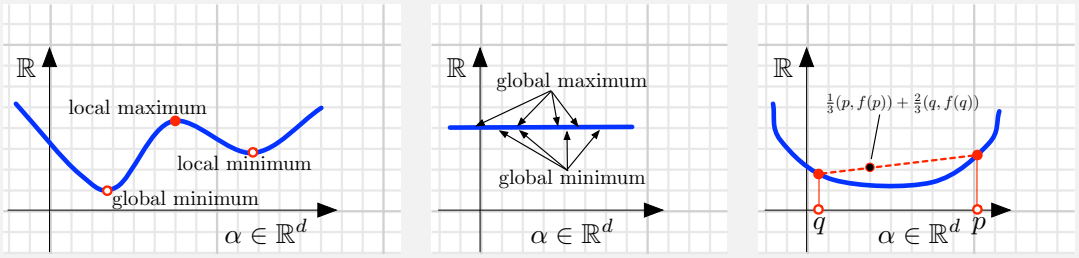
Focusing on a function restricted to a closed and bounded subset $S \subset \mathbb{R}^d$, if the function is continuous (that is, there exists a δ such that for all $\alpha \in \mathbb{R}^d$, there exists a radius r_δ such that if $p \in B_{r_\delta}(\alpha)$, then $|f(\alpha) - f(p)| \leq \delta$), then the function must have a global minimum and a global maximum. It may occur on the boundary of S.

A *saddle* point $\alpha \in \mathbb{R}^d$ (for $d > 1$) has within any neighborhood $B_r(\alpha)$ a points $p \in B_r(\alpha)$ with $f(p) < f(\alpha)$ (the lower points) and $p' \in B_r(\alpha)$ with $f(p') > f(\alpha)$ (the upper points). In particular, it is a saddle if within $B_r(\alpha)$ there are disconnected regions of upper points (and of lower points). The notion of saddle point is defined differently for $d = 1$. If these regions are connected, and it is not a minimum or maximum, then it is a *regular point*.

For an arbitrary (or randomly) chosen point α, it is usually a regular point (except for examples you are unlikely to encounter, the set of minimum, maximum, and saddle points are finite, while the set of regular points is infinite).

Example: Continuous Functions

Here we show some example functions, where the α-value on the x-axis represents a d-dimensional space. The first function has local minimum and maximum. The second function has a constant value, so every point is a global minimum and a global maximum. The third function f is convex, which is demonstrated with the line segment between points $(p, f(p))$ and $(q, f(q))$ is always above the function f.



Convex functions. In many cases we will assume (or at least desire) that our function is convex.

To define this it will be useful to define a line $\ell \subset \mathbb{R}^d$ as follows with any two points $p, q \in \mathbb{R}^d$. Then a line $\ell_{p,q}$ is the set of all points defined by any scalar $\lambda \in \mathbb{R}$ as

$$\ell_{p,q} = \{x = \lambda p + (1-\lambda)q \mid \lambda \in \mathbb{R}\}.$$

When $\lambda \in [0, 1]$, then this defines the line segment between p and q.

A function is *convex* if for any two points $p, q \in \mathbb{R}^d$, on the line segment between them has value less than (or equal) to the values at the weighted average of p and q. That is, it is convex if

$$\text{For all } p, q \in \mathbb{R} \quad \text{and for all } \lambda \in [0, 1] \quad f(\lambda p + (1-\lambda)q) \leq \lambda f(p) + (1-\lambda) f(q).$$

Removing the "or equal" condition, the function becomes *strictly convex*.

There are many very cool properties of convex functions. For instance, for two convex functions f and g, then $h(\alpha) = f(\alpha) + g(\alpha)$ is convex and so is $h(\alpha) = \max\{f(\alpha), g(\alpha)\}$. But one will be most important for us:

Convexity and Global Minimums: Any local minimum of a convex function will also be a global minimum. A strictly convex function will have at most a single minimum: the global minimum.

This means if we find a minimum, then we must have also found a global minimum (our goal).

6.2 Gradients

For a function $f(\alpha) = f(\alpha_1, \alpha_2, \ldots, \alpha_d)$, and a unit vector $u = (u_1, u_2, \ldots, u_d)$ which represents a direction, then the *directional derivative* is defined

$$\nabla_u f(\alpha) = \lim_{h \to 0} \frac{f(\alpha + hu) - f(\alpha)}{h}.$$

We are interested in functions f which are *differentiable*; this implies that $\nabla_u f(\alpha)$ is well-defined for all α and u. The converse is not necessarily true.

Let $e_1, e_2, \ldots, e_d \in \mathbb{R}^d$ be a specific set of unit vectors so that $e_i = (0, 0, \ldots, 0, 1, 0, \ldots, 0)$ where for e_i the 1 is in the ith coordinate.

Then define

$$\nabla_i f(\alpha) = \nabla_{e_i} f(\alpha) = \frac{\mathrm{d}}{\mathrm{d}\alpha_i} f(\alpha).$$

It is the derivative in the ith coordinate, treating all other coordinates as constants.

We can now, for a differentiable function f, define the *gradient of* f as

$$\nabla f = \frac{\mathrm{d}f}{\mathrm{d}\alpha_1} e_1 + \frac{\mathrm{d}f}{\mathrm{d}\alpha_2} e_2 + \ldots + \frac{\mathrm{d}f}{\mathrm{d}\alpha_d} e_d = \left(\frac{\mathrm{d}f}{\mathrm{d}\alpha_1}, \frac{\mathrm{d}f}{\mathrm{d}\alpha_2}, \ldots, \frac{\mathrm{d}f}{\mathrm{d}\alpha_d} \right).$$

Note that ∇f is a function from $\mathbb{R}^d \to \mathbb{R}^d$, which we can evaluate at any point $\alpha \in \mathbb{R}^d$.

Example: Gradient

For $\alpha = (x, y, z) \in \mathbb{R}^3$, consider the function $f(x, y, z) = 3x^2 - 2y^3 - 2xe^z$. Then $\nabla f = (6x - 2e^z, -6y^2, -2xe^z)$ and $\nabla f(3, -2, 1) = (18 - 2e, 24, -6e)$.

Linear approximation. From the gradient we can easily recover the directional derivative of f at point α, for any direction (unit vector) u as

$$\nabla_u f(\alpha) = \langle \nabla f(\alpha), u \rangle.$$

This implies the gradient describes the linear approximation of f at a point α. The slope of the tangent plane of f at α in any direction u is provided by $\nabla_u f(\alpha)$.

Hence, the direction which f is increasing the most at a point α is the unit vector u where $\nabla_u f(\alpha) = \langle \nabla f(\alpha), u \rangle$ is the largest. This occurs at $\overline{\nabla f(\alpha)} = \nabla f(\alpha) / \|\nabla f(\alpha)\|$, the normalized gradient vector.

To find the minimum of a function f, we then typically want to move from any point α in the direction $-\overline{\nabla f(\alpha)}$; at regular points this is the direction of steepest descent.

6.3 Gradient Descent

Gradient descent is a family of techniques that, for a differentiable function $f : \mathbb{R}^d \to \mathbb{R}$, try to identify either

$$\min_{\alpha \in \mathbb{R}^d} f(\alpha) \qquad \text{and/or} \qquad \alpha^* = \arg\min_{\alpha \in \mathbb{R}^d} f(\alpha).$$

This is effective when f is convex and we do not have a "closed form" solution α^*. The algorithm is iterative, in that it may never reach the completely optimal α^*, but it keeps getting closer and closer.

Algorithm 6.3.1 Gradient Descent$(f, \alpha_{\text{start}})$

initialize $\alpha^{(0)} = \alpha_{\text{start}} \in \mathbb{R}^d$.
repeat
  $\alpha^{(k+1)} := \alpha^{(k)} - \gamma_k \nabla f(\alpha^{(k)})$
until $(\|\nabla f(\alpha^{(k)})\| \leq \tau)$
return $\alpha^{(k)}$

Basically, for any starting point $\alpha^{(0)}$ the algorithm moves to another point in the direction opposite to the gradient, in the direction that locally decreases f the fastest. How fast it moves depends on the scalar learning rate γ_k and the magnitude of the gradient vector $\nabla f(\alpha^{(k)})$.

Stopping condition. The parameter τ is the tolerance of the algorithm. If we assume the function is differentiable, then at the minimum α^*, we must have that $\nabla f(\alpha) = (0, 0, \ldots, 0)$. So for α close to the minimum, $\nabla f(\alpha)$ should also have a small norm. The algorithm may never reach the true minimum (and we do not know what it is, so we cannot directly compare against the function value). So we use $\|\nabla f\|$ as a proxy.

In other settings, we may run for a fixed number T steps. Although this does not automatically tune the algorithm to the input, as using a tolerance τ may, it is easier to describe and compare. Hence, most examples in this text will use this method.

6.3.1 Learning Rate

The most critical parameter of gradient descent is γ, the *learning rate*. In many cases the algorithm will keep $\gamma_k = \gamma$ fixed for all k. It controls how fast the algorithm works. But if it is too large, when we approach the minimum, then the algorithm may go too far, and overshoot it.

How should we choose γ? There is no consensus to this answer, and often in practice it is tuned in ad-hoc ways. In the following, we will describe some mathematically described scenarios where something formal can be said about how to choose γ. In some cases, these analysis show that if the function satisfies a mild property, then many fixed choices of γ will result in accuracy guarantees. We will also show methods where it helps to adjust the learning parameter γ_k adaptively.

Lipschitz bound. We say a function $g : \mathbb{R}^d \to \mathbb{R}^k$ is *L-Lipschitz* if for all $p, q \in \mathbb{R}^d$ that

$$\|g(p) - g(q)\| \leq L\|p - q\|.$$

This property is useful when $g = \nabla f$ is describing the gradient of a cost function f. If ∇f is L-Lipschitz, and we set $\gamma \leq \frac{1}{L}$, then gradient descent will converge to a stationary point. Moreover, if f is convex with global minimum α^*, then after $k = O(1/\varepsilon)$ steps we can guarantee that

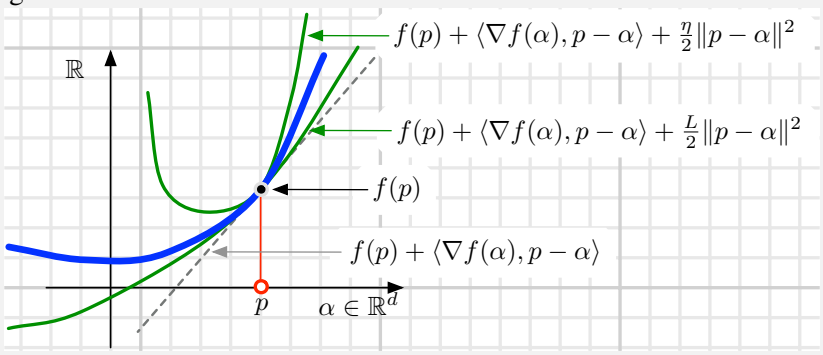
$$f(\alpha^{(k)}) - f(\alpha^*) \leq \varepsilon.$$

For the $k = O(1/\varepsilon)$ claim (and others stated below), we assume that $f(\alpha^{(0)}) - f(\alpha^*)$ is less than some absolute constant. Intuitively, the closer we start to the optimum, the fewer steps it will take.

For a convex quadratic function f (e.g., most cost functions derived by sum of squared errors), then the gradient ∇f is L-Lipschitz.

Example: Strongly Convex Function

We show an example η-strongly convex function f, in blue. At any point p, it is sandwiched between two convex quadratic functions in green. The convex quadratic function which lower bounds f has an L-Lipschitz gradient.



Strongly Convex Functions. A function $f : \mathbb{R}^d \to \mathbb{R}$ is *η-strongly convex* with parameter $\eta > 0$ if for all $\alpha, p \in \mathbb{R}^d$ then

$$f(p) \leq f(\alpha) + \langle \nabla f(\alpha), p - \alpha \rangle + \frac{\eta}{2}\|p - \alpha\|^2.$$

Intuitively, this implies that f is at least quadratic. That is, along any direction $u = \frac{p-\alpha}{\|p-\alpha\|}$ the function $\langle f, u \rangle$ is 1-dimensional, and then its second derivative $\frac{\mathrm{d}^2}{\mathrm{d}u^2}\langle f, u \rangle$ is strictly positive; it is at least $\eta > 0$. Similarly, saying a function f has an L-Lipschitz gradient is equivalent to the condition that $\frac{\mathrm{d}^2}{\mathrm{d}u^2}\langle f(\alpha), u \rangle \leq L$ for all $\alpha, p \in \mathbb{R}^d$ where $u = \frac{p-\alpha}{\|p-\alpha\|}$.

For an η-strongly convex function f, that has an L-Lipschitz gradient, with global minimum α^*, then gradient descent with learning rate $\gamma \leq 2/(\eta + L)$ after only $k = O(\log(1/\varepsilon))$ steps will achieve

$$f(\alpha^{(k)}) - f(\alpha^*) \leq \varepsilon.$$

The constant in $k = O(\log(1/\varepsilon))$ depends on the condition number L/η. The conditions of this bound, imply that f is sandwiched between two convex quadratic functions; specifically for any $\alpha, p \in \mathbb{R}^d$ that we can bound

$$f(\alpha) + \langle \nabla f(\alpha), p - \alpha \rangle + \frac{L}{2}\|p - \alpha\|^2 \leq f(p) \leq f(\alpha) + \langle \nabla f(\alpha), p - \alpha \rangle + \frac{\eta}{2}\|p - \alpha\|^2.$$

When an algorithm converges at such a rate (takes $O(\log(1/\varepsilon))$ steps to obtain ε error), it is known as *linear convergence* since the log-error $\log(f(\alpha^{(k)}) - f(\alpha^*))$ looks like a linear function of k.

In practice, since many functions we consider will be convex quadratic functions (e.g., are derived from sum of squared error cost functions), then the error will decreases exponentially fast in terms of the number of steps of gradient descent, if the learning rate is set sufficiently small. That is, only a constant number of steps are required to resolve each bit of precision in the function value at the optimum! However, if the learning rate is set too small, then the constant (number of steps to resolve one bit) will increase.

So at this point, we have explained that for many situations there is a learning rate for which gradient descent will work extremely well. If we can analytically bound various properties of the second derivative of the function, then we can use these bounds to choose such a rate. However, we have not yet explained a formal way to find such a rate in general where we can only evaluate the gradient at any point $\alpha \in \mathbb{R}^d$.

Line search. An alternative, referred to as "line search" is to solve for the (approximately) optimal γ_k at each step. Once we have computed the gradient $\nabla f(\alpha^{(k)})$ then we have reduced the high-dimensional minimization problem to a one-dimensional problem. Note if f is convex, then f restricted to this one-dimensional search is also convex. We still need to find the minimum of an unknown function, but we can perform some procedure akin to binary search. We first find a value γ' such that

$$f\left(\alpha^{(k)} - \gamma' \nabla f(\alpha^{(k)})\right) > f(\alpha^{(k)})$$

then we keep subdividing the region $[0, \gamma']$ into pieces, and excluding ones which cannot contain the minimum.

For instance the *golden section search* divides a range $[b, t]$ containing the optimal γ_k into three subintervals (based on the golden ratio) so $[b, t] = [b, b') \cup [b', t'] \cup (t', t]$. And each step, we can determine that either $\gamma_k \notin [b, b')$ if $f(t') < f(b')$, or $\gamma_k \notin (t', t]$ if $f(b') < f(t')$. This reduces the range to $[b', t]$ or $[b, t']$, respectively, and we recurse.

In other situations, we can solve for the optimal γ_k exactly at each step. This is the case if we can again analytically take the derivative $\frac{\mathrm{d}}{\mathrm{d}\gamma}\left(f(\alpha^{(k)}) - \gamma \nabla f(\alpha^{(k)})\right)$ and solve for the γ where it is equal to 0.

Adjustable rate. In practice, line search is often slow. Also, we may not have a Lipschitz bound. It is often better to try a few fixed γ values, probably being a bit conservative. As long as $f(\alpha^{(k)})$ keep decreasing, it works well. This also may alert us if there is more than one local minimum if the algorithm converges to different locations.

An algorithm called "backtracking line search" automatically tunes the parameter γ. It uses a fixed parameter $\beta \in (0,1)$ (preferably in $(0.1, 0.8)$; for instance use $\beta = 3/4$). Start with a large step size γ (e.g., $\gamma = 1$). Then at each step of gradient descent at location α, if

$$f(\alpha - \gamma \nabla f(\alpha)) > f(\alpha) - \frac{\gamma}{2}\|\nabla f(\alpha)\|^2$$

then update $\gamma = \beta\gamma$. This shrinks γ over the course of the algorithm, and if f is strongly convex, it will eventually decrease γ until it satisfies the condition for linear convergence.

Example: Gradient Descent with Fixed Learning Rate

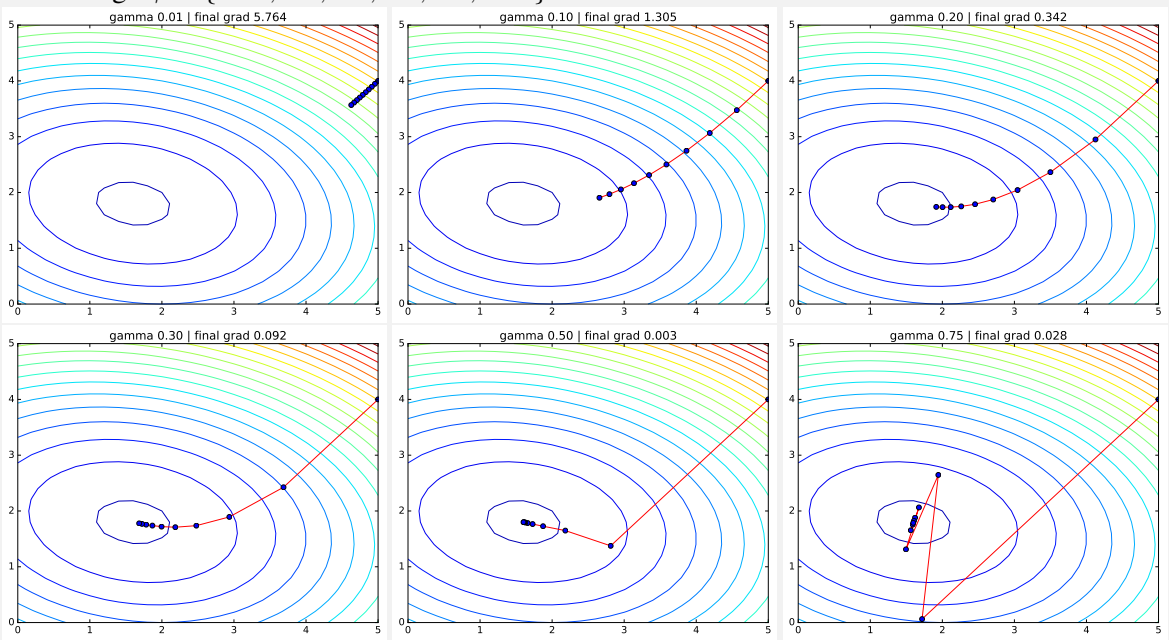
Consider the function f where $\alpha = (x, y) \in \mathbb{R}^2$ is defined

$$f(x,y) = (\frac{3}{4}x - \frac{3}{2})^2 + (y-2)^2 + \frac{1}{4}xy$$

and has gradient

$$\nabla f(x,y) = \left(\frac{9}{8}x - \frac{9}{4} + \frac{1}{4}y \;\;,\;\; 2y - 4 + \frac{1}{4}x\right).$$

We run gradient descent for 10 iterations within initial position $(5, 4)$, while varying the learning rate in the range $\gamma = \{0.01, 0.1, 0.2, 0.3, 0.5, 0.75\}$.



We see that with γ very small, the algorithm does not get close to the minimum. When γ is too large, then the algorithm jumps around a lot, and is in danger of not converging. But at a learning rate of $\gamma = 0.3$ it converges fairly smoothly and reaches a point where $\|\nabla f(x,y)\|$ is very small. Using $\gamma = 0.5$ almost overshoots in the first step; $\gamma = 0.3$ is smoother, and it is probably best to use a curve that looks smooth like that one, but with a few more iterations.

```
import matplotlib as mpl
mpl.use('PDF')
import numpy as np
import matplotlib.pyplot as plt
from numpy import linalg as LA

def func(x,y):
  return (0.75*x-1.5)**2 + (y-2.0)**2 + 0.25*x*y

def func_grad(vx,vy):
  dfdx = 1.125*vx - 2.25 + 0.25*vy
  dfdy = 2.0*vy - 4.0 + 0.25*vx
  return np.array([dfdx,dfdy])

#prepare for contour plot
xlist = np.linspace(0, 5, 26)
ylist = np.linspace(0, 5, 26)
x, y = np.meshgrid(xlist, ylist)
z = func(x,y)
lev = np.linspace(0,20,21)

#iterate location
v_init = np.array([5,4])
num_iter = 10
values = np.zeros([num_iter,2])

for gamma in [0.01, 0.1, 0.2, 0.3, 0.5, 0.75]:
  values[0,:] = v_init
  v = v_init

  # actual gradient descent algorithm
  for i in range(1,num_iter):
    v = v - gamma * func_grad(v[0],v[1])
    values[i,:] = v

  #plotting
  plt.contour(x,y,z,levels=lev)
  plt.plot(values[:,0],values[:,1],'r-')
  plt.plot(values[:,0],values[:,1],'bo')
  grad_norm = LA.norm(func_grad(v[0],v[1]))
  title = "gamma␣%0.2f␣|␣final␣grad␣%0.3f" % (gamma,grad_norm)
  plt.title(title)
  file = "gd-%2.0f.pdf" % (gamma*100)
  plt.savefig(file, bbox_inches='tight')
  plt.clf()
  plt.cla()
```

6.4 Fitting a Model to Data

For data analysis, the most common use of gradient descent is to fit a model to data. In this setting we have a data set $(X, y) = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\} \in \mathbb{R}^d \times \mathbb{R}$ and a family of models $\mathcal{M}$ so each possible model M_α is defined by a d-dimensional vector $\alpha = \{\alpha_1, \alpha_2, \ldots, \alpha_k\}$ for k parameters.

Next we define a *loss function* $L((X, y), M_\alpha)$ which measures the difference between what the model predicts and what the data values are. To choose which parameters generate the best model, we let $f(\alpha) : \mathbb{R}^d \to \mathbb{R}$ be our function of interest, defined $f(\alpha) = L((X, y), M_\alpha)$. Then we can run gradient descent to

find our model M_{α^*}. For instance we can set

$$f(\alpha) = L((X,y), M_\alpha) = \mathsf{SSE}(P, M_\alpha) = \sum_{(x_i,y_i)\in(X,y)} (y_i - M_\alpha(x_i))^2. \tag{6.1}$$

This is used for examples including maximum likelihood (or maximum log-likelihood) estimators from Bayesian inference. This includes finding a single point estimator with Gaussian (where we had a closed-form solution), but also many other variants (often where there is no known closed-form solution). It also includes least squares regression and its many variants; we will see this in much more detail next. And will include other topics (including clustering, PCA, classification) we will see later in class.

6.4.1 Least Mean Squares Updates for Regression

Now we will work through how to use gradient descent for simple quadratic regression on 1-dimensional explanatory variables. That is, this will specify the function $f(\alpha)$ in equation (6.1) to have $k = 3$ parameters as $\alpha = (\alpha_0, \alpha_1, \alpha_2)$, and so for each $(x_i, y_i) \in (X, y)$ we have $x_i \in \mathbb{R}$. Then we can write the model again as a dot product

$$M_\alpha(x_i) = \langle \alpha, (1, x_i, x_i^2)\rangle = \alpha_0 + \alpha_1 x_i + \alpha_2 x_i^2$$

It is straightforward to generalize to linear regression, multiple-explanatory variable linear regression, or general polynomial regression from here. For instance, in this case, we can represent x_i as the 3-dimensional feature vector $q = (q_0 = x_i^0 = 1, q_1 = x_i^1, q_2 = x_i^2)$ of explanatory variables. Then the model is $M_\alpha(x_i) = \langle \alpha, q\rangle$. For simple linear regression, we simply omit the quadratic term $\alpha_2 x_i^2$ from the above model. For other polynomial forms of regression, the expansion simply includes more terms and powers. And for multiple explanatory variables they are each given a corresponding feature coordinate, possible to some power, or multiplied with other explanatory variables in the polynomial models. We will continue using 1-dimensional quadratic regression as an example.

Now to specify the gradient descent step

$$\alpha := \alpha - \gamma \nabla f(\alpha)$$

we only need to define $\nabla f(\alpha)$. We will first show this for the case where $n = 1$, that is when there is a single data point (x_1, y_1). For quadratic regression, the cost function $f_1(\alpha) = (\alpha_0 + \alpha_1 x_1 + \alpha_2 x_1^2 - y_1)^2$ is convex. Next derive

$$\begin{aligned}
\frac{\mathrm{d}}{\mathrm{d}\alpha_j} f(\alpha) &= \frac{\mathrm{d}}{\mathrm{d}\alpha_j}(M_\alpha(x_1) - y_1)^2 \\
&= 2(M_\alpha(x_1) - y_1)\frac{\mathrm{d}}{\mathrm{d}\alpha_j}(M_\alpha(x_1) - y_1) \\
&= 2(M_\alpha(x_1) - y_1)\frac{\mathrm{d}}{\mathrm{d}\alpha_j}(\sum_{j=0}^{2} \alpha_j x_1^j - y_1) \\
&= 2(M_\alpha(x_1) - y_1)x_1^j
\end{aligned}$$

Using this convenient form (which generalizes to any polynomial model), we define

$$\begin{aligned}
\nabla f(\alpha) &= \left(\frac{\mathrm{d}}{\mathrm{d}\alpha_0} f(\alpha), \frac{\mathrm{d}}{\mathrm{d}\alpha_1} f(\alpha), \frac{\mathrm{d}}{\mathrm{d}\alpha_2} f(\alpha)\right) \\
&= 2\left((M_\alpha(x_1) - y_1),\ \ (M_\alpha(x_1) - y_1)x_1,\ \ (M_\alpha(x_1) - y_1)x_1^2\right).
\end{aligned}$$

Applying $\alpha := \alpha - \gamma \nabla f(\alpha)$ according to this specification is known as the *LMS (least mean squares)* update rule or the *Widrow-Huff learning rule*. Quite intuitively, the magnitude the update is proportional to the residual norm $(M_\alpha(x_1) - y_1)$. So if we have a lot of error in our guess of α, then we take a large step; if we do not have a lot of error, we take a small step.

6.4.2 Decomposable Functions

To generalize this to multiple data points ($n > 1$), there are two standard ways. Both of these take strong advantage of the cost function $f(\alpha)$ being *decomposable*. That is, we can write

$$f(\alpha) = \sum_{i=1}^{n} f_i(\alpha),$$

where each f_i depends only on the ith data point $p_i \in P$. In particular, where $p_i = (x_i, y_i)$, then for quadratic regression

$$f_i(\alpha) = (M_\alpha(x_i) - y_i)^2 = (\alpha_0 + \alpha_1 x_i + \alpha_2 x_i^2 - y_i)^2.$$

First notice that since f is the sum of f_is, where each is convex, then f must also be convex; in fact the sum of these usually becomes strongly convex (as long as the corresponding feature vectors are full rank). Also two approaches towards gradient descent will take advantage of this decomposition in slightly different ways. This decomposable property holds for most loss functions for fitting a model to data.

Batch gradient descent. The first technique, called *batch gradient descent*, simply extends the definition of $\nabla f(\alpha)$ to the case with multiple data points. Since f is decomposable, then we use the linearity of the derivative to define

$$\frac{\mathrm{d}}{\mathrm{d}\alpha_j} f(\alpha) = \sum_{i=1}^{n} \frac{\mathrm{d}}{\mathrm{d}\alpha_j} f_i(\alpha) = \sum_{i=1}^{n} 2(M_\alpha(x_i) - y_i)x_i^j$$

and thus

$$\begin{aligned}
\nabla f(\alpha) &= \left(2\sum_{i=1}^{n}(M_\alpha(x_i) - y_i),\ \ 2\sum_{i=1}^{n}(M_\alpha(x_i) - y_i)x_i,\ \ 2\sum_{i=1}^{n}(M_\alpha(x_i) - y_i)x_i^2 \right) \\
&= \sum_{i=1}^{n} \left(2(M_\alpha(x_i) - y_i),\ \ 2(M_\alpha(x_i) - y_i)x_i,\ \ 2(M_\alpha(x_i) - y_i)x_i^2 \right) \\
&= 2\sum_{i=1}^{n}(M_\alpha(x_i) - y_i)\left(1, x_i, x_i^2\right).
\end{aligned}$$

That is, the step is now just the sum of the terms from each data point. Since f is (strongly) convex, then we can apply all of the nice convergence results discussed about (strongly) convex f before. However, computing $\nabla f(\alpha)$ each step takes $O(n)$ time, which can be slow for large n (i.e., for large data sets).

Algorithm 6.4.1 Incremental Gradient Descent$(f, \alpha_{\mathsf{start}})$

initialize $\alpha^{(0)} = \alpha_{\mathsf{start}} \in \mathbb{R}^d$; $i = 1$.
repeat
  $\alpha^{(k+1)} := \alpha^{(k)} - \gamma_k \nabla f_i(\alpha^{(k)})$
  $i := (i + 1) \mod n$
until $(\|\nabla f(\alpha^{(k)})\| \leq \tau)$ $\quad (\star)$
return $\alpha^{(k)}$

Stochastic gradient descent. The second technique is called *incremental gradient descent*; see Algorithm 6.4.1. It avoids computing the full gradient each step, and only computes $\nabla f_i(\alpha)$ for a single data point $p_i \in P$. For quadratic regression with $M_\alpha(x) = \langle \alpha, (1, x, x^2) \rangle$ it is

$$\nabla f_i(\alpha) = 2(M_\alpha(x_i) - y_i)(1, x_i, x_i^2).$$

Implementation Hints:
($\star$) The norm of a single gradient is not stable. Instead a better stopping condition averages the gradient norm over several (lets say B steps). The condition may then be $\frac{1}{B}\sum_{b=0}^{B-1} \|\nabla f(\alpha^{(k-b)})\| \leq \tau$, and is only checked after B steps are taken.

A more common variant of this is called *stochastic gradient descent*; see Algorithm 6.4.2. Instead of choosing the data points in order, it selects a data point p_i at random each iteration (the term "stochastic" refers to this randomness).

Algorithm 6.4.2 Stochastic Gradient Descent$(f, \alpha_{\text{start}})$

initialize $\alpha^{(0)} = \alpha_{\text{start}} \in \mathbb{R}^d$
repeat
  Randomly choose $i \in \{1, 2, \ldots, n\}$
  $\alpha^{(k+1)} := \alpha^{(k)} - \gamma_k \nabla f_i(\alpha^{(k)})$
until $(\|\nabla f(\alpha^{(k)})\| \leq \tau)$ ($\star$)
return $\alpha^{(k)}$

On very large data sets (i.e., big data!), these algorithms are often much faster than the batch version since each iteration now takes $O(1)$ time. However, it does not automatically inherit all of the nice convergence results from what is known about (strongly) convex functions. Yet in many settings, there is an abundance of data points described by the same model. They should have a roughly similar effect. In practice when one is far from the optimal model, these steps converge about as well as the batch version (but much much faster in runtime). When one is close to the optimal model, then the incremental / stochastic variants may not exactly converge. However, if one is satisfied to reach a point that is close enough to optimal, there are some randomized (PAC-style) guarantees possible for the stochastic variant. And in fact, for very large data sets (i.e., n is very big) they typically converge before the algorithm even uses all (or even most) of the data points.

Exercises

We will use a dataset `http://www.cs.utah.edu/~jeffp/teaching/FoDA/D4.csv`

Q6.1: Consider a function $f(x, y)$ with gradient $\nabla f(x, y) = (x - 1, 2y + x)$. Starting at a value $(x = 1, y = 2)$, and a learning rate of $\gamma = 1$, execute one step of gradient descent.

Q6.2: Consider running gradient descent with a fixed learning rate γ. For each of the following, we plot the function value over 10 steps (the function is different each time). Decide whether the learning rate is probably **too high**, **too low**, or **about right**.

1. f_1: 100, 99, 98, 97, 96, 95, 94, 93, 92, 91
2. f_2: 100, 50, 75, 60, 65, 45, 75, 110, 90, 85
3. f_3: 100, 80, 65, 50, 40, 35, 31, 29, 28, 27.5, 27.3
4. f_4: 100, 80, 60, 40, 20, 0, -20, -40, -60, -80, -100

Q6.3: Consider two functions

$$f_1(x, y) = (x - 5)^2 + (y + 2)^2 - 2xy \qquad f_2(x, y) = (1 - (y - 4))^2 + 20((x + 6) - (y - 4)^2)^2$$

Starting with $(x, y) = (0, 2)$ run the gradient descent algorithm for each function. Run for T iterations, and report the function value at the end of each step.

1. First, run with a fixed learning rate of $\gamma = 0.05$ for f_1 and $\gamma = 0.0015$ for f_2.
2. Second, run with any variant of gradient descent you want. Try to get the smallest function value after T steps.

For f_1 you are allowed only $T = 10$ steps. For f_2 you are allowed $T = 100$ steps.

Q6.4: In the first `D4.csv` dataset provided, use the first three columns as explanatory variables x_1, x_2, x_3, and the fourth as the dependent variable y. Run gradient descent on $\alpha \in \mathbb{R}^4$, using the dataset provided to find a linear model

$$\hat{y} = \alpha_0 + \alpha_1 x_1 + \alpha_2 x_2 + \alpha_3 x_3$$

minimizing the sum of squared errors. Run for as many steps as you feel necessary. On each step of your run, print on a single line: (1) the model parameters $\alpha^{(i)} = [\alpha_0^{(i)}, \alpha_1^{(i)}, \alpha_2^{(i)}, \alpha_3^{(i)}]$, (2) the value of a function $f(\alpha^{(i)})$, estimating the sum of squared errors, and (3) the gradient $\nabla f(\alpha^{(i)})$. *(These are the sort of things you would do to check/debug a gradient descent algorithm; you may also want to plot some of these.)*

1. First run batch gradient descent.
2. Second run incremental gradient descent.

Choose one method which you preferred (either is ok to choose), and explain why you preferred it to the other method.

Q6.5: Explain what parts of the above procedures would change if you instead are minimizing the sum of residuals, not the sum of squared residuals?

- Is the function still convex?
- Does the gradient always exist?

7 Principal Component Analysis

This topic will build a series of techniques to deal with high-dimensional data. Unlike regression problems, our goal is not to predict a value (the y-coordinate), it is to understand the "shape" of the data, for instance a low-dimensional representation that captures most of meaning of the high-dimensional data. This is sometimes referred to as *unsupervised learning* (as opposed to regression and classification, where the data has labels, known as supervised learning). Like most unsupervised settings, it can be a lot of fun, but its easy to get yourself into trouble if you are not careful.

We will cover many interconnected tools, including the singular value decomposition (SVD), eigenvectors and eigenvalues, the power method, principal component analysis, and multidimensional scaling.

7.1 Data Matrices

We will start with data in a matrix $A \in \mathbb{R}^{n \times d}$, and will call upon linear algebra to rescue us. It is useful to think of each row a_i of A as a data point in $\mathbb{R}^d$, so there are n data points. Each dimension $j \in 1, 2, \ldots, d$ corresponds with an attribute of the data points.

Example: Data Matrices

There are many situations where data matrices arise.

- Consider a set of n weather stations reporting temperature over d points in time. Then each row a_i corresponds to a single weather station, and each coordinate $A_{i,j}$ of that row is the temperature at station i at time j.
- In movie ratings, we may consider n users who have rated each of d movies on a score of $1-5$. Then each row a_i represents a user, and the jth entry of that user is the score given to the j movie.
- Consider the price of a stock measured over time (say the closing price each day). Many time-series models consider some number of days (d days, for instance 25 days) to capture the pattern of the stock at any given time. So for a given closing day, we consider the d previous days. If we have data on the stock for 4 years (about 1000 days the stock market is open), then we can create a d-dimensional data points (the previous $d = 25$ days) for each day (except the first 25 or so). The data matrix is then comprised of n data points a_i, where each corresponds to the closing day, and the previous d days. The jth entry is the value on $(j-1)$ days before the closing day i.
- Finally consider a series of pictures of a shape (say the Utah teapot). The camera position is fixed as is the background, but we vary two things: the rotation of the teapot, and the amount of light. Here each pictures is a set of say d pixels (say 10,000 if it is 100×100), and there are n pictures. Each picture is a row of length d, and each pixel corresponds to a column of the matrix. Similar, but more complicated scenarios frequently occur with pictures of a persons face, or $3d$-imaging of an organ.

In each of these scenarios, there are many (n) data points, each with d attributes. The following will be very important:

- **all coordinates have the same units!**

If this "same units" property does not hold, then when we measure a distance between data points in $\mathbb{R}^d$, usually using the L_2-norm, then the distance is nonsensical.

The next goal is to uncover a pattern, or a model M. In this case, the model will be a low-dimensional subspace F. It will represent a k-dimensional space, where $k << d$. For instance in the example with images, there are only two parameters which are changing (rotation, and lighting), so despite having $d = 10{,}000$ dimensions of data, 2 should be enough to represent everything.

7.1.1 Projections

Different than in linear regression this family of techniques will measure error as a projection from $a_i \in \mathbb{R}^d$ to the closest point $\pi_F(a_i)$ on F. To define this we will use linear algebra.

First recall, that given a unit vector $u \in \mathbb{R}^d$ and any data point $p \in \mathbb{R}^d$, then the dot product

$$\langle u, p \rangle$$

is the norm of p projected onto the line through u. If we multiply this scalar by u then

$$\pi_u(p) = \langle u, p \rangle u,$$

and it results in the point on the line through u that is closest to data point p. This is a *projection of p onto u*.

To understand this for a subspace F, we will need to define a basis. *For now we will assume that F contains the origin $(0, 0, 0, \ldots, 0)$ (as did the line through u).* Then if F is k-dimensional, then this means there is a k-dimensional basis $U_F = \{u_1, u_2, \ldots, u_k\}$ so that

- For each $u_i \in U_F$ we have $\|u_i\| = 1$, that is u_i is a unit vector.
- For each pair $u_i, u_j \in U_F$ we have $\langle u_i, u_j \rangle = 0$; the pairs are orthogonal.
- For any point $x \in F$ we can write $x = \sum_{i=1}^{k} \alpha_i u_i$; in particular $\alpha_i = \langle x, u_i \rangle$.

Given such a basis, then the projection on to F of a point $p \in \mathbb{R}^d$ is simply

$$\pi_F(p) = \sum_{i=1}^{k} \langle u_i, p \rangle u_i.$$

Thus if p happens to be exactly in F, then this recovers p exactly.

The other powerful part of the basis U_F is the it defines a *new coordinate system*. Instead of using the d original coordinates, we can use new coordinates $(\alpha_1(p), \alpha_2(p), \ldots, \alpha_k(p))$ where $\alpha_i(p) = \langle u_i, p \rangle$. To be clear $\pi_F(p)$ is still in $\mathbb{R}^d$, but there is a k-dimensional representation if we restrict to F.

When F is d-dimensional, this operation can still be interesting. The basis we choose $U_F = \{u_1, u_2, \ldots, u_d\}$ could be the same as the original coordinate axis, that is we could have $u_i = e_i = (0, 0, \ldots, 0, 1, 0, \ldots, 0)$ where only the ith coordinate is 1. But if it is another basis, then this acts as a rotation (with possibility of also a mirror flip). The first coordinate is rotated to be along u_1; the second along u_2; and so on. In $\pi_F(p)$, the point p does not change, just its representation.

7.1.2 SSE Goal

As usual our goal will be to minimize the sum of squared errors. In this case we define this as

$$\mathsf{SSE}(A, F) = \sum_{a_i \in A} \|a_i - \pi_F(a_i)\|^2,$$

and our desired k-dimensional subspace F is

$$F^* = \arg\min_F \mathsf{SSE}(A, F)$$

As compared to linear regression, this is much less a "proxy goal" where the true goal was prediction. Now we have no labels (the y_i values), so we simply try to fit a model through all of the data.

How do we solve for this?

- Linear regression does not work, its cost function is different.
- It is not obvious how to use gradient descent. The restriction that each $u_i \in U_F$ is a unit vector puts in a constraint, in fact a non-convex one. There are ways to deal with this, but we have not discussed these yet.
- ... linear algebra will come back to the rescue, now in the form of the SVD.

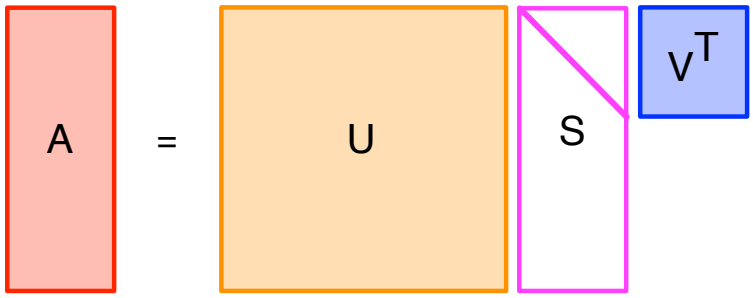
7.2 Singular Value Decomposition

A really powerful and useful linear algebra operation is called the *singular value decomposition*. It extracts an enormous amount of information about a matrix A. This section will define it and discuss many of its uses. Then we will describe one algorithm how to construct it. But in general, one simply calls the procedure in your favorite programming language and it calls the same highly-optimized back-end from the Fortran LAPACK library.

```
from scipy import linalg as LA
U, s, Vt = LA.svd(A)
```

The SVD takes in a matrix $A \in \mathbb{R}^{n \times d}$ and outputs three matrices $U \in \mathbb{R}^{n \times n}$, $S \in \mathbb{R}^{n \times d}$ and $V \in \mathbb{R}^{d \times d}$, so that $A = USV^T$.

$$[U, S, V] = \mathsf{svd}(A)$$



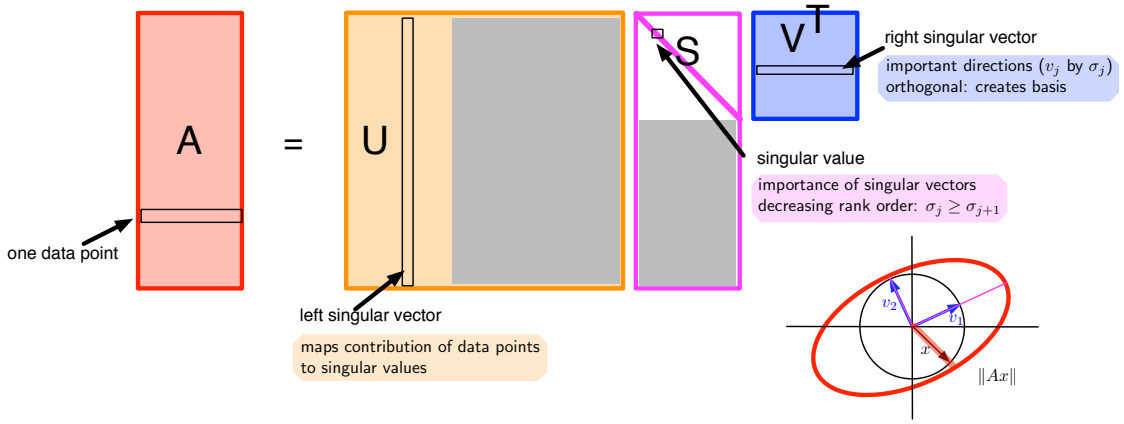
The structure that lurks beneath. The matrix S only has non-zero elements along its diagonal. So $S_{i,j} = 0$ if $i \neq j$. The remaining values $\sigma_1 = S_{1,1}$, $\sigma_2 = S_{2,2}$, ..., $\sigma_r = S_{r,r}$ are known as the *singular values*. They have the property that

$$\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_r \geq 0$$

where $r \leq \min\{n, d\}$ is the rank of the matrix A. So the number of non-zero singular values reports the rank (this is a numerical way of computing the rank or a matrix).

The matrices U and V are orthogonal. Thus, their columns are all unit vectors and orthogonal to each other (within each matrix). The columns of U, written $u_1, u_2, \ldots, u_n$, are called the *left singular vectors*; and the columns of V (i.e., rows of V^T), written $v_1, v_2, \ldots, v_d$, are called the *right singular vectors*.

This means for any vector $x \in \mathbb{R}^d$, the columns of V (the right singular vectors) provide a basis. That is, we can write $x = \sum_{i=1}^{d} \alpha_i v_i$ for $\alpha_i = \langle x, v_i \rangle$. Similarly for any vector $y \in \mathbb{R}^n$, the columns of U (the left singular vectors) provide a basis. This also implies that $\|x\| = \|V^T x\|$ and $\|y\| = \|yU\|$.



Tracing the path of a vector. To illustrate what this decomposition demonstrates, a useful exercise is to trace what happens to a vector $x \in \mathbb{R}^d$ as it is left-multiplied by A, that is $Ax = USV^T x$.

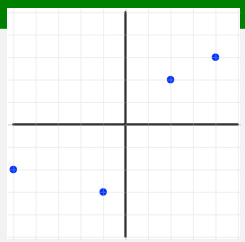
First $V^T x$ produces a new vector $\xi \in \mathbb{R}^d$. It essentially changes no information, just changes the basis to that described by the right singular values. For instance the new i coordinate $\xi_i = \langle v_i, x \rangle$.

Next $\eta \in \mathbb{R}^n$ is the result of $SV^T x = S\xi$. It scales ξ by the singular values of S. Note that if $d < n$ (the case we will focus on), then the last $n - d$ coordinates of η are 0. In fact, for $j > r$ (where $r = \mathsf{rank}(A)$) then $\eta_j = 0$. For $j \leq r$, then the vector η is stretched longer in the first coordinates since these have larger values.

The final result is a vector $y \in \mathbb{R}^n$, the result of $Ax = USV^T x = U\eta$. This again just changes the basis of η so that it aligns with the left singular vectors. In the setting $n > d$, the last $n - d$ left singular vectors are meaningless since the corresponding entries in η are 0.

Working backwards ... this final U matrix can be thought of mapping the effect of η onto each of the data points of A. The η vector, in turn, can be thought of as scaling by the content of the data matrix A (the U and V^T matrices contain no scaling information). And the ξ vector arises via the special rotation matrix V^T that puts the starting point x into the right basis to do the scaling (from the original d-dimensional coordinates to one that suits the data better).

Example: Tracing through the SVD



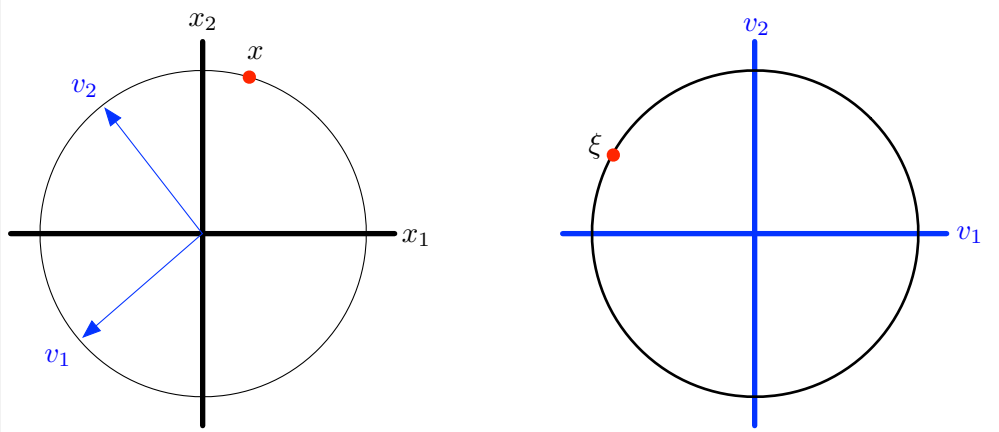
Consider a matrix

$$A = \begin{pmatrix} 4 & 3 \\ 2 & 2 \\ -1 & -3 \\ -5 & -2 \end{pmatrix},$$

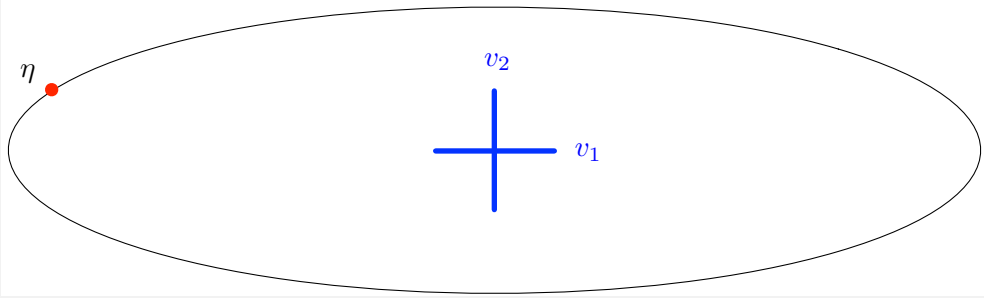
and its SVD $[U, S, V] = \mathsf{svd}(A)$:

$$U = \begin{pmatrix} -0.6122 & 0.0523 & 0.0642 & 0.7864 \\ -0.3415 & 0.2026 & 0.8489 & -0.3487 \\ 0.3130 & -0.8070 & 0.4264 & 0.2625 \\ 0.6408 & 0.5522 & 0.3057 & 0.4371 \end{pmatrix}, \quad S = \begin{pmatrix} 8.1655 & 0 \\ 0 & 2.3074 \\ 0 & 0 \\ 0 & 0 \end{pmatrix}, \quad V = \begin{pmatrix} -0.8142 & -.5805 \\ -0.5805 & 0.8142 \end{pmatrix}.$$

Now consider a vector $x = (0.243, 0.97)$ (scaled very slightly so it is a unit vector, $\|x\| = 1$). Multiplying by V^T rotates (and flips) x to $\xi = V^T x$; still $\|\xi\| = 1$



Next multiplying by S scales ξ to $\eta = S\xi$. Notice there are an imaginary third and fourth coordinates now; they are both coming out of the page! Don't worry, they won't poke you since their magnitude is 0.



Finally, $y = U\eta = Ax$ is again another rotation of η in this four dimensional space.

```
import scipy as sp
import numpy as np
from scipy import linalg as LA


A = np.array([[4.0,3.0], [2.0,2.0], [-1.0,-3.0], [-5.0,-2.0]])

U, s, Vt = LA.svd(A, full_matrices=False)

print U
#[[-0.61215255 -0.05228813]
# [-0.34162337 -0.2025832 ]
# [ 0.31300005  0.80704816]
# [ 0.64077586 -0.55217683]]
print s
#[ 8.16552039  2.30743942]
print Vt
#[[-0.81424526 -0.58052102]
# [ 0.58052102 -0.81424526]]

x = np.array([0.243,0.97])
x = x/LA.norm(x)

xi = Vt.dot(x)
print xi
#[-0.7609864  -0.64876784]

S = LA.diagsvd(s,2,2)
eta = S.dot(xi)
print eta
#[-6.21384993 -1.49699248]

y = U.dot(eta)
print y
#[ 3.88209899  2.42606187 -3.1530804  -3.15508046]

print A.dot(x)
#[ 3.88209899  2.42606187 -3.1530804  -3.15508046]
```

7.2.1 Best Rank-k Approximation

So how does this help solve the initial problem of finding F^*, which minimized the SSE? The singular values hold the key.

It turns out that there is a *unique* singular value decomposition, up to ties in the singular values. This means, there is exactly one (up to singular value ties) set of right singular values which rotate into a basis so that $\|Ax\| = \|SV^T x\|$ for all $x \in \mathbb{R}^d$ (recall that U is orthogonal, so it does not change the norm, $\|U\eta\| = \|\eta\|$).

Next we realize that the singular values come in sorted order $\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_r$. In fact, they are defined so that we choose v_1 so it maximizes $\|Av_1\|$, then we find the next singular vector v_2 which is orthogonal to v_1 and maximizes $\|Av_2\|$, and so on. Then $\sigma_i = \|Av_i\|$.

If we define F with the basis $U_F = \{v_1, v_2, \ldots, v_k\}$, then

$$\|x - \pi_F(x)\|^2 = \left\| \sum_{i=1}^{d} v_i \langle x, v_i \rangle - \sum_{i=1}^{k} v_i \langle x, v_i \rangle \right\|^2 = \sum_{i=k+1}^{d} \langle x, v_i \rangle^2.$$

so the projection error is that part of x in the last $(d-k)$ right singular vectors.

But we are not trying to directly predict new data here (like in regression). Rather, we are trying to approximate the data we have. We want to minimize $\sum_i \|a_i - \pi_F(a_i)\|^2$. But for any unit vector u, we recall now that

$$\|Au\|^2 = \sum_{i=1}^{n} \langle a_i, u \rangle.$$

Thus the projection error can be measured with a set of orthonormal vectors $w_1, w_2, \ldots, w_{d-k}$ which are each orthogonal to F, as $\sum_{j=1}^{n-k} \|Aw_j\|^2$. When defining F as the first k right singular values, then these orthogonal vectors are the remaining $(n-k)$ right singular vectors, so the projection error is

$$\sum_{i=1}^{n} \|a_i - \pi_F(a_i)\|^2 = \sum_{j=k+1}^{d} \|Av_j\|^2 = \sum_{j=k+1}^{d} \sigma_j^2.$$

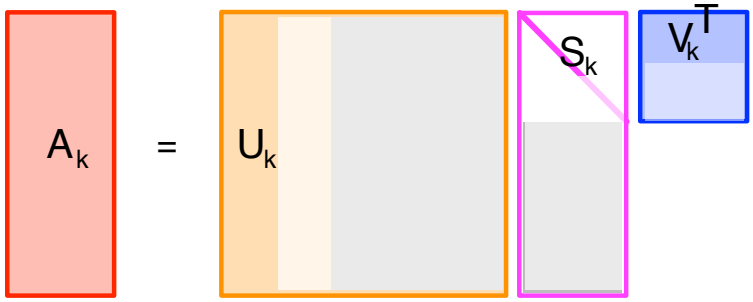
And thus by how the right singular vectors are defined, this expression is minimized when F is defined as the span of the first k singular values.

Best rank-k approximation. A similar goal is to find the *best rank-k approximation of A*. That is a matrix $A_k \in \mathbb{R}^{n \times d}$ so that $\mathsf{rank}(A_k) = k$ and it minimizes both

$$\|A - A_k\|_2 \quad \text{and} \quad \|A - A_k\|_F.$$

Note that $\|A - A_k\|_2 = \sigma_{k+1}$ and $\|A - A_k\|_F^2 = \sum_{j=k+1}^{d} \sigma_j^2$.

Remarkably, this A_k matrix also comes from the SVD. If we set S_k as the matrix S in the decomposition so that all but the first k singular values are 0, then it has rank k. Hence $A_k = US_kV^T$ also has rank k and is our solution. But we can notice that when we set most of S_k to 0, then the last $(d-k)$ columns of V are meaningless since they are only multiplied by 0s in US_kV^T, so we can also set those to all 0s, or remove them entirely (along with the last $(d-k)$ columns of S_k). Similar we can make 0 or remove the last $(n-k)$ columns of U. These matrices are referred to as V_k and U_k respectively, and also $A_k = U_kS_kV_k^T$.



7.3 Eigenvalues and Eigenvectors

A related matrix decomposition to SVD is the eigendecomposition. This is only defined for a square matrix $B \in \mathbb{R}^{n \times n}$.

An *eigenvector* of B is a vector v such that there is some scalar λ that

$$Bv = \lambda v.$$

That is, multiplying B by v results in a scaled version of v. The associated value λ is called the *eigenvalue*. As a convention, we typically normalize v so $\|v\| = 1$.

In general, a square matrix $B \in \mathbb{R}^{n \times n}$ may have up to n eigenvectors (a matrix $\mathtt{V} \in \mathbb{R}^{n \times n}$) and values (a vector $\mathtt{l} \in \mathbb{R}^n$). Some of the eigenvalues may be complex numbers (even when all of its entries are real!).


```
from scipy import linalg as LA
l, V = LA.eig(B)
```

For this reason, we will focus on positive semidefinite matrices. A *positive definite matrix* $B \in \mathbb{R}^{n \times n}$ is a symmetric matrix with all positive eigenvalues. Another characterization is for every vector $x \in \mathbb{R}^n$ then $x^T B x$ is positive. A *positive semidefinite matrix* $B \in \mathbb{R}^{n \times n}$ may have some eigenvalues at 0 and are otherwise positive; equivalently for any vector $x \in \mathbb{R}^n$, then $x^T B x$ may be zero or positive.

How do we get positive semi-definite matrices? Lets start with a data matrix $A \in \mathbb{R}^{n \times d}$. Then we can construct two positive semidefinite matrices

$$B_R = A^T A \qquad \text{and} \qquad B_L = A A^T.$$

Matrix B_R is $d \times d$ and B_L is $n \times n$. If the rank of A is d, then B_R is positive definite. If the rank of A is n, then B_L is positive definite.

Eigenvectors and eigenvalues relation to SVD. Next consider the SVD of A so that $[U, S, V] = \mathsf{svd}(A)$. Then we can write

$$B_R V = A^T A V = (V S U^T)(U S V^T) V = V S^2.$$

Note that the last step follows because for orthogonal matrices U and V, then $U^T U = I$ and $V^T V = I$, where I is the identity matrix, which has no effect. The matrix S is a diagonal square[1] matrix $S = \mathsf{diag}(\sigma_1, \sigma_2, \ldots, \sigma_d)$. Then $S^2 = SS$ (the product of S with S) is again diagonal with entries $S^2 = \mathsf{diag}(\sigma_1^2, \sigma_2^2, \ldots, \sigma_d^2)$.

Now consider a single column v_i of V (which is the ith right singular vector of A). Then extracting this column's role in the linear system $B_R V = V S^2$ we obtain

$$B_R v_i = v_i \sigma_i^2.$$

This means that ith right singular vector of A is an eigenvector (in fact the ith eigenvector) of $B_R = A^T A$. Moreover, the ith eigenvalue λ_i of B_R is the ith singular value of A squared: $\lambda_i = \sigma_i^2$.

Similarly we can derive

$$B_L U = A A^T U = (U S V^T)(V S U^T) U = U S^2,$$

and hence the left singular vectors of A are the eigenvectors of $B_L = A A^T$ and the eigenvalues of B_L are the squared singular values of A.

Eigendecomposition. In general, the eigenvectors provide a basis for a matrix $B \in \mathbb{R}^{n \times n}$ in the same way that the right V or left singular vectors U provide a basis for matrix $A \in \mathbb{R}^{n \times d}$. In fact, it is again a very special basis, and is unique up to the multiplicity of eigenvalues. This implies that all eigenvectors are orthogonal to each other.

Let $V = [v_1, v_2, \ldots, v_n]$ be the eigenvectors of the matrix $B \in \mathbb{R}^{n \times n}$, as columns in the matrix V. Also let $L = \mathsf{diag}(\lambda_1, \lambda_2, \ldots, \lambda_d)$ be the eigenvalues of B stored on the diagonal of matrix L. Then we can decompose B as

$$B = V L V^{-1}.$$

[1] Technically, $S \in \mathbb{R}^{n \times d}$. To make this simple argument work, lets first assume w.l.o.g. (without loss of generality) that $d \leq n$. Then the bottom $n - d$ rows of S are all zeros, which mean the right $n - d$ rows of U do not matter. So we can ignore both these $n - d$ rows and columns. Then S is square. This makes U no longer orthogonal, so $U^T U$ is then a projection, not identity; but it turns out this is a project to the span of A, so the argument still works.

Note that the inverse of L is $L^{-1} = \mathsf{diag}(1/\lambda_1, 1/\lambda_2, \ldots, 1/\lambda_n)$. Hence we can write

$$B^{-1} = VL^{-1}V^{-1}.$$

When B is positive definite, it has n positive eigenvectors and eigenvalues; hence V is orthogonal, so $V^{-1} = V^T$. Thus in this situation, given the eigendecomposition, we now have a way to compute the inverse

$$B^{-1} = VL^{-1}V^T,$$

which was required in our almost closed-form solution for linear regression. Now we just need to compute the eigendecomposition, which we will discuss next.

7.4 The Power Method

The *power method* refers to what is probably the simplest algorithm to compute the first eigenvector and value of a matrix. By factoring out the effect of the first eigenvector, we can then recursively repeat the process on the remainder until we have found all eigenvectors and values. Moreover, this implies we can also reconstruct the singular value decomposition as well.

We will consider $B \in \mathbb{R}^{n \times n}$, a positive semidefinite matrix: $B = A^T A$.

Algorithm 7.4.1 PowerMethod(B, q)

initialize $u^{(0)}$ as a random unit vector.
for $i = 1$ **to** q **do**
  $u^{(i)} := Bu^{(i-1)}$
return $v := u^{(q)}/\|u^{(q)}\|$

We can unroll the for loop to reveal another interpretation. We can directly set $v^{(q)} = B^q v^{(0)}$, so all iterations are incorporated into one matrix-vector multiplication. Recall that $B^q = B \cdot B \cdot B \cdot \ldots \cdot B$, for q times. However, these q matrix multiplications are much more expensive then q matrix-vector multiplications.

Alternatively we are provided only the matrix A (where $B = A^T A$) then we can run the algorithm without explicitly constructing B (since for instance if $d > n$ and $A \in \mathbb{R}^{n \times d}$, then the size of B (d^2) may be much larger than A (nd)). Then we simply replace the inside of the for-loop with

$$u^{(i)} := A^T(Au^{(i-1)})$$

where we first multiply $\tilde{u} = Au^{(i-1)}$ and then complete $u^{(i)} = A^T\tilde{u}$.

Recovering all eigenvalues. The output of PowerMethod$(B = A^T A, q)$ is a single unit vector v, which we will argue is arbitrarily close to the first eigenvector v_1. Clearly we can recover the first eigenvalue as $\lambda_1 = \|Bv_1\|$. Since we know the eigenvectors form a basis for B, they are orthogonal. Hence, after we have constructed the first eigenvector v_1, we can factor it out from B as follows:

$$A_1 := A - Av_1v_1^T$$
$$B_1 := A_1^T A_1$$

Then we run PowerMethod$(B_1 = A_1^T A_1, q)$ to recover v_2, and λ_2; factor them out of B_1 to obtain B_2, and iterate.

Geometry of Why the Power Method Works?

To understand why the power method works, assume we know the eigenvectors $v_1, v_2, \ldots, v_n$ and eigenvalues $\lambda_1, \lambda_2, \ldots, \lambda_n$ of $B \in \mathbb{R}^{n \times n}$.
Since the eigenvectors form a basis for B, and assuming it is full rank, then also for all of $\mathbb{R}^n$ (if not, then it does not have n eigenvalues, and we can fill out the rest of the basis of $\mathbb{R}^n$ arbitrarily). Hence, for any vector, including the initialization random vector $u^{(0)}$ can be written as

$$u^{(0)} = \sum_{j=1}^{n} \alpha_j v_j.$$

Recall that $\alpha_j = \langle u^{(0)}, v_j \rangle$, and since it is random, it is possible to claim that with probability at least $1/2$ that for any α_j we have that $|\alpha_j| \geq \frac{1}{2}\sqrt{n}$[a]. We will now assume that this holds for $j = 1$, so $\alpha_1 > 1/2\sqrt{n}$.
Next since we can interpret that algorithm as $v = B^q u^{(0)}$, then lets analyze B^q. If B has jth eigenvector v_j and eigenvalue λ_j, that is, $Bv_j = \lambda_j v_j$, then B^q has jth eigenvalue λ_j^q since

$$B^q v_j = B \cdot B \cdot \ldots \cdot B v_j = B^{q-1}(v_j \lambda_j) = B^{q-2}(v_j \lambda_j)\lambda_j = v_j \lambda_j^q.$$

This holds for each eigenvalue of B^q. Hence we can rewrite output by summing over the terms in the eigenbasis as

$$v = \frac{\sum_{j=1}^{n} \alpha_j \lambda_j^q v_j}{\sqrt{\sum_{j=1}^{n} (\alpha_j \lambda_j^q)^2}}.$$

Finally, we would like to show our output v is close to the first eigenvector v_1. We can measure closeness with the dot product (actually we will need to use its absolute value since we might find something close to $-v_1$).

$$\begin{aligned}
|\langle B^q u^{(0)}, v_1 \rangle| &= \frac{\alpha_1 \lambda_1^q}{\sqrt{\sum_{j=1}^{n} (\alpha_j \lambda_j^q)^2}} \\
&\geq \frac{\alpha_1 \lambda_1^q}{\sqrt{\alpha_1^2 \lambda_1^{2q} + n\lambda_2^{2t}}} \geq \frac{\alpha_1 \lambda_1^q}{\alpha_1 \lambda_1^q + \lambda_2^q \sqrt{n}} = 1 - \frac{\lambda_2^q \sqrt{n}}{\alpha_1 \lambda_1^q + \lambda_2^q \sqrt{n}} \\
&\geq 1 - 2\sqrt{n} \left(\frac{\lambda_2}{\lambda_1}\right)^q.
\end{aligned}$$

The first inequality holds because $\lambda_1 \geq \lambda_2 \geq \lambda_j$ for all $j > 2$. The third inequality (going to third line) holds by dropping the $\lambda_2^q \sqrt{n}$ term in the denominator, and since $\alpha_1 > 1/2\sqrt{n}$.
Thus if there is "gap" between the first two eigenvalues (λ_1/λ_2 is large), then this algorithm converges quickly to where $|\langle v, v_1 \rangle| = 1$.

[a] Since $u^{(0)}$ is a unit vector, its norm is 1, and because $\{v_1, \ldots, v_n\}$ is a basis, then $1 = \|u^{(0)}\|^2 = \sum_{j=1}^{n} \alpha_j^2$. Since it is random, then $\mathbf{E}[\alpha_j^2] = 1/n$ for each j. Applying a concentration of measure (almost a Markov Inequality, but need to be more careful), we can argue that with probability $1/2$ any $\alpha_j^2 > (1/4) \cdot (1/n)$, and hence $\alpha_j > (1/2) \cdot (1/\sqrt{n})$.

7.5 Principal Component Analysis

Recall that the original goal of this topic was to find the k-dimensional subspace F to minimize

$$\|A - \pi_F(A)\|_F^2 = \sum_{a_i \in A} \|a_i - \pi_F(a_i)\|^2.$$

We have not actually solved this yet. The top k right singular values V_k of A only provided this bound assuming that F contains the origin: $(0, 0, \ldots, 0)$. However, this might not be the case!

Principal Component Analysis (PCA) is an extension of the SVD when we do not restrict that the subspace V_k must go through the origin. It turns out, like with simple linear regression, that the optimal F must go through the mean of all of the data. So we can still use the SVD, after a simple preprocessing step called centering to shift the data matrix so its mean is exactly at the origin.

Specifically, *centering* is adjusting the original input data matrix $A \in \mathbb{R}^{n \times d}$ so that each column (each dimension) has an average value of 0. This is easier than it seems. Define $\bar{a}_j = \frac{1}{n}\sum_{i=1}^n A_{i,j}$ (the average of each column j). Then set each $\tilde{A}_{i,j} = A_{i,j} - \bar{a}_j$ to represent the entry in the ith row and jth column of centered matrix $\tilde{A}$.

There is a *centering matrix* $C_n = I_n - \frac{1}{n}\mathbf{1}\mathbf{1}^T$ where I_n is the $n \times n$ identity matrix, $\mathbf{1}$ is the all-ones column vector (of length n) and thus $\mathbf{1}\mathbf{1}^T$ is the all-ones $n \times n$ matrix. Then we can also just write $\tilde{A} = C_n A$.

Now to perform PCA on a data set A, we compute $[U, S, V] = \texttt{svd}(C_n A) = \texttt{svd}(\tilde{A})$.

Then the resulting singular values $\texttt{diag}(S) = \{\sigma_1, \sigma_2, \ldots, \sigma_r\}$ are known as the *principal values*, and the top k right singular vectors $V_k = [v_1\ v_2\ \ldots\ v_k]$ are known as the top-k *principal directions*.

This often gives a better fitting to the data than just SVD. The SVD finds the best rank-k approximation of A, which is the best k-dimensional subspace (up to Frobenius and spectral norms) **which passes through the origin**. If all of the data is far from the origin, this can essentially "waste" a dimension to pass through the origin. However, we also need to store the shift from the origin, a vector $\tilde{c} = (\tilde{a}_1, \tilde{a}_2, \ldots, \tilde{a}_d) \in \mathbb{R}^d$.

7.6 Multidimensional Scaling

Dimensionality reduction is an abstract problem with input of a high-dimensional data set $P \subset \mathbb{R}^d$ and a goal of finding a corresponding lower dimensional data set $Q \subset \mathbb{R}^k$, where $k << d$, and properties of P are preserved in Q. Both low-rank approximations through direct SVD and through PCA are examples of this: $Q = \pi_{V_k}(P)$. However, these techniques require an explicit representation of P to start with. In some cases, we are only presented P more abstractly. There two common situations:

- We are provided a set of n objects X, and a bivariate function $\texttt{d} : X \times X \to \mathbb{R}$ that returns a distance between them. For instance, we can put two cities into an airline website, and it may return a dollar amount for the cheapest flight between those two cities. This dollar amount is our "distance."
- We are simply provided a matrix $D \in \mathbb{R}^{n \times n}$, where each entry $D_{i,j}$ is the distance between the ith and jth point. In the first scenario, we can calculate such a matrix D.

Multi-Dimensional Scaling (MDS) has the goal of taking such a distance matrix D for n points and giving low-dimensional (typically) Euclidean coordinates to these points so that the embedded points have similar spatial relations to that described in D. If we had some original data set A which resulted in D, we could just apply PCA to find the embedding. It is important to note, in the setting of MDS we are typically just given D, and *not* the original data A. However, as we will show next, we can derive a matrix that will act like AA^T using only D.

A *similarity matrix* M is an $n \times n$ matrix where entry $M_{i,j}$ is the similarity between the ith and the jth data point. The similarity often associated with Euclidean distance $\|a_i - a_j\|$ is the standard inner (or dot

product) $\langle a_i, a_j \rangle$. We can write

$$\|a_i - a_j\|^2 = \|a_i\|^2 + \|a_j\|^2 - 2\langle a_i, a_j \rangle,$$

and hence

$$\langle a_i, a_j \rangle = \frac{1}{2}\left(\|a_i\|^2 + \|a_j\|^2 - \|a_i - a_j\|^2\right). \tag{7.1}$$

Next we observe that for the $n \times n$ matrix AA^T the entry $[AA^T]_{i,j} = \langle a_i, a_j \rangle$. So it seems hopeful we can derive AA^T from D using equation (7.1). That is we can set $\|a_i - a_j\|^2 = D_{i,j}^2$. However, we need also need values for $\|a_i\|^2$ and $\|a_j\|^2$.

Since the embedding has an arbitrary shift to it (if we add a shift vector s to *all* embedding points, then no distances change), then we can arbitrarily choose a_1 to be at the origin. Then $\|a_1\|^2 = 0$ and $\|a_j\|^2 = \|a_1 - a_j\|^2 = D_{1,j}^2$. Using this assumption and equation (7.1), we can then derive the similarity matrix AA^T. Then we can run the eigen-decomposition on AA^T and use the coordinates of each point along the first k eigenvectors to get an embedding. This is known as *classical MDS*.

It is often used for k as 2 or 3 so the data can be easily visualized.

There are several other forms that try to preserve the distance more directly, where as this approach is essentially just minimizing the squared residuals of the projection from some unknown original (high-dimensional embedding). One can see that we recover the distances with no error if we use all n eigenvectors – if they exist. However, as mentioned, there may be less than n eigenvectors, or they may be associated with complex eigenvalues. So if our goal is an embedding into $k = 3$ or $k = 10$, there is no guarantee that this will work, or even what guarantees this will have. But MDS is used a lot nonetheless.

Exercises

We will use a dataset, here: `http://www.cs.utah.edu/~jeffp/teaching/FoDA/A.csv`

Q7.1: Read data set `A.csv` as a matrix $A \in \mathbb{R}^{30\times 6}$. Compute the SVD of A and report

1. the third right singular vector,
2. the second singular value, and
3. the fourth left singular vector.
4. What is the rank of A?

Compute A_k for $k = 2$.

1. What is $\|A - A_k\|_F^2$?
2. What is $\|A - A_k\|_2^2$?

Center A. Run PCA to find the best 2-dimensional subspace F to minimize $\|A - \pi_F(A)\|_F^2$. Report

1. $\|A - \pi_F(A)\|_F^2$ and
2. $\|A - \pi_F(A)\|_2^2$.

Q7.2: Consider another matrix $A \in \mathbb{R}^{8\times 4}$ with squared singular values $\sigma_1^2 = 10$, $\sigma_2^2 = 5$, $\sigma_3^2 = 2$, and $\sigma_4^2 = 1$.

1. What is the rank of A?
2. What is $\|A - A_2\|_F^2$, where A_2 is the best rank-2 approximation of A.
3. What is $\|A - A_2\|_2^2$, where A_2 is the best rank-2 approximation of A.
4. What is $\|A\|_2^2$?
5. What is $\|A\|_F^2$?

Let v_1, v_2, v_3, v_4 be the right singular vectors of A.

1. What is $\|Av_2\|^2$?
2. What is $\langle v_1, v_3\rangle$?
3. What is $\|v_4\|$?

Let $a_1 \in \mathbb{R}^4$ be the first row of A.

1. Write a_1 in the basis defined by the right singular vectors of A.

Q7.3: Consider two matrices A_1 and A_2 both in $\mathbb{R}^{10\times 3}$. A_1 has singular values $\sigma_1 = 20$, $\sigma_2 = 2$, and $\sigma_3 = 1.5$. A_2 has singular values $\sigma_1 = 8$, $\sigma_2 = 4$, and $\sigma_3 = 0.001$.

1. For which matrix will the power method converge faster to the top eigenvector of $A_1^T A_1$ (or $A_2^T A_2$, respectively), and why?

Given the eigenvectors v_1, v_2, v_3 of $A^T A$. Explain step by step how to recover the following. Specifically, you should write the answers as linear algebraic expressions in terms of v_1, v_2, v_3, and A; it can involve taking norms, matrix multiply, addition, subtraction, but not something more complex like SVD.

2. the second singular value of A
3. the first right singular vector of A
4. the third left singular vector of A

8 Clustering

This topic will focus on automatically grouping data points into subsets of similar points. There are numerous ways to define this problem, and most of them are quite messy. And many techniques for clustering actually lack a mathematical formulation. We will initially focus on what is probably the cleanest and most used formulation: assignment-based clustering which includes k-center and the notorious k-means clustering. But, for background, we will begin with a mathematical detour in Voronoi diagrams. Then we will also describe various other clustering formulations to give insight into the breadth and variety of approaches in this subdomain.

On clusterability: While we present a variety of clustering approaches, and there are many more in the data mining literature, its important to remember the following:
When data is easily or naturally clusterable, most clustering algorithms work quickly and well.
When data is not easily or naturally clusterable, then no algorithm will find good clusters.
These statement is perhaps debatable, and "easily or naturally clusterable" is not defined here. But there are definitely many ways to cluster data, and the most common ones all work on common sense data examples with clusters.

8.1 Voronoi Diagrams

Consider a set $S = \{s_1, s_2, \ldots, s_k\} \subset \mathbb{R}^d$ of sites. We would like to understand how these points can carve up the space $\mathbb{R}^d$.

We now think of this more formally as the *post office problem*. Let these k sites each define the location of a post office. For all points in $\mathbb{R}^d$ (e.g., a point on the map for points in $\mathbb{R}^2$), we would like to assign it to the closest post office. For a fixed point $x \in \mathbb{R}^d$, we can just check the distance to each post office:

$$\phi_S(x) = \arg\min_{s_i \in S} \|x - s_i\|.$$

However, this may be slow (naively take $O(k)$ time for each point x), and does not provide a general representation or understanding for all points. The "correct" solution to this problem is the Voronoi diagram.

The Voronoi diagram decomposes $\mathbb{R}^d$ into k regions (a *Voronoi cell*), one for each site. The region for site s_i is defined.

$$R_i = \{x \in \mathbb{R}^d \mid \phi_S(x) = s_i\}.$$

If we have these regions nicely defined, this solves the post office problem. For any point x, we just need to determine which region it lies in (for instance in $\mathbb{R}^2$, once we have defined these regions, through an extension of binary search, we can locate the region containing any $x \in \mathbb{R}^2$ in only $O(\log k)$ time). But what do these regions look like, and what properties do they have.

Voronoi edges and vertices. We will start our discussion in $\mathbb{R}^2$. Further, we will assume that the sites S are in *general position*: in this setting, it means that no set of three points lie on a common line, and that no set of four points lie on a common circle.

The boundary between two regions R_i and R_j, called a *Voronoi edge*, is a line or line segment. This edge $e_{i,j}$ is defined

$$e_{i,j} = \{x \in \mathbb{R}^2 \mid \|x - s_i\| = \|x - s_j\| \leq \|x - s_\ell\| \text{ for all } \ell \neq i, j\}$$

as the set of all points equal distance to s_i and s_j, and not closer to any other point s_ℓ.

Why is this set a line segment? If we only have two points in S, then it is the bisector between them. Draw a circle centered at any point x on this bisector, and if it intersects one of s_i or s_j, it will also intersect the other. This is true since we can decompose the squared distance from x to s_i along orthogonal components: along the edge, and perpendicular to the edge from s_i to $\pi_{e_{i,j}}(s_i)$.

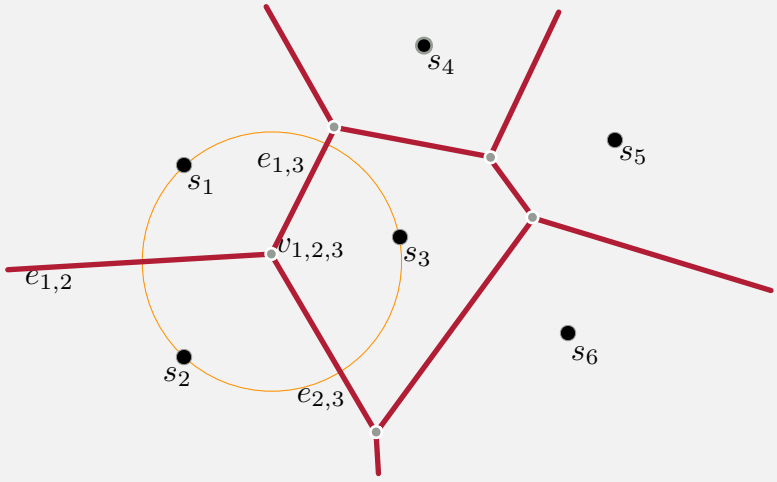
Similarly, a Voronoi vertex $v_{i,j,\ell}$ is a point where three sites s_i, s_j, and s_ℓ are all equidistance, and no other points are closer:

$$v_{i,j,k} = \{x \in \mathbb{R}^2 \mid \|x - s_i\| = \|x - s_j\| = \|x = s_\ell\| \leq \|x - s_k\| \text{ for all } k \neq i, j, \ell\}.$$

This vertex is the intersection (and end point) of three Voronoi edges $e_{i,i}$, $e_{i,\ell}$, and $e_{j,\ell}$. Think of sliding a point x along an edge $e_{i,j}$ and maintaining the circle centered at x and touching s_i and s_j. When this circle grows to where it also touches s_ℓ, then $e_{i,j}$ stops.

Example: Voronoi Diagram

See the following example with $k = 6$ sites in $\mathbb{R}^2$. Notice the following properties: edges may be unbounded, and the same with regions. The circle centered at $v_{1,2,3}$ passes through s_1, s_2, and s_3. Also, Voronoi cell R_3 has $5 = k - 1$ vertices and edges.



Size complexity. So how complicated can these Voronoi diagrams get? A single Voronoi cell can have $k-1$ vertices and edges. So can the entire complex be of size $O(k^2)$ (each of k regions requiring complexity $O(k)$)? No. The Voronoi vertices and edges describe a planar graph (i.e., can be drawn in the plane, $\mathbb{R}^2$, with no edges crossing). And planar graphs have asymptotically the same number of edges, faces, and vertices. In particular, Euler's Formula for a planar graph with n vertices, m edges, and k faces is that $k+n-m = 2$. And Kuratowski's criteria says for $n \geq 3$, then $m \leq 3n - 6$. Hence, $k \leq 2n - 4$ for $n \geq 3$. The duality construction to Delauney triangulations (discussed below) will complete the argument. Since there are k faces (the k Voronoi cells, one for each site), then there are also $O(k)$ edges and $O(k)$ vertices. In particular, there will be precisely $2k - 5$ vertices and $3k - 6$ edges.

However, this does not hold in $\mathbb{R}^3$. In particular, for $\mathbb{R}^3$ and $\mathbb{R}^4$, the complexity (number of cells, vertices, edges, faces, etc) is $O(k^2)$. This means, there could be roughly as many edges as their are pairs of vertices!

But it can get much worse. In $\mathbb{R}^d$ (for general d) then the complexity is $O(k^{\lceil d/2 \rceil})$. This is a lot. Hence, this structure is impractical to construct in high dimensions.

Curse of dimensionality: The *curse* of dimensionality, refers to when a problem has a nice, simple, and low-complexity structure in low dimensions, but then becomes intractable and unintuitive in high-dimensions. For instance, many geometric properties, like the size complexity of Voronoi diagrams, have linear complexity and are easy to draw in low-dimensions, but are unintuitive, and have size complexity that grows exponentially as the dimension grows.

Moreover, since this structure is explicitly tied to the post office problem, and the nearest neighbor function ϕ_S, it indicates that (a) in $\mathbb{R}^2$ this function is nicely behaved, but (b) in high dimensions, it is quite complicated.

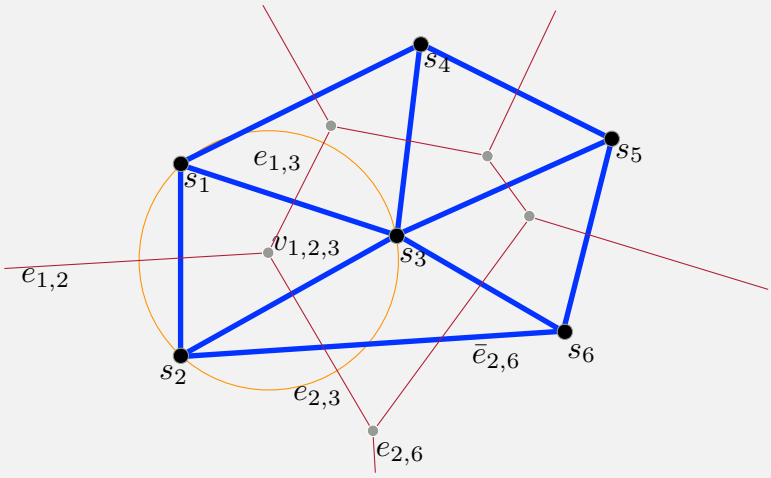
8.1.1 Delaunay Triangulation

A fascinating aspect of the Voronoi diagram is that it can be converted into a very special graph called the *Delaunay triangulation* where the sites S are vertices. This is the *dual* of the Voronoi diagram.

- Each face R_i of the Voronoi diagram maps to a vertex s_i in the Delaunay triangulation.
- Each vertex $v_{i,j,\ell}$ in the Voronoi diagram maps to a triangular face $f_{i,j,\ell}$ in the Delaunay triangulation.
- Each edge $e_{i,j}$ in the Voronoi diagram maps to an edge $\bar{e}_{i,j}$ in the Delaunay triangulation.

Example: Delaunay Triangulation

See the following example with 6 sites in $\mathbb{R}^2$. Notice that every edge, face, and vertex in the Delaunay triangulation corresponds to a edge, vertex, and face in the Voronoi diagram. Interestingly, the associated edges may not intersection; see $e_{2,6}$ and $\bar{e}_{2,6}$.



Because of the duality between the Voronoi diagram and the Delaunay triangulation, their complexities are the same. That means the Voronoi diagram is of size $O(k)$ for k sites in $\mathbb{R}^2$, but more generally is of size $O(k^{\lceil d/2 \rceil})$ in $\mathbb{R}^d$.

The existence of of the Delaunay triangulation shows that there always exist a triangulation: A graph with vertices of a given set of points $S \subset \mathbb{R}^2$ so that all edges are straightline segments between the vertices, and each face is a triangle. In fact, there are many possible triangulations: one can always simply construct some triangulation greedily, draw any possible edges that does not cross other edges until no more can be drawn.

The Delaunay triangulation, however, is quite special. This is the triangulation that maximizes the smallest angle over all triangles; for meshing applications in graphics and simulation, skinny triangles (with small angles) cause numerical issues, and so these are very useful.

In circle property. Another cool way to define the Delaunay triangulation is through the *in circle* property. For any three points, the smallest enclosing ball either has all three points on the boundary, or has two points on the boundary and they are antipodal to each other. Any circle with two points antipodal on the boundary s_i and s_j (i.e., s_i and s_j are on exact opposite spots on the circle), and contains no other points, then the edge $e_{i,j}$ is in the Delaunay triangulation. The set of edges defined by pairs of points defining empty circles is a subset of the Delaunay triangulation called the *Gabriel graph*.

Any circle with three points on its boundary s_i, s_j, and s_ℓ, and no points in its interior, then the face $f_{i,j,\ell}$ is in the Delaunay triangulation, as well as its three edges $e_{i,j}$, $e_{i,\ell}$ and $e_{j,\ell}$. But does not imply those edges are in the Gabriel graph.

For instance, on a quick inspection, (in the example above) it may not be clear if edge $e_{3,5}$ or $e_{4,6}$ should be in the Delaunay triangulation. Clearly it can not be both since they cross. But the ball with boundary through s_3, s_4, and s_6 would contain s_5, so the face $f_{3,4,6}$ cannot be in the Delaunay triangulation. On the other hand the ball with boundary through s_3, s_6, and s_5 does not contain s_4 or any other points in S, so the face $f_{3,5,6}$ is in the Delaunay triangulation.

8.1.2 Connection to Assignment-based Clustering

So what is the connection to clustering? Given a large set $X \subset \mathbb{R}^d$ of size n, we would like to find a set of k sites S (post office locations) so that each point $x \in X$ is near some post office. This is a proxy problem. So given a set of sites S, determining for each $x \in X$ which site is closest is exactly determined by the Voronoi diagram.

In particular, in *assignment-based clustering*, each cluster is represented by a single site $s \in S$, to which all other points in the cluster are "assigned." Consider a set X, and distance $\mathbf{d} : X \times X \to \mathbb{R}_+$, and the output is a set $S = \{s_1, s_2, \ldots, s_k\}$. Using the Voronoi diagram this implicitly defines a set of clusters where $\phi_S(x) = \arg\min_{s \in S} \mathbf{d}(x, s)$. Then the *$k$-means clustering problem* is to find the set S of k clusters (often, but not always as a subset of X) to

$$\text{minimize} \sum_{x \in X} \mathbf{d}(\phi_S(x), x)^2.$$

So we want every point assigned to the closest center, and want to minimize the sum of the squared distance of all such assignments.

There are several other useful variants including:

- the *k-center clustering problem*: minimize $\max_{x \in X} \mathbf{d}(\phi_S(x), x)$
- the *k-median clustering problem*: minimize $\sum_{x \in X} \mathbf{d}(\phi_S(x), x)$
  The *k-mediod* variant is similar to k-median, but restricts that the centers S must be a subset of P.

Moreover, the *mixture of Gaussians* approach will allow for more flexibility in the assignment and more modeling power in the shape of each cluster.

8.2 Gonzalez Algorithm for k-Center Clustering

We begin with what is arguably the simplest and most general clustering algorithm: the Gonzalez algorithm. This algorithm directly maps to the k-center formulation, where again every point is assigned to the closest center, and the goal is to minimize the length of the *longest* distance of any such assignment pairing.

Unfortunately, the k-center clustering problem is NP-hard to solve exactly.[1] In fact, for the general case, it is NP-hard to find a clustering within a factor 2 of the optimal cost!

Luckily, there is a simple, elegant, and efficient algorithm that achieves this factor 2 approximation. That is, for value k and a set X, it finds a set of k sites $\hat{S}$ so

$$\max_{x \in X} \mathbf{d}(\phi_{\hat{S}}(x), x) \leq 2 \max_{x \in X} \mathbf{d}(\phi_{S^*}(x), x),$$

where S^* is the set of k sites with optimal cost. That is $S^* = \text{argmin}_{S:|S|=k} \max_{x \in X} \mathbf{d}(\phi_S(x), x)$. Moreover, in practice, this often works better than the worst case theoretical guarantee. This algorithm, presented as Algorithm 8.2.1, is usually attributed to Teofilo F. Gonzalez (1985), hence the name. It can be described with the following maxim: *Be greedy, and avoid your neighbors!*.

Algorithm 8.2.1 Gonzalez Algorithm(X, k)

Choose $s_1 \in X$ arbitrarily. Initialize $S = \{s_1\}$.
for $i = 2$ to k **do**
  Set $s_i = \arg\max_{x \in X} \mathbf{d}(x, \phi_S(x))$.
  Include $S = S \cup \{s_i\}$.

The algorithm is iterative, building up the set of sites S over the run of algorithm. It initializes the first site s_1 arbitrarily. And it maintains a set of sites S which have i sites after i steps; that is initially the set only contains the one site s_1 chosen so far. Then it adds to the set the point $x \in X$ which is furthest from any of the current sites; that is the one with largest $\phi_S(x)$ value. This function ϕ_S changes over the course of the algorithm as more sites are added to S.

In other words, back to our maxim, it always adds a new site that is furthest from current set of sites.

[1]The term *NP-hard* (i.e., non-deterministic polynomial time hard), refers to being as hard as a set of problems in terms of runtime with respect to the size of their input. For these problems, if the correct solution is found, it can be verified quickly (in time polynomial in the size of the input), but to find that correct solution the only known approaches are essentially equivalent to a brute-force search over an exponentially large set of possible solutions, taking time exponential in the input size. It is not known if it *must* take time exponential in the input size, or if there may be a solution which takes time polynomial in the input size (the class of problems P); but in practice, it is often assumed this is not possible. In short, these problems are probably very hard to solve efficiently.

Implementation Hints:
This algorithm only takes time proportional to $kn = O(kn)$. There are k rounds, and each round can be done in about n time. We maintain the map $\phi_{S_i}(x)$ for each x; in the algorithm we will just use an array ϕ, where $\phi[j]$ stores the index of the assigned site for the jth point x_j. That is, at any point in the algorithm, point x_j is assigned to $s_{\phi[j]}$. When a new s_i is found, and added to the set of centers, all n assignments $\phi_{S_i}(x)$ can be updated in linear time (i.e., $O(n)$ time), by checking each distance $\mathtt{d}(x, \phi_{S_{i-1}}(x))$ against $\mathtt{d}(x, s_i)$ and switching the assignment if the latter is smaller. Then the minimum can be found in the next round on a linear scan (or on the same linear scan).

Algorithm: Gonzalez-Clustering(X, k)

Choose $s_1 \in X$ arbitrarily, and set $\phi[j] = 1$ for all $j \in [n]$
for $i = 2$ to k **do**
  $M = 0, \quad s_i = x_1$
  for $j = 1$ to n **do**
    if $\mathtt{d}(x_j, s_{\phi[j]}) > M$ **then**
      $M = \mathtt{d}(x_j, s_{\phi[j]}), \quad s_i = x_j$
  for $j = 1$ to n **do**
    if $\mathtt{d}(x_j, s_{\phi[j]}) > \mathtt{d}(x_j, s_i)$ **then**
      $\phi[j] = i$

We summarize the properties of this simple, elegant, and efficient algorithm. This works for any metric $\mathtt{d}$, and the 2-approximation guarantee will still hold. The resulting set of sites S are a subset of the input points X, this means it does not rely on the input points being part of a nice, easy to visualize, space, e.g., $\mathbb{R}^d$. However, this algorithm biases the choice of centers to be on the "edges" of the dataset, each chosen site is as far away from existing sites as possible, and then is never adjusted. There are heuristics to adjust centers afterwards, including the algorithms for k-means and k-mediod clustering.

8.3 Lloyd's Algorithm for k-Means Clustering

Probably the most famous clustering formulation is k-means, which we recall is an instance of assignment-based clustering. Specifically, the *k-means clustering problem* is to find the set S of k clusters to minimize

$$\mathsf{cost}(X, S) = \sum_{x \in X} \|\phi_S(x) - x\|^2.$$

So we want every point assigned to the closest site, and want to minimize the sum of the squared distance of all such assignments.

We emphasize the term "k-means" refers to a problem formulation, *not to any one algorithm*. There are many algorithms with aim of solving the k-means problem formulation, exactly or approximately. We will mainly focus on the most common: Lloyd's algorithm. Unfortunately, it is commonly written in data mining literature "the k-means algorithm," which typically should be more clearly stated as Lloyd's algorithm.

8.3.1 Lloyd's Algorithm

When people think of the k-means problem, they usually think of the following algorithm. It is usually attributed to Stuart P. Lloyd from a document in 1957, although it was not published until 1982.[2]

[2] Apparently, the IBM 650 computer Lloyd was using in 1957 did not have enough computational power to run the (very simple) experiments he had planned. This was replaced by the IBM 701, but it did not have quite the same "quantization" functionality as

The algorithm is again fairly simple and elegant; however, unlike Gonzalez's algorithm for k-center, the sites are all iteratively updated. It initializes with any set of k sites, and then iteratively updates the locations of these sites. Moreover, this assumes that the input data X lies in $\mathbb{R}^d$; it can be generalized to X in a few other Euclidean-like spaces as well.

As shown in Algorithm 8.3.1, each iteration is split into two steps. In the first step each point $x \in X$ is explicitly mapped to the closest site $s_i = \phi_S(x)$. In the second step, the set of points $X_i = \{x \in X \mid \phi_S(x) = s_i\}$ which are mapped to the ith site s_i are gathered, and that site is adjusted to the average of all points in X_i.

This second step is why we assume $X \in \mathbb{R}^d$ since the average operation can be naturally defined for $\mathbb{R}^d$ (as $\frac{1}{|X_i|}\sum_{x \in X_i} x$). This results in a point in $\mathbb{R}^d$, but not necessarily a point in X. So, differing from Gonzalez algorithm, this does not in general return a set of sites S which are drawn from the input set X.

Algorithm 8.3.1 Lloyd's Algorithm(X, k)

Choose k points $S \subset X$ *arbitrarily?*
repeat
  for all $x \in X$: assign x to X_i so $\phi_S(x) = s_i$ *the closest site $s_i \in S$ to x*
  for all $s_i \in S$: update $s_i = \frac{1}{|X_i|}\sum_{x \in X_i} x$ *the average of $X_i = \{x \in X \mid \phi_S(x) = s_i\}$*
until (the set S is unchanged, *or other termination condition*)

Convergence. This iterative process is continued until some termination condition is obtained. Algorithm 8.3.1 describes this as when the location of S is unchanged between iterates.

In general if the main loop has R iterations (or rounds), then this takes time proportional to Rnk (and can be made closer to $Rn \log k$ with faster nearest neighbor search in some low-dimensional cases). But how large is R; that is, how many iterations do we need?

We argue that the cost function $\mathsf{cost}(X, S)$ always decreases and thus the number of rounds is finite. In fact, each round improves upon the results of the previous round. However, the number of steps until this absolute termination condition is met may be quite large (indeed in certain cases exponentially large!). So in practice such a strict variant is rarely used. Rather, it is common to run for a specified and fixed number of rounds, say $R = 10$ or $R = 20$. Another common approach, as in gradient descent, is to check if the change in $\mathsf{cost}(X, S)$ is less than a sufficiently small threshold.

the IBM 650, and the work was forgotten. Lloyd was also worried about some issues regarding the k-means problem not having a unique minimum.

Geometry of Iterative Convergence

The number of rounds is finite. This is true since the cost function $\mathsf{cost}(X, S)$ will always decrease. To see this, writing it as a sum over S.

$$\mathsf{cost}(X, S) = \sum_{x \in X} \|\phi_S(x) - x\|^2 = \sum_{s_i \in S} \sum_{x \in X_i} \|s_i - x\|^2.$$

Then in each step of the **repeat-until** loop, this must decrease. The first step holds since it moves each $x \in X$ to a subset X_i with the corresponding center s_i closer to (or the same distance to) x than before. So for each x the term $\|x - s_i\|$ is reduced (or the same). The second step holds since for each inner sum $\sum_{x \in X_i} \|s_i - x\|^2$, the single point s_i which minimizes this cost is precisely the average of X_i (see T2). So reassigning s_i as described also decreases the cost (or keeps it the same). Importantly, if the cost decreases each step, then it cannot have the same set of centers S on two different steps, since that would imply the assignment sets $\{X_i\}$ would also be the same. Thus, in order for this to happen, the cost would need to decrease after the first occurrence, and then increase to obtain the second occurrence, which is not possible.
Since, there are finite ways each set of points can be assigned to different clusters, then, the algorithm terminates in a finite number of steps.

Unfortunately, no matter how the convergence condition is chosen, this algorithm is **not** guaranteed to find the optimal set of sites. It is easy to get stuck in a local minimum, and this may happen somewhat commonly with a large k. There are two typically ways to help manage the consequences of this property: careful initialization and random restarts.

A random restart, uses the insight that the choice of initialization can be chosen randomly, and this starting point is what determines what minimum is reached. So, then the basic Lloyd's algorithm can be run multiple times (e.g., 3-5 times) with different random initializations. Then, ultimately the set of sites S, across all random restarts, which obtain the lowest $\mathsf{cost}(X, S)$ is returned.

Initialization. The initial paper by Lloyd advocates to choose the initial partition of X into disjoint subsets $X_1, X_2, \ldots, X_k$ *arbitrarily*. However, some choices will not be very good. For instance, if we randomly place each $x \in X$ into some X_i, then (by the central limit theorem) we expect all $s_i = \frac{1}{|X_i|} \sum_{x \in X_i} x$ to all be close to the mean of the full data set $\frac{1}{|X|} \sum_{x \in X} x$.

A bit safer way to initialize the data is to choose a set $S \subset X$ at random. Since each s_i is chosen separately (not as an average of data points), there is no convergence to mean phenomenon. However, even with this initialization, we may run Lloyd's algorithm to completion, and find a sub-optimal solution (*a local minimum!*). However, there are scenarios where such random initialization is unlikely to find the right local minimum, even under several random restarts.

A more principled way to choose an initial set S is to use an algorithm like Gonzalez (see T8.2), or k-means++ (see T.8.4). In particular, the initialization by Gonzalez is guaranteed to be within a factor of 2 of the optimal k-center objective. While this objective is different than the k-means objective, it in that sense cannot be too bad; and then Lloyd's algorithm will only improve from that point on. k-means++ is similar, but randomized, and more tuned to the specific k-means objective.

```
from sklearn.cluster import KMeans
import numpy as np
X = np.array([[2, 3], [2, 4], [2, 2],[7, 1], [7, 2], [7, 3]])
kmeans = KMeans(n_clusters=2).fit(X)

print kmeans.labels_
```

```
# [1 1 1 0 0 0]

print kmeans.cluster_centers_
#[[ 7.  2.]
# [ 2.  3.]]
```

Number of clusters. So what is the right value of k? Like with PCA, there is no perfect answer towards choosing how many dimensions the subspace should be. When k is not given to you, typically, you would run with many different values of k. Then create a plot of $\mathsf{cost}(S, X)$ as a function of k. This cost will always decrease with larger k; but of course $k = n$ is of no use. At some point, the cost will not decrease much between values (this implies that probably two centers are used in the same grouping of data, so the squared distance to either is similar). Then this is a good place to choose k.

8.3.2 k-Means++

The initialization approach for Lloyd's algorithm, that is most tuned to the k-means objective is known as k-means++, or D^2-sampling. This algorithm is randomized, unlike Gonzalez, so it is compatible with random restarts. Indeed analyses which argues for approximation guarantees in Lloyd's algorithm require several random restarts with k-means++.

Algorithm 8.3.2 k-Means++ Algorithm(X, k)

Choose $s_1 \in X$ arbitrarily. Initialize $S = \{s_1\}$.
for $i = 2$ to k **do**
  Choose s_i from X with probability proportional to $\mathbf{d}(x, \phi_S(x))^2$.
  Update $S = S \cup \{s_i\}$.

As Algorithm 8.3.2 describes, the structure is like Gonzalez algorithm, but is not completely greedy. It iteratively chooses each next center randomly – the further the squared distances is from an existing center, the more likely it is chosen. For a large set of points (perhaps grouped together) which are far from an existing center, then it is very likely that one (it does not matter so much which one) of them will be chosen as the next center. This makes it likely that any "true" cluster will find some point as a suitable representative.

The critical step in this algorithm, and the difference from Gonzalez, is choosing the new center proportional to some value (the value $\mathbf{d}(x, \phi_S(x))^2$). Recall that this task has an elegant solution called the Partition of Unity, discussed in the context of Importance Sampling.

8.3.3 k-Mediod Clustering

Like many algorithms in this book, Lloyd's depends on the use of a SSE cost function. Critically, this works because for $X \in \mathbb{R}^d$, then

$$\frac{1}{|X|}\sum_{x \in X} x = \mathsf{average}(X) = \arg\min_{s \in \mathbb{R}^d} \sum_{x \in X} \|s - x\|^2.$$

There are not in general similar properties for other costs functions, or when X is not in $\mathbb{R}^d$. For instance, one may want to solve the k-medians problem where one just minimizes the sum of (non-squared) distances. In particular, this subset has no closed form solution for $X \in \mathbb{R}^d$ for $d > 1$. Most often gradient descent-type approaches are used for finding an updated center under a k-median objective.

An alternative to the averaging step is to choose

$$s_i = \arg\min_{s \in X_i} \sum_{x \in X_i} \mathbf{d}(x, s)$$

where (x, s) is an arbitrary measure (like non-squared Euclidean distance) between x and s. That is, we choose an s from the set X_i. This is particularly useful when X is in a non-Euclidean metric space where averaging may not be well-defined. For the specific case where $\mathbf{d}(x, s) = \|x - s\|$ (for the k-median problem), then this variant of the formulation is called *k-mediods*. Moreover, the most common hueristic for this problem is adapt Lloyd's algorithm by substituting the averaging step with the minimum over all as points in that set, as in Algorithm 8.3.3.

Algorithm 8.3.3 k-Mediod Hueristic(X, k)

Choose k points $S \subset X$ *arbitrarily?*
repeat
  for all $x \in X$: assign x to X_i so $\phi_S(x) = s_i$ *the closest site $s_i \in S$ to x*
  for all $s_i \in S$: update $s_i = \arg\min_{s \in X_i} \sum_{x \in X_i} \mathbf{d}(x, s)$ *the best choice of site from X_i*
until (the set S is unchanged, *or other termination condition*)

This optimization step of choosing each s_i in each iteration is now considerably slower to implement than the averaging step in Lloyd's algorithm. However, that it always chooses s_i so that it is an element of X can be an advantage. This allows the algorithm to generalize to use arbitrary distance functions $\mathbf{d}$ in place of Euclidean distance. Now if the data objects X are customers, documents, or websites an appropriate distance function can be used, and the site at the center of a cluster can be an element of the data set, and thus an interpretable quantity.

Ethical Questions with Clustering and Outliers

The k-mediod clustering formulation is often used because it is general, but also fairly robust to outliers. A single outlier point may be chosen as a separate cluster center for k-center or k-means clustering, with no other points in that cluster. However, for k-mediods or k-median clustering it is more likely to be added to an existing cluster without as dramatically increasing the cost function. Typically this property is viewed as a positive for k-mediods in that the resulting cluster subsets are not greatly distorted by a single outlier.

However, if these outliers are then added to a cluster for which it does not fit, it may cause other problems. For instance, if a model is fit to each cluster, then this outlier may distort this model.

Consider the case where each data point is a person applying for a loan. It may be an outlier is included in a cluster, and has had multiple bankruptcies that caused them to default on the loan, and cost previous banks to lose a lot of money. This outlier may distort the model of that cluster so it predicts all of the customers in it are not expected to return on the loan, in expectation. On the other hand, excluding that data point would allow the model to predict more accurately that most of the customers will repay the loan, and be profitable. As a data modeler, what obligations do you have to check for such outliers within a clustering? What is a good way to mitigate the effects of instability in the way data points are clustered? Does the answer to the previous questions remain the same, even if the expected overall profit level for the bank should stay the same?

8.3.4 Soft Clustering

Sometimes it is not desirable to assign each point to exactly one cluster. Instead, we may split a point between one or more clusters, assigning a fractional value to each. This is known as *soft clustering* whereas the original formulation is known as *hard clustering*.

There are many ways to achieve a soft clustering. For instance, consider the following Voronoi diagram-based approach based on natural neighbor interpolation (NNI). Let $V(S)$ be the Voronoi diagram of the sites S (which decomposes $\mathbb{R}^d$). Then construct $V(S \cup x)$ for a particular data point x; the Voronoi diagram of the sites S with the addition of one point x. For the region R_x defined by the point x in $V(S \cup x)$, overlay it on the original Voronoi diagram $V(S)$. This region R_x will overlap with regions R_i in the original Voronoi diagram; compute the volume v_i for the overlap with each such region. Then the fractional weight for x into each site s_i is defined $w_i(x) = v_i / \sum_{i=1}^{n} v_i$.

We can plug any such step into Lloyd's algorithm, and then recalculate s_i as the weighted average of all points partially assigned to the ith cluster.

8.4 Mixture of Gaussians

The k-means formulation tends to define clusters of roughly equal size. The squared cost discourages points far from any center. It also, does not adapt much to the density of individual centers.

An extension is to fit each cluster X_i with a Gaussian distribution $\mathcal{G}_d(\mu_i, \Sigma_i)$, defined by a mean μ_i and a covariance matrix Σ_i. Recall that the pdf of a d-dimensional Gaussian distribution is defined

$$f_{\mu,\Sigma}(x) = \frac{1}{(2\pi)^{d/2}} \frac{1}{\sqrt{|\Sigma|}} \exp\left(-\frac{1}{2}(x-\mu)^T \Sigma^{-1}(x-\mu)\right)$$

where $|\Sigma|$ is the determinant of Σ . Previously we had only considered this distribution where $\Sigma = I$ was the identity matrix, and it was ignored in the notation. For instance, for $d = 2$, and the standard deviation in the x-direction of X is σ_x, and in the y-direction is σ_y, and their correlation is ρ, then

$$\Sigma = \begin{bmatrix} \sigma_x^2 & \rho\sigma_x\sigma_y \\ \rho\sigma_x\sigma_y & \sigma_y^2 \end{bmatrix}.$$

Now the goal is, given a parameter k, find a set of k pdfs $F = \{f_1, f_2, \ldots, f_k\}$ where $f_i = f_{\mu_i,\Sigma_i}$ to maximize

$$\prod_{x \in X} \max_{f_i \in F} f_i(x),$$

or equivalently to minimize

$$\sum_{x \in X} \min_{f_i \in F} -\log(f_i(x)).$$

For the special case where we restrict that $\Sigma_i = I$ (the identity matrix) for each mixture, then one can check that the second formulation (the log-likelihood version) is equivalent to the k-means problem (depending on choice of hard or soft clustering).

This hints that we can adapt Lloyds algorithm towards this problem as well. To replace the first step of the inner loop, we assign each $x \in X$ to the Gaussian which maximizes $f_i(x)$:

$$\text{for all } x \in X\text{: assign } x \text{ to } X_i \text{ so } i = \arg\max_{i \in 1 \ldots k} f_i(x).$$

But for the second step, we need to replace a simple average with an estimation of the best fitting Gaussian to a data set X_i. This is also simple. First, calculate the mean as $\mu_i = \frac{1}{|X_i|} \sum_{x \in X_i} x$. Then calculate the covariance matrix Σ_i of X_i as the sum of outer products

$$\Sigma_i = \sum_{x \in X_i} (x - \mu_i)(x - \mu_i)^T.$$

Indeed the covariance of the Gaussian fit to each X_i is better understood by its principal component analysis. Calculating μ_i, and subtracting from each $x \in X_i$ is the centering step. Letting $\bar{X}_i = \{x - \mu_i \mid x \in X_i\}$, then $\Sigma_i = VS^2V^T$ where $[U, S, V] = \mathsf{svd}(\bar{X}_i)$. Now, the top principle directions $v_1, v_2, \ldots$ describe the directions of most variance.

8.4.1 Expectation-Maximization

The standard way to fit a mixture of Gaussians actually uses a soft-clustering. Each point $x \in X$ is given a weight $w_i = f_i(x)/\sum_i f_i(x)$ for its assignment to each cluster. Then the mean and covariance matrix is estimated using weighted averages. The soft-clustering variant of the procedure extension to Lloyd's algorithm described above is outlined in Algorithm 8.4.1.

Algorithm 8.4.1 EM Algorithm for Mixture of Gaussians

Choose k points $S \subset X$
for all $x \in X$: set $w_i(x) = 1$ for $\phi_S(x) = s_i$, and $w_i(x) = 0$ otherwise
repeat
  for $i \in [1 \dots k]$ **do**
    Calculate $W_i = \sum_{x \in X} w_i(x)$ — *the total weight for cluster i*
    Set $\mu_i = \frac{1}{W_i} \sum_{x \in X} w_i(x) x$ — *the weighted average*
    Set $\Sigma_i = \frac{1}{W_i} \sum_{x \in X} w_i (x - \mu_i)(x - \mu_i)^T$ — *the weighted covariance*
  for $x \in X$ **do**
    for all $i \in [1 \dots k]$: set $w_i(x) = f_i(x)/\sum_i f_i(x)$ — *partial assignments using $f_i = f_{\mu_i, \Sigma_i}$*
until ($\sum_{x \in X} \sum_{i=1}^{k} -\log(w_i(x) \cdot f_i(x))$ has small change)

This procedure is the classic example of a framework called *expectation-maximization*. This is an alternating optimization procedure, which alternates between maximizing the probability of some model (the partial assignment step) and calculating the most likely model using expectation (the average, covariance estimating step).

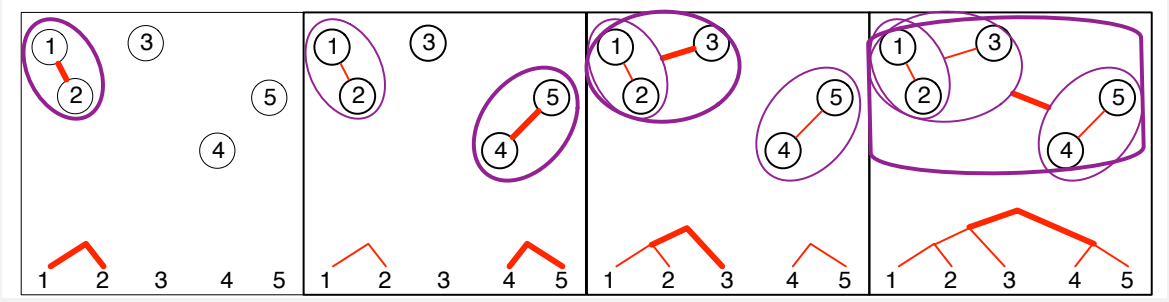
But this is a much more general framework. It is particularly useful in situations (like this one) where the true optimization criteria is messy and complex, often non-convex; but it can be broken into two or more steps where each step can be solved with a (near) closed form. Or if there is no closed form, but each part is individually convex, then gradient descent can be invoked.

8.5 Hierarchical Clustering

Clustering can provide more than a partition, it can also provide a hierarchy of the clusters. That is, at the root of a hierarchy all data points are in the same cluster. And at the leaf nodes, each data point is in its own cluster. Then the intermediate layers can provide various levels of refinement, depending on how many or how tightly related clusters are desired.

Example: Hierarchy of Clusters

This small example shows the hierarchy of clusters found on $n = 5$ data points. Associated with these clusters and this progression is a tree, showing the local order in which points are merged into clusters.



There are two basic approaches towards this: top-down and bottom-up. The top-down approach starts with all data points in the same cluster, and iteratively partitions the data into 2 (or sometimes more) parts. The main task then, is given a data set, how to split it into two components as best possible. The canonical top-down clustering approach is spectral clustering. This is primarily based on graph ideas, but can be generalized to work with any choice of similarity function $\mathbf{s}$ (or implicitly a linked distance function $\mathbf{d}$), so the base object is an affinity matrix instead of the adjacency matrix of a graph. An in depth discussion is deferred to the Graphs chapter.

The bottom-up approach, often called *hierarchical agglomerative clustering* takes the opposite tact, outlined in Algorithm 8.5.1. Starting with a cluster for each data point, its main task is to determine which pair of clusters to join into a single cluster. As the standard choice is, *the closest two clusters*, then the key element is to study distances between clusters.

Algorithm 8.5.1 Hierarchical Agglomerative Clustering

Each $x_i \in X$ is a separate cluster S_i.
while there exist at least 2 clusters **do**
  Find the *closest* two clusters S_i, S_j
  Merge S_i and S_j into a single cluster

There are many ways to define such a distance $\mathbf{d}_C$. Each is defined with respect to another general distance function $\mathbf{d}$ which can be applied to individual data points, and in some cases also representative objects. The most common are:

- *Single Link:* $\mathbf{d}_C(S_i, S_j) = \text{argmin}_{s_i \in S_i, s_j \in S_j} \mathbf{d}(s_i, s_j)$.
  This takes the distance between the closest two points among all points in the clusters. This allows the clusters to adjust to the shape and density of the data, but can lead to oddly shaped clusters as well.

- *Mean Link:* $\mathbf{d}_C(S_i, S_j) = \sum_{s_i \in S_i} \sum_{s_j \in S_j} \mathbf{d}(s_i, s_j)$.
  This takes the average distance between pairs of points in a cluster. It behaves similar to the k-means objective.

- *Complete Link:* $\mathbf{d}_C(S_i, S_j) = \text{argmax}_{s_i \in S_i, s_j \in S_j} \mathbf{d}(s_i, s_j)$.
  This enforces "round" clusters. It can only join two clusters if all points are similar.

- *Center Link:* $\mathbf{d}_C(S_i, S_j) = \mathbf{d}(c_i, c_j)$ where c_i and c_j are central points representing S_i and S_j.
  The choice of how to define the central point is varied. For instance it could be the average, median, representative median as in k-means, k-medians, or k-mediods objectives. Or it could be any other easy to determine or robust way to represent the clusters.

These variants of clusters provide additional structural information beyond center-based clusters, in that they also provide which clusters are close to each other, or allow a user to refine to different levels of granularity. However, this comes at the cost of efficiency. For instance, on n data points, these algorithms naively take time proportional to n^3. Some improvements or approximations can reduce this to closer to time proportional to n^2, but may add instability to the results.

Ethical Questions with Forced Hierarchies

Hierarchies can create powerful visuals to help make sense of complex data sets that are compared using a complex distance function over abstract data representations. For instance, phylogentic trees are the dominate way to attempt to present evolutionary connections between species. However, these connections can provide false insights by those over-eager to make new scientific or business connections.

A common way to model and predict the careers of professional athletes is to use their early years to cluster and group athletes with others who had careers before them. Then an up-and-coming athlete can be predicted to have careers similar to those in the same cluster. However, if a new mold of athlete – e.g., basketball players who shoot many 3-pointers, and are very tall – has many junior players, but not senior players to use to predict their career. If they are grouped with other tall players, a coach or general manager may treat them as rebounders minimizing their effectiveness. Or if they are treated as shooting guards, they may be given more value. How might the use of the clustering and its hierarchy be used to mitigate this potential downsides of this possible models?

Such effects are amplified in less scrutinize hiring and management situations. Consider a type of job applicant who has traditionally not be hired in a role, how will their clustering among past applicants be harmful or helpful for their chance at being hired, or placed in a management role? And as a data modeler, what is your role in aiding in these decisions?

8.6 Mean Shift Clustering

Now for something completely different. Clustering is a very very broad field with no settled upon approach. To demonstrate this, we will quickly review an algorithm called *mean shift clustering*. This algorithm shifts each data point individually to its weighted center of mass. It terminates when all points converge to isolated sets.

First begin with a bivariate kernel function $K : X \times X \to \mathbb{R}$ such as the (unnormalized) Gaussian kernel

$$K(x, p) = \exp(-\|x - p\|^2/\sigma^2)$$

for some given bandwidth parameter σ. The weighted center of mass around each point $p \in X$ is then defined as

$$\mu(p) = \frac{\sum_{x \in X} K(x, p)x}{\sum_{x \in X} K(x, p)}.$$

The algorithm just shifts each point to its center of mass: $p \leftarrow \mu(p)$.

Algorithm 8.6.1 Mean Shift

repeat
  for all $p \in X$: calculate $\mu(p) = \frac{\sum_{x \in X} K(x,p)x}{\sum_{x \in X} K(x,p)}$.
  for all $p \in X$: set $p \leftarrow \mu(p)$.
until (the average change $\|p - \mu(p)\|$ is small)

This algorithm does not require a parameter k. However, it has other parameters, most notably the choice of kernel K and its bandwidth σ. With the Gaussian kernel (since it has infinite support, $K(x, p) > 0$ for all x, p), it will only stop when all x are at the same point. Thus the termination condition is also important. Alternatively, a different kernel with bounded support may terminate automatically (without a

specific condition); for this reason (and for speed) truncated Gaussians (i.e., $K_\tau^{\text{trunc}}(x,p) = \{0 \text{ if } \|x-p\| > \tau;\ \text{ otherwise } K(x,p)\}$) are often used.

This algorithm not only clusters the data, but also is a key technique for *de-noising* data. This is a process that not just removes noise (as often thought of as outliers), but attempts to adjusts points to where they should have been before being perturbed by noise – similar to mapping a point to its cluster center.

Geometry of Kernel Density Estimates

A kernel density estimate is a powerful modeling tool that allows one to take a finite set of points $P \subset \mathbb{R}^d$ and a smoothing operator, a kernel K, and transform them into a continuous function $\text{KDE}_P : \mathbb{R}^d \to \mathbb{R}$.

The most common kernel is the Gaussian kernel

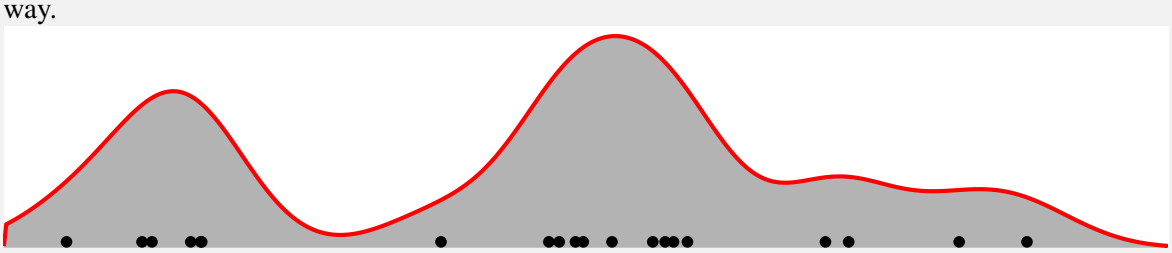
$$K(p,x) = \frac{1}{\sigma^d\sqrt{2\pi^d}} \exp(-\|x-p\|^2/(2\sigma^2)),$$

which is precisely the evaluation of the Gaussian distribution $\mathcal{G}_d(x)$ with mean p and standard deviation σ evaluated at x. Thus it can be interpreted as taking the "mass" of a point p and spreading it out according to this probability distribution.

Now, a *kernel density estimate* is defined at a point $x \in \mathbb{R}^d$ as

$$\text{KDE}_P(x) = \frac{1}{|P|}\sum_{p\in P} K(p,x).$$

That is, it gives each point $p \in P$ a mass of $\frac{1}{|P|}$ (each weighted uniformly), then spreads each of these $|P|$ masses out using the Gaussian kernels, and sums them up. While P is discrete in nature, the kernel density estimate KDE_P allows one to interpolate between these points in a smooth natural way.



Exercises

We will use two datasets, here: `http://www.cs.utah.edu/~jeffp/teaching/FoDA/P.csv` and here: `http://www.cs.utah.edu/~jeffp/teaching/FoDA/Q.csv`

Q8.1: Download data sets P and Q. Both have 120 data points, each in 6 dimensions, can be thought of as data matrices in $\mathbb{R}^{120\times 6}$. For each, run some algorithm to construct the k-means clustering of them. Diagnose how many clusters you think each data set should have by finding the solution for k equal to $1, 2, 3, \ldots, 10$.

Q8.2: Draw the Voronoi diagram of the following set of points.

Q8.3: What should you do, if running Lloyd's algorithm for k-means clustering ($k = 2$), and you reach this scenario, where the algorithm terminates? (The black circles are data points and red stars are the centers).

Q8.4: Construct a data set X with 5 points in $\mathbb{R}^2$ and a set S of $k = 3$ sites so that Lloyds algorithm will have converged, but there is another set S', of size $k = 3$, so that $\mathsf{cost}(X, S') < \mathsf{cost}(X, S)$. Explain why S' is better than S, but that Lloyds algorithm will not move from S.

9 Classification

This topic returns to prediction. Unlike linear regression where we were predicting a numeric value, in this case we are predicting a class: winner or loser, yes or no, rich or poor, positive or negative. Ideas from linear regression can be applied here, but we will instead overview a different, but still beautiful family of techniques based on linear classification.

This is perhaps *the* central problem in data analysis. For instance, you may want to predict:

- will a sports team win a game?
- will a politician be elected?
- will someone like a movie?
- will someone click on an ad?
- will I get a job? (If you can build a good classifier, then probably yes!)

Each of these is typically solved by building a general purpose classifier (about sports or movies etc), then applying it to the problem in question.

9.1 Linear Classifiers

Our input here is a point set $X \subset \mathbb{R}^d$, where each element $x_i \in X$ also has an associated label y_i. And $y_i \in \{-1, +1\}$.

Like in regression, our goal is prediction and generalization. We assume each $(x_i, y_i) \sim \mu$; that is, each data point pair, is drawn iid from some fixed but unknown distribution. Then our goal is a function $g : \mathbb{R}^d \to \mathbb{R}$, so that if $y_i = +1$, then $g(x_i) \geq 0$ and if $y_i = -1$, then $g(x_i) \leq 0$.

We will restrict that g is linear. For a data point $x \in \mathbb{R}^d$, written $x = (x^{(1)}, x^{(2)}, \ldots, x^{(d)})$ we enforce that

$$g(x) = \alpha_0 + x^{(1)}\alpha_1 + x^{(2)}\alpha_2 + \ldots + x^{(d)}\alpha_d = \alpha_0 + \sum_{j=1}^{d} x^{(j)}\alpha_j,$$

for some set of scalar parameters $\alpha = (\alpha_0, \alpha_1, \alpha_2, \ldots, \alpha_d)$. Typically, different notation is used: we set $b = \alpha_0$ and $w = (w_1, w_2, \ldots, w_d) = (\alpha_1, \alpha_2, \ldots, \alpha_d) \in \mathbb{R}^d$. Then we write

$$g(x) = b + x^{(1)}w_1 + x^{(2)}w_2 + \ldots + x^{(d)}w_d = \langle w, x \rangle + b.$$

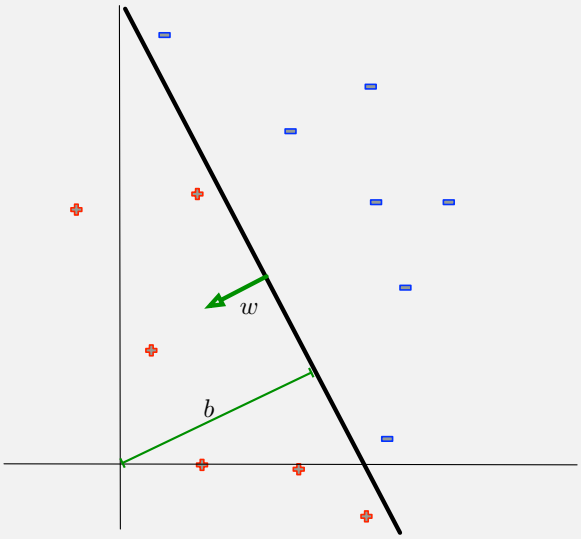
We can now interpret (w, b) as defining a halfspace in $\mathbb{R}^d$. Here w is the normal of that halfspace boundary (the single direction orthogonal to it) and b is the distance from the origin $\mathbf{0} = (0, 0, \ldots, 0)$ to the halfspace boundary in the direction $w/\|w\|$. Because w is normal to the halfspace boundary, b is also distance from the closest point on the halfspace boundary to the origin (in any direction).

We typically ultimately use w as a unit vector, but it is not important since this can be adjusted by changing b. Let w, b be the desired halfspace with $\|w\| = 1$. Now assume we have another w', b' with $\|w'\| = \beta \neq 1$ and $w = w'/\|w'\|$, so they point in the same direction, and b' set so that they define the same halfspace. This implies $b' = b/\beta$. So the normalization of w can simply be done post-hoc without changing any structure.

Recall, our goal is $g(x) \geq 0$ if $y = +1$ and $g(x) \leq 0$ if $y = -1$. So if x lies directly on the halfspace then $g(x) = 0$.

Example: Linear Separator in $\mathbb{R}^2$

Here we show a set $X \in \mathbb{R}^2$ of 13 points with 6 labeled $+$ and 7 labeled $-$. A linear classifier perfectly separates them. It is defined with a normal direction w (pointing towards the positive points) and an offset b.



Using techniques we have already learned, we can immediately apply two approaches towards this problem.

Linear classification via linear regression. For each data points $(x_i, y_i) \in \mathbb{R}^d \times \mathbb{R}$, we can immediately represent x_i as the value of d explanatory variables, and y_i as the single dependent variable. Then we can set up a $n \times (d+1)$ matrix $\tilde{X}$, where the ith row is $(1, x_i)$; that is the first coordinate is 1, and the next d coordinates come from the vector x_i. Then with a $y \in \mathbb{R}^n$ vector, we can solve for

$$\alpha = (\tilde{X}^T \tilde{X})^{-1} \tilde{X}^T y$$

we have a set of $d+1$ coefficients $\alpha = (\alpha_0, \alpha_1, \ldots, \alpha_d)$ describing a linear function $g : \mathbb{R}^d \to \mathbb{R}$ defined

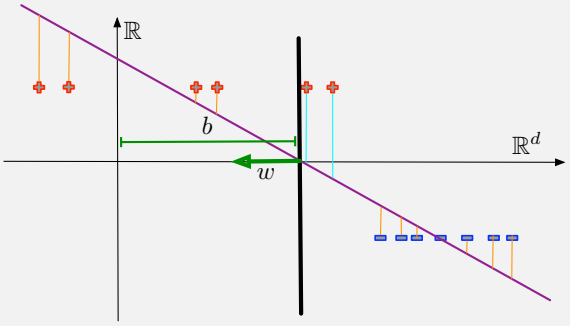
$$g(x) = \langle \alpha, (1, x) \rangle.$$

Hence $b = \alpha_0$ and $w = (\alpha_1, \alpha_2, \ldots, \alpha_d)$. For x such that $g(x) > 0$, we predict $y = +1$ and for $g(x) < 0$, we predict $y = -1$.

However, this approach is optimizing the wrong problem. It is minimizing how close our predictions $g(x)$ is to -1 or $+1$, by minimizing the sum of squared errors. But our goal is to minimize the number of mispredicted values, not the numerical value.

Example: Linear Regression for Classification

We show 6 positive points and 7 negative points in $\mathbb{R}^d$ mapped to $\mathbb{R}^{d+1}$. All of the d-coordinates are mapped to the x-axis. The last coordinate is mapped to the y-axis and is either $+1$ (a positive points) or -1 (a negative points). Then the best linear regression fit is shown, and the points where it has y-coordinate 0 defines the boundary of the halfspace. Note, despite there being a linear separator, this method misclassifies two points because it is optimizing the wrong measure.



Linear classification via gradient descent. Since, the linear regression SSE cost function is not the correct one, what is the correct one? We might define a cost function Δ

$$\Delta(g,(X,y)) = \sum_{i=1}^{n}(1 - \mathbb{1}(\mathsf{sign}(y_i) = \mathsf{sign}(g(x_i)))$$

which uses the *identity function* $\mathbb{1}$ (defined $\mathbb{1}(\textsc{True}) = 1$ and $\mathbb{1}(\textsc{False}) = 0$) to represent the *number* of misclassified points. This is what we would like to minimize.

Unfortunately, this function is discrete, so it does not have a useful (or well-defined) gradient. And, it is also not convex. Thus, encoding g as a $(d+1)$-dimensional parameter vector $(b, w) = \alpha$ and running gradient descent is not feasible.

However, most classification algorithms run some variant of gradient descent. To do so we will use a different cost function as a proxy for Δ, called a *loss function*. We explain this next.

9.1.1 Loss Functions

To use gradient descent for classifier learning, we will use a proxy for Δ called a *loss* functions $\mathcal{L}$. These are sometimes implied to be convex, and their goal is to approximate Δ. And in most cases, they are *decomposable*, so we can write

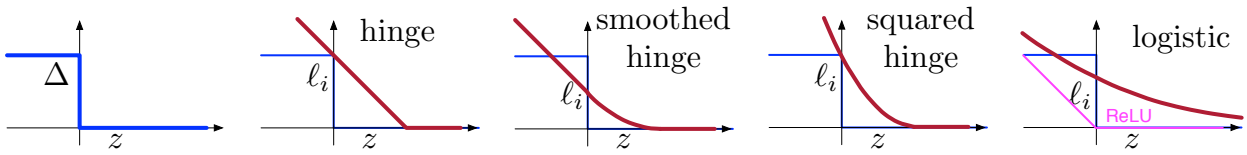
$$\begin{aligned}\mathcal{L}(g,(X,y)) &= \sum_{i=1}^{n} \ell_i(g,(x_i,y_i)) \\ &= \sum_{i=1}^{n} \ell_i(z_i) \text{ where } z_i = y_i g(x_i).\end{aligned}$$

Note that the clever expression $z_i = y_i g(x_i)$ handles when the function $g(x_i)$ correctly predicts the positive or negative example in the same way. If $y_i = +1$, and correctly $g(x_i) > 0$, then $z_i > 0$. On the other hand, if $y_i = -1$, and correctly $g(x_i) < 0$, then also $z_i > 0$. For instance, the desired cost function, Δ is written

$$\Delta(z) = \begin{cases} 0 & \text{if } z \geq 0 \\ 1 & \text{if } z < 0. \end{cases}$$

Most *loss functions* $\ell_i(z)$ which are convex proxies for Δ mainly focus on how to deal with the case $z_i < 0$ (or $z_i < 1$). The most common ones include:

- hinge loss: $\ell_i = \max(0, 1 - z)$
- smoothed hinge loss: $\ell_i = \begin{cases} 0 & \text{if } z \geq 1 \\ (1-z)^2/2 & \text{if } 0 < z < 1 \\ \frac{1}{2} - z & \text{if } z \leq 0 \end{cases}$
- squared hinge loss: $\ell_i = \max(0, 1 - z)^2$
- logistic loss: $\ell_i = \ln(1 + \exp(-z))$



The hinge loss is the closest convex function to Δ; in fact it strictly upper bounds Δ. However, it is non-differentiable at the "hinge-point," (at $z = 1$) so it takes some care to use it in gradient descent. The smoothed hinge loss and squared hinge loss are approximations to this which are differentiable everywhere. The squared hinge loss is quite sensitive to outliers (similar to SSE). The smoothed hinge loss (related to the Huber loss) is a nice combination of these two loss functions.

The logistic loss can be seen as a continuous approximation to the ReLU (rectified linear unit) loss function, which is the hinge loss shifted to have the hinge point at $z = 0$. The logistic loss also has easy-to-take derivatives (does not require case analysis) and is smooth everywhere. Minimizing this loss for classification is called *logistic regression*.

9.1.2 Cross Validation and Regularization

Ultimately, in running gradient descent for classification, one typically defines the overall cost function f also using a regularization term $r(\alpha)$. For instance $r(\alpha) = \|\alpha\|^2$ is easy to use (has nice derivatives) and $r(\alpha) = \|\alpha\|_1$ (the L_1 norm) induces sparsity, as discussed in the context of regression. In general, the regularizer typically penalizes larger values of α, resulting in some bias, but less over-fitting of the data.

The regularizer $r(\alpha)$ is combined with a loss function $\mathcal{L}(g_\alpha, (X, y)) = \sum_{i=1}^n \ell_i(g_\alpha, (x_i, y_i))$ as

$$f(\alpha) = \mathcal{L}(g_\alpha, (X, y)) + \eta r(\alpha),$$

where $\eta \in \mathbb{R}$ is a *regularization parameter* that controls how drastically to regularize the solution.

Note that this function $f(\alpha)$ is still decomposable, so one can use batch, incremental, or most commonly stochastic gradient descent.

Cross-validation. Backing up a bit, the *true* goal is not minimizing f or $\mathcal{L}$, but predicting the class for new data points. For this, we again assume all data is drawn iid from some fixed but unknown distribution. To evaluate how well out results generalizes, we can use cross-validation (holding out some data from the training, and calculating the expected error of Δ on these held out "testing" data points).

We can also choose the regularization parameter η by choosing the one that results in the best generalization on the test data after training using each on some training data.

Ethical Questions with Data Imbalance and Classification

Generalization goals are typically phrased so a data point has the smallest expected loss. However, this can lead to issues when the data is imbalanced. Consider a data set consisting for two types of students applying to college. Let us use hair color as a proxy, and assume each applicant has blue hair or green hair. If each type has different criteria that would make them successful, a classifier can still use the hair color attribute in conjunction with other traits to make useful prediction across both types of applicant. However, if there are significantly more applicants with green hair, then the classifier will have a smaller expected loss (and even better generalization error) if it increases weights for the traits that correspond with green-haired applicants succeeding. This will make it more accurate on the green-haired applicants, but perhaps less accurate on the blue-haired applicants, yet more accurate overall on average across all applicants. This provides worse prediction and more uncertainty for the green-haired students due only to the color of their hair; it may even lower their overall acceptance rate. What can be done as a data analyst to find such discrepancies? And should something be changed in the modeling if these discrepancies are found?

9.2 Perceptron Algorithm

Of the above algorithms, generic linear regression is not solving the correct problem, and gradient descent methods do not really use any structure of the problem. In fact, as we will see we can replace the linear function $g_\alpha(x) = \langle \alpha, (1, x) \rangle$ with any function g (even non-linear ones) as long as we can take the gradient.

Now we will introduce the *perceptron algorithm* which explicitly uses the linear structure of the problem. (Technically, it only uses the fact that there is an inner product – which we will exploit in generalizations.)

Simplifications: For simplicity, we will make several assumptions about the data. First we will assume that the best linear classifier (w^*, b^*) defines a halfspace whose boundary passes through the origin. This implies $b^* = 0$, and we can ignore it. This is basically equivalent to (for data point $(x_i', y_i) \in \mathbb{R}^{d'} \times \mathbb{R}$ using $x_i = (1, x_i') \in \mathbb{R}^d$ where $d' + 1 = d$.

Second, we assume that for all data points (x_i, y_i) that $\|x_i\| \leq 1$ (e.g., all data points live in a unit ball). This can be done by choosing the point $x_{\max} \in X$ with largest norm $\|x_{\max}\|$, and dividing all data points by $\|x_{\max}\|$ so that point has norm 1, and all other points have smaller norms.

Finally, we assume that there exists a perfect linear classifier. One that classifies each data point to the correct class. There are variants to deal with the cases without perfect classifiers, but are beyond the scope of this text.

The algorithm. Now to run the algorithm, we start with some normal direction w (initialized as any positive point), and then add mis-classified points to w one at a time.

Algorithm 9.2.1 Perceptron(X, y)

Initialize $w = y_i x_i$ for any $(x_i, y_i) \in (X, y)$
repeat
  For any (x_i, y_i) such that $y_i \langle x_i, w \rangle < 0$ (is mis-classified) : update $w \leftarrow w + y_i x_i$
until (no mis-classified points **or** T steps)
return $w \leftarrow w / \|w\|$

Basically, if we find a mis-classified point (x_i, y_i) and $y_i = +1$, then we set $w = w + x_i$. This "pushes" w more in the direction of x_i, but also makes it longer. Having w more in the direction of w, tends to make it have dot-product (with a normalized version of w) closer to 1.

Similar, if we find a mis-classified point (x_i, y_i) with $y_i = -1$, the we set $w = w - x_i$; this pushes the negative of w more towards x_i, and thus w more away from x_i, and thus its dot product more likely to be negative.

Implementation Hints:
To implement the Perceptron algorithm, inside the inner loop we need to find some misclassified point (x_i, y_i), if one exists. This can require another implicit loop. A common approach would be to, for some ordering of points $(x_1, y_1), (x_2, y_2), \ldots (x_n, y_n)$ keep an iterator index i that is maintained outside the **repeat-until** loop. It is modularly incremented every step: it loops around to $i = 1$ after $i = n$. That is, the algorithm keeps cycling through the data set, and updating w for each misclassified point if observes.

Algorithm: Perceptron(X, y)

Initialize $w = y_i x_i$ for any $(x_i, y_i) \in (X, y)$; Set $i = 1; t = 0;$ LAST-UPDATE $= 1$
repeat
  if $(y_i \langle x_i, w \rangle < 0)$
    $w \leftarrow w + y_i x_i$
    $t = t + 1;$ LAST-UPDATE $= i$
  $i = i + 1 \mod n$
until $(t = T$ or LAST-UPDATE $= i)$
return $w \leftarrow w / \|w\|$

The margin. To understand why the perceptron works, we need to introduce the concept of a margin. Given a classifier (w, b), the *margin* is

$$\gamma = \min_{(x_i, y_i) \in X} y_i(\langle w, x_i \rangle + b).$$

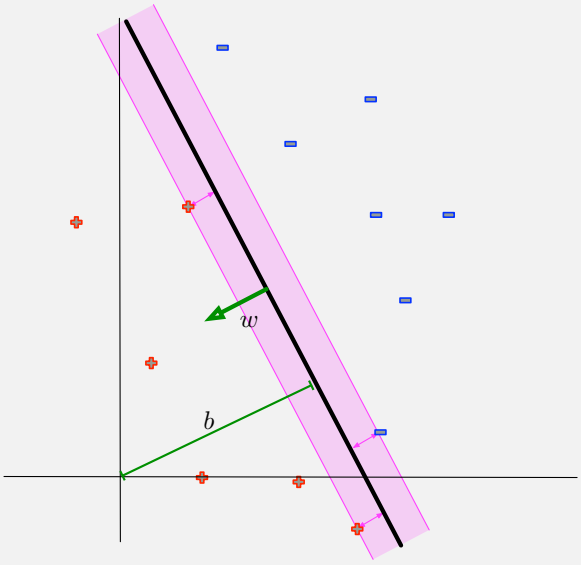
Its the minimum distance of any data point x_i to the boundary of the halfspace. In this sense the optimal classifier (or the maximum margin classifier) (w^*, b^*) is the one that maximizes the margin

$$(w^*, b^*) = \arg\max_{(w,b)} \min_{(x_i, y_i) \in X} y_i(\langle w, x_i \rangle + b)$$
$$\gamma^* = \min_{(x_i, y_i) \in X} y_i(\langle w^*, x_i \rangle + b^*).$$

A max-margin classifier, is one that not just classifies all points correctly, but does so with the most "margin for error." That is, if we perturbed any data point or the classifier itself, this is the classifier which can account for the most perturbation and still predict all points correctly. It also tends to *generalize* (in the cross-validation sense) to new data better than other perfect classifiers.

Example: Margin of Linear Classifier

For a set X of 13 points in $\mathbb{R}^2$, and a linear classifier defined with (w, b). We illustrate the margin in pink. The margin $\gamma = \min_{(x_i,y_i)} y_i(\langle w, x_i\rangle + b)$. The margin is drawn with an $\leftrightarrow$ for each support point.



The maximum margin classifier (w^*, b^*) for $X \subset \mathbb{R}^d$ can always be defined uniquely by $d + 1$ points (at least one negative, and at least one positive). These points $S \subset X$ are such that for all $(x_i, y_i) \in S$

$$\gamma^* = y_i\langle w^*, x_i\rangle + b.$$

These are known as the *support points*, since they "support" the margin strip around the classifier boundary.

Geometry of Why Perceptron Works

Here we will show that after at most $T = (1/\gamma^*)^2$ steps (where γ^* is the margin of the maximum margin classifier), then there can be no more mis-classified points.

To show this we will bound two terms as a function of t, the number of mistakes found. The terms are $\langle w, w^* \rangle$ and $\|w\|^2 = \langle w, w \rangle$; this is before we ultimately normalize w in the **return** step.
First we can argue that $\|w\|^2 \leq t$, since each step increases $\|w\|^2$ by at most 1:

$$\langle w+y_i x_i, w+y_i x_i \rangle = \langle w, w \rangle + (y_i)^2 \langle x_i, x_i \rangle + 2 y_i \langle w, x_i \rangle \leq \langle w, w \rangle + 1 + 0.$$

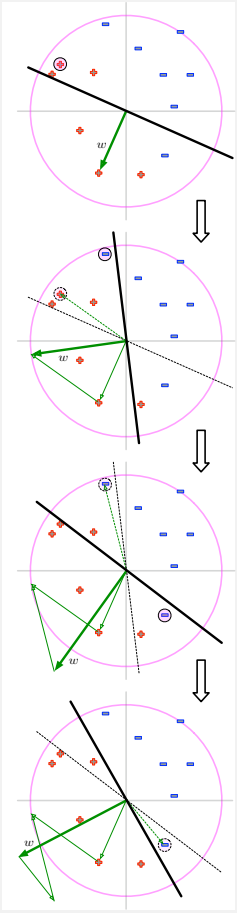
This is true since each $\|x_i\| \leq 1$, and if x_i is mis-classified, then $y_i \langle w, x_i \rangle$ is negative.
Second, we can argue that $\langle w, w^* \rangle \geq t\gamma^*$ since each step increases it by at least γ^*. Recall that $\|w^*\| = 1$

$$\langle w + y_i x_i, w^* \rangle = \langle w, w^* \rangle + (y_i) \langle x_i, w^* \rangle \geq \langle w, w^* \rangle + \gamma^*.$$

The inequality follows from the margin of each point being at least γ^* with respect to the max-margin classifier w^*.
Combining these facts ($\langle w, w^* \rangle \geq t\gamma^*$ and $\|w\|^2 \leq t$) together we obtain

$$t\gamma^* \leq \langle w, w^* \rangle \leq \langle w, \frac{w}{\|w\|} \rangle = \|w\| \leq \sqrt{t}.$$

Solving for t yields $t \leq (1/\gamma^*)^2$ as desired.



9.3 Kernels

It turns out all we need to get any of the above perceptron machinery to work is a well-defined (generalized) inner-product. For two vectors $p = (p_1, \ldots, p_d), x = (x_1, \ldots, x_d) \in \mathbb{R}^d$, we have always used as the inner product:

$$\langle p, x \rangle = \sum_{i=1}^{d} p_i \cdot x_i.$$

However, we can define inner products more generally as a kernel $K(p, x)$. For instance, we can use the following non-linear functions

- $K(p, x) = \exp(-\|p - x\|^2/\sigma^2)$ for the Gaussian kernel, with bandwidth σ,
- $K(p, x) = \exp(-\|p - x\|/\sigma)$ for the Laplace kernel, with bandwidth σ, and
- $K(p, x) = (\langle p, x \rangle + c)^r$ for the polynomial kernel of power r, with control parameter $c > 0$.

Are these linear classifiers? No. In fact, this is how you model various forms of non-linear classifiers. The "decision boundary" is no longer described by the boundary of a halfspace. For the polynomial kernel, the boundary must now be a polynomial surface of degree r. For the Gaussian and Laplace kernel it can be even more complex; the σ parameter essentially bounds the curvature of the boundary.

9.3.1 The Dual: Mistake Counter

To use a more general kernel within the Perceptron algorithm, we need a different interpretation of how to keep track of the weight vector w. Recall, that each step we increment w by $y_i x_i$ for some misclassified data point (x_i, y_i). Instead we will maintain a length n vector $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_n)$, where α_i represents the number of times that data point (x_i, y_i) has been misclassified. That we can rewrite

$$w = \sum_{i=1}^{n} \alpha_i y_i x_i.$$

That is, instead of directly maintaining w, we maintain a length n set of counters α, and keep track of which signed data points $y_i x_i$ to be added to reconstruct w as needed. Now in the linear case the function g, applied to a new data point $p \in \mathbb{R}^d$ becomes

$$g(p) = \langle w, p \rangle = \langle \sum_{i=1}^{n} \alpha_i y_i x_i, p \rangle = \sum_{i=1}^{n} \alpha_i y_i \langle x_i, p \rangle.$$

This seems wasteful to keep such a large vector $\alpha \in \mathbb{R}^n$ around, especially if the number of data points n becomes very large. In contrast, the size of the original version $w \in \mathbb{R}^d$ does not change as the dataset increases. However, we only need to keep track of the non-zero elements of α, and if we run perceptron, there are at most $(1/\gamma^*)^2$ of these. So this is not significantly more space, depending on the relationship between d and $1/\gamma^*$.

The beauty of this form, is that now we can easily replace $\langle x_i, p \rangle$ with any other kernel $K(x_i, p)$. That is, the function $g(p)$ now becomes generalized to

$$g(p) = \langle w, p \rangle = \sum_{i=1}^{n} \alpha_i y_i K(x_i, p).$$

Note this g is precisely a kernel density estimate, with some elements having negative weights (if $y_i = -1$). Then a point p is classified as positive if $g(p) > 0$ and negative otherwise.

9.3.2 Feature Expansion

If the margin is small, the data is not separable, or we simply do not want to deal with the unknown size of a mistake counter, there is another option to use these non-linear kernels. We can take all data, and apply a non-linear transformation to a higher-dimensional space so the problem is linear again.

For a polynomial kernel, on d-dimensional data points, this is equivalent to the polynomial regression expansion, described in Chapter **5.3**. For a two dimensional data point $p = (p_1, p_2)$, we map this to a 5-dimensional space

$$p \mapsto q = (q_1 = p_1, q_2 = p_2, q_3 = p_1^2, q_4 = p_1 p_2, q_5 = p_2^2).$$

Then we search over a 6-dimensional parameter space (b, w) with $w = (w_1, w_2, w_3, w_4, w_5)$ and (with an abuse of notation, since the dimension of w is not d) the kernel defined $K(p, w) = \langle q, w \rangle$. More specifically, the z associated with a data point (p, y) as input to a loss function $\ell(z)$ is defined

$$z = y \cdot (K(p, w) + b) = y \cdot (\langle q, w \rangle + b).$$

Note that the number of free parameters in the feature expansion version of the polynomial kernel is larger than when retaining the kernel in the form $K(x, p) = (\langle x, p \rangle + b)^r$, when it is only $d + 1$. In particular,

when $p \in \mathbb{R}^d$ and the polynomial degree r is large, then this dimensionality can get high very quickly; the dimension of q is $O(q^r)$.

Such expansion is also possible for many other radius basis kernel (e.g., Gaussian, Laplace), but it is only approximate and randomized. Usually it requires the dimensionality of q to be about 100 or more to get a faithful representation.

Like in the case of regression, overfitting can be an issue. However, this effect can be mostly controlled by the regularization term. In the case of the polynomial kernel limiting the polynomial degree r, prevents too complex a model. With the Gaussian and Laplace, and other similar kernels, the increased dimensionality does not lead to overfitting, but too small a value of σ may. In both cases these parameters (r and σ) can be appropriately chosen with cross-validation.

9.3.3 Support Vector Machines

A more general way to work with complex kernels is called a *support vector machine* or SVM. Like with the illustration of the margin for linear classifiers, there are a small number of data points which determine the margin, the support vectors. Just these points are enough to determine the optimal margin.

In the case of complex non-linear kernels (e.g., Gaussians), all of the points may be support vectors. Worse, the associated linear expanded space, the result of complete variable expansion, is actually infinite! This means, the true weight vector w would be infinite as well, so there is no feasible way to run gradient descent on its parameters.

However, in most cases the actual number of support vectors is small. Thus it will be useful to *represent* the weight vector w as a linear combination of these support vectors, without every explicitly constructing them. Consider a linear expansion of a kernel K to an m-dimensional space (think of m as being sufficiently large that it might as well be infinite). However, consider if there are only k support vectors $\{s_1, s_2, \ldots, s_k\} \subset X$ where X is the full data set. Each support vector $s_i \in \mathbb{R}^d$ has a representation $q_i \in \mathbb{R}^m$. But the normal vector $w \in \mathbb{R}^m$ can be written as a linear combination of the q_is; that is, for some parameters $\alpha_1, \alpha_2, \ldots$, we must be able to write

$$w = \sum_{i=1}^{k} \alpha_i q_i$$

Thus, given the support vectors $\{s_1, \ldots, s_k\}$ we can represent w in the span of S (and the origin), reparametrized as a k-dimensional vector $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_k)$. This α vector is precisely the mistake counter for only the support vectors (the non-zero components), although in this case the coordinates need not be integers.

More concretely, we can apply this machinery without ever constructing the q_i vectors. Each can be implicitly represented as the function $q_i = K(s_i, \cdot)$. Recall, we only ever need to use the q_i in $\langle q_i, w \rangle$. And we can expand w to

$$w = \sum_{i=1}^{k} \alpha_i q_i = \sum_{i=1}^{k} \alpha_i K(s_i, \cdot).$$

Given this expansion, if we consider a special class of kernels called "reproducing kernels" which include Gaussian and Laplace, then we have a super-cool property:

$$K(w, p) = \sum_{i=1}^{k} \alpha_i K(s_i, p).$$

Ultimately, for a data point (p, y), the z in the loss function $\ell(z)$ is defined

$$z = yK(w, p) = y \sum_{i=1}^{k} \alpha_i K(s_i, p).$$

There are multiple ways to actually optimize SVMs: the task of finding the support vectors $S = \{s_1, \ldots, s_k\}$, and assigning their weights $\alpha = \{\alpha_1, \ldots, \alpha_k\}$. One is to run the kernelized Perceptron algorithm, as outlined above. Alternatively, given a fixed set of support vectors S, one can directly optimize over α using gradient descent, including any loss function and regularizer as before with linear classifiers. Thus, if we do not know S, we can just assume $S = X$, the full data set. Then we can apply standard gradient descent over $\alpha \in \mathbb{R}^n$. As mentioned, in most cases, most α_i values are 0 (and those close enough to 0 can often be rounded to 0). Only the points with non-zero weights are kept as support vectors.

Alternatively, stochastic gradient descent works like perceptron, and may only use a fraction of the data points. If we use a version of Hinge Loss, only misclassified points or those near the boundary have a non-zero gradient. The very strongly classified points have zero gradient, and the associated α_i coordinates may remain 0. The proper choice of loss function, and regularizer, can induces sparsity on the α values; and the data points not used are not support vectors.

9.4 kNN Classifiers

Now for something completely different. There are many ways to define a classifier, and we have just touched on some of them. These include decision trees (which basically just ask a series of yes/no questions and are very interpretable) to deep neural networks (which are more complex, far less interpretable, but can achieve more accuracy). We will describe one more simple classifier.

The k-NN classifier (or *k-nearest neighbors classifier*) works as follows. Choose a scalar parameter k (it will be far simpler to choose k as an odd number, say $k = 5$). Next define a *majority* function $\mathsf{maj} : \{-1, +1\}^k \to \{-1, +1\}$. For a set $Y = (y_1, y_2, \ldots, y_k) \in \{-1, +1\}^k$ it is defined

$$\mathsf{maj}(Y) = \begin{cases} +1 & \text{if more than } k/2 \text{ elements of } Y \text{ are } +1 \\ -1 & \text{if more than } k/2 \text{ elements of } Y \text{ are } -1. \end{cases}$$

Then for a data set X where each element $x_i \in X$ has an associated label $y_i \in \{-1, +1\}$, define a k-nearest neighbor function $\phi_{X,k}(q)$ that returns the k points in X which are closest to a query point q. Next let sign report y_i for any input point x_i; for a set of inputs x_i, it returns the set of values y_i.

Finally, the k-NN classifier is

$$g(q) = \mathsf{maj}(\mathsf{sign}(\phi_{X,k}(q))).$$

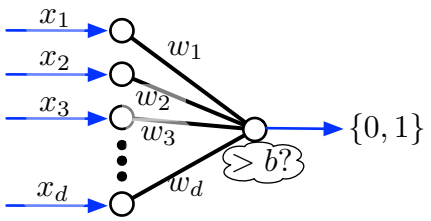
That is, it finds the k-nearest neighbors of query point q, and considers all of the class labels of those points, and returns the majority vote of those labels.

A query point q near many other positive points will almost surely return $+1$, and symmetrically for negative points. This classifier works surprisingly well for many problems but relies on a good choice of distance function to define $\phi_{X,k}$.

Unfortunately, the model for the classifier depends on all of X. So it may take a long time to evaluate on a large data set X. In contrast the functions g for non-kernelized methods above take $O(d)$ time to evaluate for points in $\mathbb{R}^d$, and thus are very efficient.

9.5 Neural Networks

A *neural network* is a learning algorithm intuitively based on how a neuron works in the brain. A neuron takes in a set of inputs $x = (x_1, x_2, \ldots, x_d) \in \mathbb{R}^d$, weights each input by a corresponding scalar $w = (w_1, w_2, \ldots, w_d)$ and "fires" a signal if the total weight $\sum_{i=1}^{d} w_i x_i$ is greater than some threshold b.



$$\sum_{j=1}^{d} w_i \cdot x_i - b = \langle x, w \rangle - b > 0?$$

A neural network, is then just a network or graph of neurons like these. Typically, these are arranged in layers. In the first layer, there may be d input values $x_1, x_2, \ldots, x_d$. These may provide the input to t neurons (each neuron might use fewer than all inputs). Each neuron produces an output $y_1, y_2, \ldots, y_t$. These outputs then serve as the input to the second layer, and so on.

In a neural net, typically each x_i and y_i is restricted to a range $[-1, 1]$ or $[0, 1]$ or $[0, \infty)$, not just the two classes $\{-1, +1\}$. Since a linear function does not guarantee this of its output, instead of a binary threshold, to achieve this at the output of each node, they typically add an *activation function* $\phi(y)$. Common ones are

- hyperbolic tangent : $\phi(y) = \tanh(y) = \frac{e^y - e^{-y}}{e^y + e^{-y}} \in [-1, 1]$
- sigmoid : $\phi(y) = \frac{1}{1+e^{-y}} = \frac{e^y}{e^y+1} \in [0, 1]$
- ReLu : $\phi(y) = \max(0, y) \in [0, \infty)$.

These functions are not linear, and nor binary. They act as a "soft" version of binary. The hyperbolic tangent and sigmoid stretch values near 0 away from 0. Large values stay large (in the context of the range). So it makes most values almost on the boundaries of the range. And importantly, they are differentiable.

The ReLu has become very popular. It is not everywhere differentiable, but is convex. Its basically the most benign version of activation, and is more likely the original neuron, in that if the value y is negative, it gets snapped to 0. If its positive, it keeps its value.

A two-layer neural network is already a powerful way to build classifiers, and with enough neurons in the middle layer, is capable of learning *any function*. However, *deep neural nets* with 3 or often many more layers (say 20 or 100 or more) have become very popular due to their effectiveness in many tasks, ranging from image classification to language understanding. To be effective, typically, this requires heavy engineering in how the layers are defined, and how the connections between layers are made.

Once the connections are determined, then the goal is to learn the weights on each neuron so that for a given input, a final neuron fires if the input satisfies some pattern (e.g., the input are pixels to a picture, and it fires if the picture contains a car). This is theorized to be "loosely" how the human brain works. Although, neural nets have pretty much diverged in how they learn from attempts to replicate the structure and process of learning in the human brain.

Training. Given a data set X with labeled data points $(x, y) \in X$ (with $x \in \mathbb{R}^d$ and $y \in \{-1, +1\}$), we already know how to train a single neuron so for input x it tends to fire if $y = 1$ and not fire if $y = -1$. It is just a linear classifier! So, we can use the perceptron algorithm, or gradient descent with a well-chosen loss function.

However, for neural networks to attain more power than simple linear classifiers, they need to be at least two layers, and are often deep (e.g., for "deep learning") with 20, 100, or more layers. For these networks, the perceptron algorithm no longer works since it does not properly propagate across layers. However, a version of gradient descent called *back-propagation* can be used. In short, it computes the gradient across the edge weights in a network by chaining partial derivatives backwards through the network.

Training deep nets to work can be quite finicky. Their optimization function is not convex, and without various training tricks, it can be very difficult to find a good global set of weights. Indeed the best and "right" methods are an active research area.

Exercises

Q9.1: Consider the following "loss" function. $\ell_i(z_i) = (1 - z_i)^2/2$, where for a data point (x_i, y_i) and prediction function f, then $z_i = y_i \cdot f(x_i)$. Predict how this might work within a gradient descent algorithm for classification.

Q9.2: Consider a data set (X, y), where each data point $(x_{1,i}, x_{2,i}, y_i)$ is in $\mathbb{R}^2 \times \{-1, +1\}$. Provide the psuedo-code for the Perceptron Algorithm using a polynomial kernel of degree 2. You can have a generic stopping condition, where the algorithm simply runs for T steps for some parameter T. *(There are several correct ways to do this, but be sure to explain how to use a polynomial kernel clearly.)*

Q9.3: Consider a set of 1-dimensional data points

$$(x_1 = 0, y_1 = +1)\ (x_2 = 1, y_1 = -1)\ (x_3 = 2, y_1 = +1)\ (x_4 = 4, y_1 = +1)$$

$$(x_5 = 6, y_1 = -1)\ (x_6 = 7, y_1 = -1)\ (x_7 = 8, y_1 = +1)\ (x_8 = 9, y_1 = -1)$$

Predict **-1** or **+1** using a kNN (k-nearest neighbor) classifier with $k = 3$ on the following queries.

1. $x = 3$
2. $x = 9$
3. $x = -1$

Q9.4: Consider the following Algorithm 1, called the *Double-Perceptron*. We will run this on an input set X consisting of points $X \in \mathbb{R}^{n \times d}$ and corresponding labels $y \in \{-1, +1\}$.

Algorithm 9.5.1 Double-Perceptron(X)

Initialize $w = y_i x_i$ for any $(x_i, y_i) \in (X, y)$
repeat
  For any (x_i, y_i) such that $y_i \langle x_i, w \rangle < 0$ (is mis-classified) : update $w \leftarrow w + 2 \cdot y_i x_i$
until (no mis-classified points **or** T steps)
return $w \leftarrow w/\|w\|$

For each of the following questions, the answer can be **faster**, **slower**, **the same**, or **not at all**. And should be accompanied with an explanation.

1. Compared with Algorithm 9.2.1 (Perceptron) in the notes, explain how this algorithm with converge.
2. Next consider transforming the input data set X (not the y labels) so that all coordinates are divided by 2. Now if we run *Double-Perceptron* how will the results compare to regular Perceptron (Algorithm 9.2.1) on the original data set X.
3. Finally, consider taking the original data set (X, y) and multiplying all entries in y by -1, then running the original Perceptron algorithm. How will the convergence compare to running the same Perceptron algorithm, on the original data set.

Q9.5: Consider a matrix $A \in \mathbb{R}^{n \times 4}$. Each row represents a customer (there are n customers in the database). The first column is the age of the customer in years, the second column is the number of days since the customer entered the database, the third column is the total cost of all purchases ever by the customer

in dollars, and the last column is the total profit in dollars generated by the customer. So each column has a different unit.

For each of the following operations, decide if it is **reasonable** or **unreasonable**.

1. Run simple linear regression using the first three columns to build a model to predict the fourth column.
2. Use k-means clustering to group the customers into 4 types using Euclidean distance between rows as the distance.
3. Use PCA to find the best 2-dimensional subspace, so we can draw the customers in a $\mathbb{R}^2$ in way that has the least projection error.
4. Use the linear classification to build a model based on the first three columns to predict if the customer will make a profit $+1$ or not -1.

Q9.6: Consider a "loss" function, called an *double-hinged loss function*

$$\ell_i(z) = \begin{cases} 0 & \text{if } z > 1 \\ 1 - z & \text{if } 0 \leq z \leq 1 \\ 1 & \text{if } z \leq 0. \end{cases}$$

where the overall cost for a dataset (X, y), given a linear function $g(x) = \langle (1, x), \alpha \rangle$ is defined $\mathcal{L}(g, (X, y)) = \sum_{i=1}^{n} \ell_i(y_i \cdot g(x_i))$.

1. What problems might a gradient descent algorithm have when attempting to minimize $\mathcal{L}$ by choosing the best α?
2. Explain if the problem would be better or worse using stochastic gradient descent?

10 Graphs

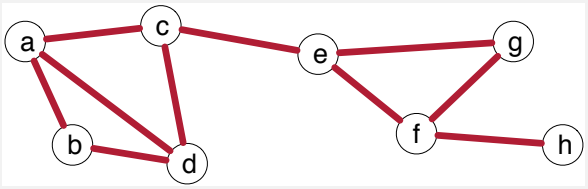
A central object in data analysis is a graph $G = (V, E)$ defined by a set of vertices V and edges between those vertices E. The vertices can serve as proxy for any data type (e.g., social network users, a companies products, or waypoints on a map), and the graph representation then simplifies the structure of these data items down to only their connectivity. This structure can of course be composed with many other structures we have studied (vectors, matrices, etc); but this chapter will focus specifically on what can be inferred from the graphs. In this section, we overview how graphs can be used to model movement of information, which reveals which vertices are most important. We also show how to cluster or find interesting subgraphs that capture where more interesting or meaningful patterns appear.

Basic definitions and models for graphs. Formally, a (undirected) graph $G = (V, E)$ is defined by a set of vertices $V = \{v_1, v_2, \ldots, v_n\}$ and a set of edges $E = \{e_1, e_2, \ldots, e_m\}$ where each edge e_j is an unordered pair of edges: $e_j = \{v_i, v_{i'}\}$. In a *directed graph* edges are ordered and $e_j = (v_i, v_{i'})$ indicates that node v_i points to $v_{i'}$, but not the other direction.

Two vertices v_1 and v_k are *connected* if there is a sequence of edges $\langle e_1, \ldots, e_{k-1} \rangle$ such that e_1 contains v_1, e_{k-1} contains v_k, and consecutive edges can be ordered so $e_j = \{v_i, v_{i+1}\}$ and $e_{j+1} = \{v_{i+1}, v_{i+2}\}$ where the second element in e_j is the same as the first in e_{j+1}. The graph distance $\mathbf{d}_E(v_i, v_{i'})$ (induced by edges set E) between any two vertices $v_i, v_{i'} \in V$ is the minimum number of edges requires to get from v_i to $v_{i'}$. It is a metric for undirected graphs; for directed graphs it may not be symmetric.

Example: Undirected Graph

Consider graph $G = (V, E)$ with the following vertices $V = \{a, b, c, d, e, f, g\}$ and edges $E = \Big\{\{a, b\}, \{a, c\}, \{a, d\}, \{b, d\}, \{c, d\}, \{c, e\}, \{e, f\}, \{e, g\}, \{f, g\}, \{f, h\}\Big\}$.



Alternatively, G can be represented as a matrix with 1 if there is an edge, and 0 otherwise.

	a	b	c	d	e	f	g	h
a	0	1	1	1	0	0	0	0
b	1	0	0	1	0	0	0	0
c	1	0	0	1	1	0	0	0
d	1	1	1	0	0	0	0	0
e	0	0	1	0	0	1	1	0
f	0	0	0	0	1	0	1	1
g	0	0	0	0	1	1	0	0
h	0	0	0	0	0	1	0	0

$$= \begin{pmatrix} 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

The distance $\mathbf{d}_E(a, g) = 3$, realized by the edges $\{a, c\}$, $\{c, e\}$, and $\{e, g\}$.
We will use this graph as a running example throughout much of this chapter.

Matrix representations of graphs. The matrix representation of a graph is essential for understanding many of its structure properties. We initiate this below with the two most basic and essential such representations, but will generalize and extend from these in the sections below. It should be noted that in practice, this can be *extremely* space-inefficient way to store the graph. These matrices have $|V|^2$ cells, however, in many cases the number of actual edges $|E|$ is much closer to the number of vertices (say $|E| \leq 20|V|$ or $|E| \leq 5|V|\log_2|V|$ or $|E| \leq |V|^{1.2}$ are often not unreasonable). On the small examples we can present in this text, these assumptions are not meaningful. But many graph datasets in industry (e.g., where $|V| > 1{,}000{,}000$) this difference can correspond with the dataset fitting easily on a laptop, or with requiring a cluster of computers.

The most natural representation of a graph as a matrix is the *adjacency matrix* A. For an n vertex graph, it is an $n \times n$ matrix that records 1 in $A_{i,j}$ for each each edge $e_{i,j} = \{v_i, v_j\}$. Entries are 0 otherwise.

The *degree matrix* is a square and diagonal $n \times n$ matrix. This means it only has non-zero entries on the diagonals $D_{i,i}$. In particular, $D_{i,i}$ records the degree of vertex v_i, which is the number of edges containing v_i. In particular, we will sometimes use the matrix $D^{-1/2}$. Recall a square matrix can be multiplied by itself, so in general for a positive integer power p, the matrix operation D^p refers to multiplying D by itself p times as $D \cdot D \cdot \ldots \cdot D$. However, for fractional powers this is less intuitive, and in generally not always defined. For diagonal matrices with positive entries on all diagonal elements, we can simply define D^p as the diagonal matrix replacing each $D_{i,i}$ with $D_{i,i}^p$. In particular, this means it is easy to invert a diagonal matrix since D^{-1} is just replaces each $D_{i,i}$ with $1/D_{i,i}$). One can check that definition is consistent with the more general definition of a matrix power; that is so $(D^p)^{1/p} = D$.

Example: Adjacency and Degree Matrices

Again using our example graph, we can define the adjacency A and degree matrix D:

$$A = \begin{pmatrix} 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix} \quad D = \begin{pmatrix} 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}.$$

Taking D to the power $-1/2$ is now

$$D^{-1/2} = \begin{pmatrix} 0.577 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0.707 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.577 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.577 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.577 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0.577 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0.707 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}.$$

10.1 Markov Chains

A *Markov chains* (V, P, q) is defined by a set of nodes V, a probability transition matrix P, and an initial state q. We will see soon how P can be induced from the matrix representation of an edge set, so this structure is closely connected to understanding graphs; for now we describe it in a more general context. Moveover, in many of the most useful and often desirable contexts q is not needed, and the initial part of this discussion will be aimed at showing how to remove the dependence on q.

The point of a Markov chain (V, P, q) is to understand the "movement around" or "flow through" the vertex set V. This movement is governed by the matrix P, but to get a grasp on this, it is important to start with how to think about this, via its initial state q.

The initial state q represents a probability distribution over vertices. For instance, if the mental model of this process is a random walk around the vertices of a graph, then an initial state q (i.e., specifically at a vertex $b \in V$ with probability 1, in our running example) is

$$q^T = [0\ 1\ 0\ 0\ 0\ 0\ 0\ 0].$$

A more general model is when there is not a single vertex which dictates the state, but a distribution over possible vertices. Then if there is a 10% chance of being in state a, a 30% chance of being in state d and a 60% change of being in state f, this represents as

$$q^T = [0.1\ 0\ 0\ 0.3\ 0\ 0.6\ 0\ 0].$$

In general we need to enforce that $q \in \Delta^{|V|}$, that is, it represents a probability distribution, so

- each $q[i] \geq 0$
- $\sum_i q[i] = 1$.

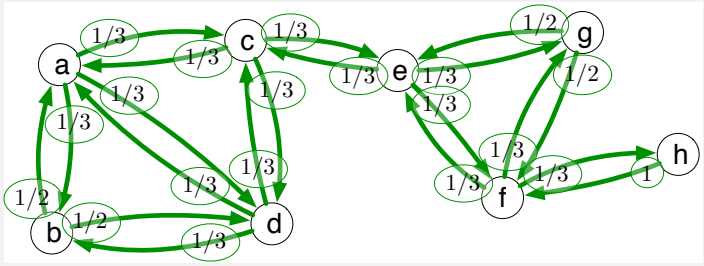
Now the *probability transition matrix* P is a column normalized matrix. That is each column must satisfy a probability distribution; each column $P_j \in \Delta^{|V|}$, so the entrees are non-negative and sum to 1. Each ith column represents the probability that, conditioned on starting at vertex v_i where the next vertex would be. That is, entry $P_{i,i'}$ describes that from vertex v_i the probability is $P_{i,i'}$ that the next vertex is $v_{i'}$. Using an adjacency matrix A, we can derive P by normalizing all of the columns so $P_j = A_j / \|A\|_1$.

Example: Probability Transition Matrix

The running example adjacency matrix A can derive a probability transition matrix P as

$$P = \begin{pmatrix} 0 & 1/2 & 1/3 & 1/3 & 0 & 0 & 0 & 0 \\ 1/3 & 0 & 0 & 1/3 & 0 & 0 & 0 & 0 \\ 1/3 & 0 & 0 & 1/3 & 1/3 & 0 & 0 & 0 \\ 1/3 & 1/2 & 1/3 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1/3 & 0 & 0 & 1/3 & 1/2 & 0 \\ 0 & 0 & 0 & 0 & 1/3 & 0 & 1/2 & 1 \\ 0 & 0 & 0 & 0 & 1/3 & 1/3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1/3 & 0 & 0 \end{pmatrix}.$$

We can also present this as a graph, where directed edges show the probability transition probability from one node to another. The lack of an edge represents 0 probability of a transition. This sort of representation is already very cluttered, and thus is not scalable to much larger graphs.



Note that although the initial graph was undirected, this new graph is directed. For instance edge (a, b) from a to b has probability $1/3$ while edge (b, a) from b to a has probability $1/2$. Or more dramatically, edge (f, h) from f to h has probability $1/3$, where as edge (h, f) from f to h has probability 1; that is, vertex h *always* transitions to f.

Now given a state $q^T = [0\ 1\ 0\ 0\ 0\ 0\ 0\ 0]$ can "transition" to the next state as (using our example P)

$$q_1 = Pq = \begin{bmatrix} \frac{1}{2} & 0 & 0 & \frac{1}{2} & 0 & 0 & 0 & 0 \end{bmatrix}^T.$$

Then we can get to the next state as

$$q_2 = Pq_1 = PPq = P^2q = \begin{bmatrix} \frac{1}{6} & \frac{2}{6} & \frac{2}{6} & \frac{1}{6} & 0 & 0 & 0 & 0 \end{bmatrix}^T.$$

and

$$q_3 = Pq_2 = \begin{bmatrix} \frac{1}{3} & \frac{1}{9} & \frac{1}{9} & \frac{1}{3} & \frac{1}{9} & 0 & 0 & 0 \end{bmatrix}^T.$$

In general we can write $q_n = P^n q$, that is starting with q and "hitting" q on the left n times by P, the transition matrix.

This is called a "Markov" chain after Andrey Markov, because it is a *Markov process*. This naming refers to that this process only depends on its current state, and nothing prior to that (unless it is implicitly encoded in the current state).

There are *two* ways to think about this Markov chain process.

- It describes a *random walk* of a point starting at one vertex, this corresponds with a single 1 coordinate in q. Then at each step it decides where to go next randomly based on the appropriate column of P, with the jth column of P encoding the transtion probability of the jth vertex. It moves to *exactly one* new state. Then repeat.

- It describes the *probability distribution of a random walk*. At each state, we only track the distribution of where it *might* be: this is q_n after n steps. Alternatively, we can consider P^n, then for any initial state q_0, $P^n q_0$ describers the distribution of where q_0 might be after n steps. So entry $P^n_{j,i}$ (jth column, ith row) describes the probability that a point starting in j will be in state i after n steps.

Usually, only one of these two interpretations is considered. They correspond to quite different algorithms and purposes, each with their own advantages. We will discuss both.

10.1.1 Ergodic Markov Chains

A Markov chain is *ergodic* if there exists some t such that for all $n \geq t$, then each entry in P^n is positive. This means that from any starting position, after t steps there is *always* a chance we are in every state. That is, for any q, then $q_n = P^n q$ is positive in all entries. It is important to make the distinction in the definition that it is not that we have some positive entry for *some* $n \geq t$, but for *all* $n \geq t$, as we will see.

When is a Markov chain *not* ergodic? To characterize when a Markov chain is ergodic, it is simpler to rule out the cases when it is not ergodic, and then if it does not satisfy these properties, it must be ergodic. There are three such non-ergodic properties:

- It is *cyclic*. This means that it alternates between different sets of states every 2 or 3 or in general p steps. This strict, even a single low probability event that it deviates from this makes it not cyclic. The cyclic nature does not need to be on the entire graph, it may only be on a disconnected part of the graph.

Example: Cyclic Probability Transition Matrices

Here are some example cyclic transition matrices:

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \qquad \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix} \qquad \begin{pmatrix} 0 & 1/2 & 1/2 & 1/2 & 1/2 & 0 \\ 1/4 & 0 & 0 & 0 & 0 & 1/4 \\ 1/4 & 0 & 0 & 0 & 0 & 1/4 \\ 1/4 & 0 & 0 & 0 & 0 & 1/4 \\ 1/4 & 0 & 0 & 0 & 0 & 1/4 \\ 0 & 1/2 & 1/2 & 1/2 & 1/2 & 0 \end{pmatrix}$$

- It has *absorbing and transient states*. This corresponds to some Markov chains which can separate V into two class $A, T \subset V$ so that if a random walk leaves some node in T and lands in a state in A, then it *never* returns to any state in T. In this case, the nodes A are *absorbing*, and the nodes in T are *transient*. Note that this only happens when the initial graph is *directed*, so the walk cannot go backwards on an edge.

Example: Absorbing and Transient Probability Transition Matrices

Here are some examples:

$$\begin{pmatrix} 1/2 & 0 \\ 1/2 & 1 \end{pmatrix} \qquad \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix} \qquad \begin{pmatrix} 1/2 & 1/2 & 0 & 0 & 0 & 0 \\ 1/2 & 49/100 & 0 & 0 & 0 & 0 \\ 0 & 1/100 & 1/4 & 1/4 & 1/4 & 1/4 \\ 0 & 0 & 1/4 & 1/4 & 1/4 & 1/4 \\ 0 & 0 & 1/4 & 1/4 & 1/4 & 1/4 \\ 0 & 0 & 1/4 & 1/4 & 1/4 & 1/4 \end{pmatrix}$$

- It is not *connected*. This property indicates that there are two sets of notes $A, B \subset V$ such that there is no possible way to transition from any node in A to any node in B.

Example: Disconnected Probability Transition Matrices

And some examples:

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \qquad \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \qquad \begin{pmatrix} 1/2 & 1/2 & 0 & 0 & 0 & 0 \\ 1/2 & 1/2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1/3 & 1/2 & 1/3 & 0 \\ 0 & 0 & 1/3 & 0 & 1/3 & 0 \\ 0 & 0 & 1/3 & 1/2 & 1/3 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

When it *is* ergodic. From now on, we will assume that the Markov chain *is* ergodic. At a couple of critical points we will show simple modifications to existing chains to help ensure this.

Now there is an **amazing** property that happens. Let $P^* = P^n$ as $n \to \infty$; this is well-defined, and it will converge to a limiting matrix P^*. Now let $q_* = P^* q$. That is, there is also a limiting state, and this *does not* depend on the choice of q.

Ergodic Markov Chains: They have a limiting probability transition matrix ($P^* = P^n$ as $n \to \infty$) and a limiting state $q^* = P^* q$, which is the same for *any* initial state q.

This property has are varied and important consequences.

- For all starting states q, the final state is q_* (if we run the chain long enough). So we can analyze such chains, just based on V and P; or if P is derived from an adjacency matrix, this is analyzing properties of the inducing graph.

- As we do a random walk, we will eventually have an expected state precisely described by q_*. So we can analyze the process of such random walks without actually simulating an indefinite number of such processes, as long as we can derive the state q^* and final matrix P^*.

- The final state is stable, specifically $q_* = PP^* q$ thus $q_* = Pq_*$. That is the probability of being in a state i and leaving to j, is the same as being in another state j and arriving at i; this is called the *delicate balance*. Globally, we can generalize this to say, the probability of being in any state i and leaving (to any other state) is the same as being any other state and arriving in i. Thus, if a distribution

starts in $q_0 = q_*$ it is already in the final distribution. And the "further" it starts (e.g. q_0 is more different from q_*), the longer it will takes to converge.

Moreover, q_* is the *first* eigenvector of P, after normalizing so the sum of its elements are 1. In Matlab `[V,L] = eig(P);` and let `v1 = V(:,1);` and `qstar = v1/sum(v1)` to get the stable distribution `qstar`. . This *second* eigenvalue λ_2 determines the rate of convergence. The smaller λ_2, the faster the rate of convergence.

Example: Limiting States

In our example graph,

$$q_* = (0.15, 0.1, 0.15, 0.15, 0.15, 0.15, 0.1, 0.05)^T = (\frac{3}{20}, \frac{1}{10}, \frac{3}{20}, \frac{3}{20}, \frac{3}{20}, \frac{3}{20}, \frac{1}{10}, \frac{1}{20})^T.$$

Note that this distribution is *not uniform* and is also not directly proportional to the transition probabilities. It is a global property about the connectivity of the underlying graph. Vertex h which is the most isolated has the smallest probability, and more central nodes have higher probability.

The second eigenvalue of P is 0.875 which is small enough that the convergence is fast. If the part of the graph containing $\{a, b, c, d\}$ was made harder to transition to or from the other part of the graph, this value could be much larger (e.g., 0.99). On the other hand if another edge was added between say d and f, then this would be much smaller (e.g., 0.5), and the convergence would be even faster.

10.1.2 Metropolis Algorithm

The Metropolis Algorithm, sometimes referred to as Markov Chains Monte Carlo (MCMC) was developed by Metropolis, Rosenbluth, Rosenbluth, Teller, and Teller in 1953 to help develop the atomic bomb. There is some controversy over who really deserves credit for the invention. But, the lesson is, it pays to have a name that is both cool sounding, and earliest in alphabetical order! This has had enormous influence in statistical physics and in computing Bayesian statistics.

Here each state $v \in V$ has a weight associated with it:

$$w(v) \qquad \text{where} \qquad \sum_{v \in V} w(v) = W.$$

More generally, V may be continuous and then $W = \int_{v \in V} w(v)\, dv$. Then we want to land in a state v with probability $w(v)/W$. But...

- V might be *very* large, and W unknown.
- V can be continuous, so there can be *no* way to calculate W. One can think of this as a *probe-only* distribution, since you can measure $\mu(v) = cw(v)$ at any one state at a time where c is some unknown constant (related to W).

Then the goal is to design a special Markov chain so $q_*[v] = w(v)/W$ (without knowing W).

The algorithm sketched in Algorithm 10.1.1, starts with some $v_0 \in V$ so $q = [0\ 0\ 0\ \ldots\ 1\ \ldots\ 0\ 0]^T$. Then iterate as follows. Choose neighbor u (proportional to $K(v, u)$) where K is some notion of neighborhood/similarity (for instance a kernel, like a Gaussian kernel). And move to u with probability $\min\{1, w(u)/w(v)\}$.

This implicitly defines a Markov chain on the state space V. The transition matrix is defined by the algorithm, but is not realized as a matrix. Importantly, if the chain is ergodic, then there exists some t such that $i \geq t$, then $\mathbf{Pr}[v_i = v] = w(v)/W$. This value t is not only a limiting notion, but holds for some finite t

Algorithm 10.1.1 Metropolis on V and w

Initialize $v_0 = [0\ 0\ 0\ \ldots\ 1\ \ldots\ 0\ 0]^T$.
repeat
  Generate $u \sim K(v, \cdot)$
  if $(w(u) \geq w(v_i))$ **then**
    Set $v_{i+1} = u$
  else
    With probability $w(u)/w(v)$ set $v_{i+1} = u$; otherwise set $v_{i+1} = v_i$.
until "converged"
return $V = \{v_1, v_2, \ldots, \}$

(even if V is continuous), through a property called "coupling from the past". However, determining when such a t has occurred analytically placing an upper bound on the value required for t can be challenging.

Often the goal is to create many samples from w, which can then be used as a proxy for the unknown w to estimate various quantities via concentration of measure bounds. The most formal analysis of such algorithms often dictates that it is run for t steps, take *one* sample, then run for another t steps and take *one* additional sample, and so on repeating tk times to get k samples. This of course seems wasteful, but is necessary to strictly ensure samples are independent.

In practice, it is more common to run for $t = 1000$ steps (the "burn in" period), then take the next $k = 5000$ steps as a random sample. This repeats a total of only $t + k$ steps. This second method has "auto-correlation", as samples v_i and v_{i+1} are likely to be "near" each other (either since K is local, or because it did not accept a new state). Officially, we should take only one point every s steps, where s depends on the degree of auto-correlation. But in practice, we take all k samples, but treat them (for purpose of concentration bounds) as k/s samples.

10.2 PageRank

Search engines were revolutionized in the late 1990s when Google was formed, with the PageRank algorithm as the basis for ranking webpages within its search engine. Before PageRank other search engines (e.g., Altavista, Lycos, Infoseek) and indexes (e.g., Yahoo!, LookSmart) were based almost entirely on a combination of the content of the pages and manually curated lists. These aspects are still used as part of an overall search and ranking method, but PageRank added the perspective of also considering the importance of pages based on the global structure of the webgraph.

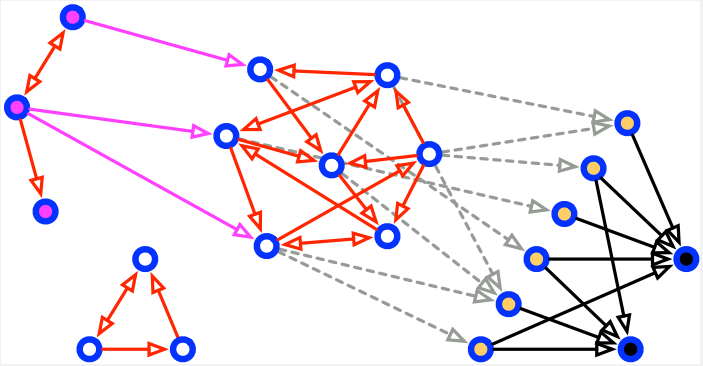
The webgraph is a graph where each vertex is a webpage, and directed edges are created when one webpage links to another one. Search engines implicitly had stored this graphs already, since they ran "crawlers." These were algorithms which randomly followed links from one webpage to another, in this case, with the purpose of cataloging the content of each webpage so it could be put into a large nearest-neighbor search algorithm (often based on cosine similarity using bag-of-words models or Jaccard similarity using k-gram models).

Intuitively, the PageRank model extended the idea of a crawler to be a "random surfer," someone who randomly browses webpages. The goal was to identify webpages which a random surfer would commonly reach, and to mark them as more important in the search engine. This model can be formalized as a Markov chain, and the importance is given by the limiting state distribution q_*.

However, this only works when the graph is ergodic. And the webgraph is very non-ergodic. It is not connected. And there are webpages which are linked to, but do not link to anything else; these are absorbing states. Worse, spammers could (and do!) intentionally create such sets webpages to capture the attention of crawlers and random surfers.

Example: Webgraph

Below is an example directed webgraph. It is disconnected and has absorbing and transient states.



This includes a large part with black and grey edges; this two-layer structure mirrors those used by spammers to attract traffic of random surfers and automated ranking in search engines. The grey edges are planted links to the yellow pages they control. These grey edges might be in comments on blogs, or links on twitter, or any other click bate. Then the yellow pages can be easily updated to direct traffic to the black pages which pay the spammers to get promoted. The effect of this sort of structure can be reduced with PageRank, but is not completely eliminated.

So PageRank adds one additional, but crucial, change to the standard Markov chain analysis of the webgraph: teleportation. Roughly every seven steps (15% of the time), it instructs the random surfer to jump to a completely random node in the webgraph. This makes the graph connected, eliminates absorbing states, increases convergence rates, and jumps out of traps set up by webgraph spammers. That is, it is now ergodic, so the limiting state q_* exists. Moreover, there is an efficient way to implement this without making the webgraph artificially dense, and exploding the memory requirements.

Formalization of the model. Let P be the $n \times n$ probability transition matrix formed by normalizing the webgraph adjacency matrix. Let Q be another probability transition matrix that is a normalized complete graph. That is, each entry is precisely $1/n$. Now the probability transition matrix which represents the PageRank algorithm is

$$R = (1 - \beta)P + \beta Q,$$

where β is the probability of jumping to a random node. As mentioned a typical setting is $\beta = 0.15$.

Now consider a random surfer whose state is q. Then in one iteration, the state is updated as

$$q_1 = Rq.$$

This operation itself is slow to implement since R is dense, since each entry has value at least β/n. However we can use the linearity to decompose this as:

$$q_1 = Rq = (1 - \beta)Pq + \beta Qq = (1 - \beta)Pq + (\beta/n)\mathbf{1}.$$

Here $\mathbf{1}$ represents the length n, all 1s vector. Since Q has identical columns, then it does not matter which column is chosen, and q is eliminated from the second term. And more importantly, the second term is now a constant; it just adds β/n to each coordinate created by the sparse matrix multiply $(1 - \beta)Pq$. The state after i rounds can be computed inductively as

$$q_i = Rq_{i-1} = (1 - \beta)Pq_{i-1} + (\beta/n)\mathbf{1}.$$

The limiting state q_* of R is known as the *PageRank vector*. Each coordinate is associated with a webpage, and provides an "importance" of that page based on the structure of the webgraph. It is one of the factors that goes into the ranking of pages within Google's search engine.

However, we cannot yet just create q_* based on the first eigenvector of R. Because R is dense, this is too slow to do using standard libraries. But, we can adapt the power method, to use the efficient updates in the sparse matrix P. That is the inductively constructed q_i, for i large enough, is a good approximation of q_*.

Moreover, because the graph has a β-fraction of the complete graph, it is well connected, and the second eigenvalue will be large, and it will converge quickly. Also, after a good estimate of the q_* is found, then if the graph is updated, we can used the old estimate to have a good starting point towards the new converged to distribution. Thus, even on a graph as large as the webgraph, only a few (no more than 50, often much fewer) iterations are required to get a good precision on the estimate of q_* of R.

Ethical Questions with TrustRank

It is generally accepted that some webpages are more trustworthy than others. For instance, Wikipedia pages, top-level pages at reputable universities, established newspapers, and government pages. These sources typically have a formal review process and a reputation to uphold. On the other hand, pages which are parts of blogs, comment sections on news articles, and personal social media pages are typically much less trustworthy.

With this in mind, variants on PageRank have been developed with this trust in mind, as a way to help combat advanced spamming attempts. This works by having trusted pages more frequently teleported to, and less likely to be teleported from. This is easy to implement by adjusting the $\mathbf{1}$ vector in $(\beta/n)\mathbf{1}$. Denote the limiting vector of this process the *trustRank* score. It has been suggested that webpages which deviate in their standard PageRank score and their trustRank score are more likely to be spam webpages, and can be down-weighted in the search results.

What are the ethical considerations one should consider when choosing to implement this? And what considerations should be taken in how to build the updated trust weighting?

10.3 Spectral Clustering on Graphs

Spectral clustering is an example of top-down hierarchical clustering. That is, it finds the best way to split the data into two parts, then recurses on both parts until some desired level of resolution has been reached. Within this top-down framework, the key operation of splitting the data is through a mapping to one dimension – a sorted order – and then uses this order to find the best split.

Spectral clustering is most simply defined starting from a graph representation, although we will see who it can use a similarity function $\mathbf{s}$ to construct an equivalent form for any data set. From the graph representation, it defines a special conversion to another matrix called a Laplacian, and uses the eigen-decomposition to define the one-dimension ordering. For the remainder of this section we will focus on interpreting these graph and matrix representations, and how to use them to define the one-dimensional ordering.

So how should we cluster a graph? Typically, on graphs the clustering is a partition of the vertex set. That is a single hard cluster is a subset $S \subset V$. And to perform top-down clustering, the elements to analyze is a subset $S \subset V$ and its compliment $\bar{S} = V \setminus S$.

The edges are then the quantity used to determine how good the clustering is. In general, we want many edges within a cluster, and few edges between clusters. Define the *volume* of a cluster $\mathsf{vol}(S)$ as the number of edges with at least one vertex in V. Also, the *cut* $\mathsf{cut}(S, T)$ between two clusters S, T is defined as the

number of edges with one vertex in S and the other in T. Ultimately a good clustering has large $\mathsf{vol}(S)$ for each cluster and a small $\mathsf{cut}(S,T)$ for each pair of clusters.

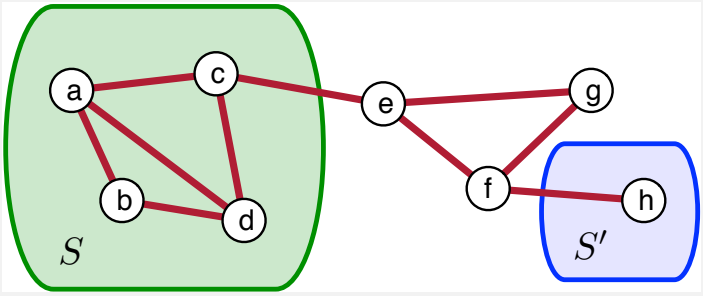
Specifically, the *normalized cut* between S and T is

$$\mathsf{ncut}(S,T) = \frac{\mathsf{cut}(S,T)}{\mathsf{vol}(S)} + \frac{\mathsf{cut}(S,T)}{\mathsf{vol}(T)}.$$

And as a result the key step in top-down graph clustering is to find the cluster S (and compliment $T = V \setminus S$) that has the *minimum* $\mathsf{ncut}(S,T)$. Dividing by $\mathsf{vol}(S)$ and $\mathsf{vol}(T)$ prevents us from finding either S or T that is too small, and the $\mathsf{cut}(S,T)$ in the numerator will ensure a small number of edges are crossing this partition.

Example: Normalized Cut

In the running example graph, clusters $S = \{a,b,c,d\}$ and the singleton cluster with $S' = \{h\}$ both have a small cut value with both $\mathsf{cut}(S', \bar{S}') = 1$ and $\mathsf{cut}(S, \bar{S}) = 1$. The volumes however are very different with $\mathsf{vol}(S) = 6$ and $\mathsf{vol}(\bar{S}) = 5$ while $\mathsf{vol}(S') = 1$ and $\mathsf{vol}(\bar{S}') = 10$.



The difference in volumes shows up in the normalized cut score for S and S'. Specifically $\mathsf{ncut}(S', \bar{S}') = 1 + \frac{1}{10} = 1.1$, whereas $\mathsf{ncut}(S, \bar{S}) = \frac{1}{6} + \frac{1}{5} = 0.367$. Overall S results in the smallest normalized cut score; this aligns with how it intuitively is the partition which best divides the vertices in a balanced way without separating along too many edges.

Affinity matrix. This algorithm will start with the adjacency matrix A of a graph, and then transform it further. However, the adjacency matrix A need not be $0-1$. It can be filled with the *similarity* value defined by some similarity function $\mathbf{s} : X \times X \to [0,1]$ defined between elements of a dataset X; then A stands for *affinity*. The degree matrix is still diagonal but is now defined as the sum of elements in a row (or column — it must be symmetric). The remainder of the spectral clustering formulation and algorithm will be run the same way; however we continue the description the graph representation as it more clean. However, this generalization allows us to apply spectral clustering to point sets with a arbitrarily data set and an appropriate similar measure $\mathbf{s}$.

When the similarity of a pair is very small, it is often a good heuristic to round the values down to 0 in the matrix to allow algorithms to take advantage of fast sparse linear algebraic subroutines.

10.3.1 Laplacians and their Eigen-Structure

The key step in spectral clustering is found by mapping a graph to its Laplacian, and then using the top eigenvector to guide the normalized cut. The term "spectral" refers to the use of eigenvalues. We start by defining the (unnormalized) *Laplacian matrix* of a graph G with n vertices as $L_0 = D - A$.

Example: Laplacian matrix

The *Laplacian matrix* of our example graph is

$$L_0 = D - A = \begin{pmatrix} 3 & -1 & -1 & -1 & 0 & 0 & 0 & 0 \\ -1 & 2 & 0 & -1 & 0 & 0 & 0 & 0 \\ -1 & 0 & 3 & -1 & -1 & 0 & 0 & 0 \\ -1 & -1 & -1 & 3 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 3 & -1 & -1 & 0 \\ 0 & 0 & 0 & 0 & -1 & 3 & -1 & -1 \\ 0 & 0 & 0 & 0 & -1 & -1 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & 1 \end{pmatrix}.$$

Note that the entries in each row and column of L_0 sum up to 0. We can think of D as the *flow* into a vertex, and think of A as the *flow* out of the vertex (related to the Markov chain formulation). This describes a process where the "flow keeps flowing," so it does not get stuck anywhere. That is, as much flows in as flows out of each vertex.

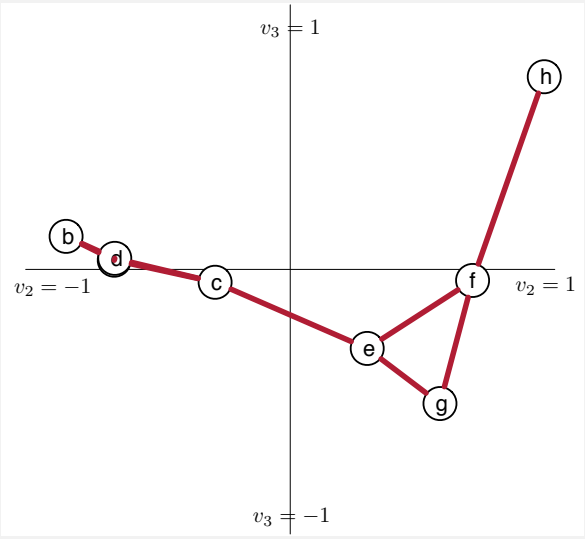
It is now useful to consider the eigen-decomposition of $L_0 = U_0 \Lambda_0 U_0^T$. The first eigenvalue/vector is not useful: the first eigenvalue is always 0 and the first eigenvector has each element the same. However, the second eigenvalue/vector are important descriptors of the graph. In particular, the second eigenvector, known as the *Fielder vector*, can be interpreted as providing a useful 1-dimensional representation of vertices of the graph which preserves connectivity. Taking k eigenvectors provides a k-dimensional representation; however, these are all unit vectors, but should not be treated equally. Rather, the vertices can be given a k-dimensional representation using the corresponding part of the eigenvectors u_j scaled by $1/\sqrt{\lambda_j}$, the jth eigenvalue. For eigenvectors $u_j = (u_{j,1}, u_{j,2}, \ldots, u_{j,n})$, the ith vertex can be represented as $(u_{2,i}/\sqrt{\lambda_2}, u_{3,i}/\sqrt{\lambda_3}, \ldots, u_{k+1,i}/\sqrt{\lambda_{k+1}}) \in \mathbb{R}^k$. This scaling correctly implies that the relative importance between two eigenvectors is governed by the ratio between their eigenvalues.

Example: Laplacian-based Embedding

The following table shows the eigenvalues $\lambda_1, \ldots, \lambda_8$ and eigenvectors $u_1, \ldots, u_8$ of the Laplacian L_0.

λ	0	**0.278**	1.11	2.31	3.46	4	4.82
U	$1/\sqrt{8}$	$-.36$	0.08	0.10	0.28	0.25	$1/\sqrt{2}$
	$1/\sqrt{8}$	$-.42$	0.18	0.64	$-.38$	0.25	0
	$1/\sqrt{8}$	$-.20$	$-.11$	0.61	0.03	$-.25$	0
	$1/\sqrt{8}$	$-.36$	0.08	0.10	0.28	0.25	$-1/\sqrt{2}$
	$1/\sqrt{8}$	0.17	$-.37$	0.21	$-.54$	$-.25$	0
	$1/\sqrt{8}$	0.36	$-.08$	$-.10$	$-.28$	0.75	0
	$1/\sqrt{8}$	0.31	$-.51$	$-.36$	$-.56$	0.56	0
	$1/\sqrt{8}$	0.50	0.73	0.08	0.11	0.11	0

Next we plot the vertices using the implied 2-dimensional representation from $(u_2/\sqrt{\lambda_2})$ and $(u_3/\sqrt{\lambda_3})$. Note that drawing keeps edge-connected vertices nearby. Moreover, points a and d are directly on top of each other. From the perspective of the graph, they are indistinguishable. Indeed, the eigenstructure does not separate them until u_7.



This k-dimensional representation hints at how to perform the cut into two subsets. In fact, the typical strategy is to only use a single eigenvector, u_2. This provides a 1-dimensional representation of the vertices, a sorted order. There are then two common approaches to find the cut.

The first approach is to just select vertices with values less than 0 in S, and those greater or equal to 0 in $\bar{S}$. For large complex graphs, this does not always work as well as the next approach; in particular, there may be two vertices which have values both very close to 0, but one is negative and one is positive. Often, we would like to place these into the same cluster.

The second approach is to consider the cut defined by any threshold in this sorted order. For instance we can define $S_\tau = \{v_i \in V \mid u_{2,i} \leq \tau\}$ for some threshold τ. In particular, it is easy to find the choice of S_τ which maximizes $\mathsf{ncut}(S_\tau, \bar{S}_\tau)$ by updating this score as τ is increased by incrementally adding points to S_τ (and removing from $\bar{S}_\tau$).

Normalized Laplacian. However, to optimize the cut found using this family of approaches to maximize the normalized cut, it is better to use a different form of the Laplacian known as the normalized Laplacian

L of a graph. For a graph G, an identity matrix I, and the graph's diagonal matrix D and adjacency matrix A, its normalized Laplacian is defined as $L = I - D^{-1/2}AD^{-1/2}$.

We can also convert L_0 to the normalized Laplacian L using the $D^{-1/2}$ matrix as

$$L = I - D^{-1/2}AD^{-1/2} = D^{-1/2}L_0D^{-1/2}.$$

The left- and right-multiplication by $D^{-1/2}$ can be thought of as normalizing by the degrees. That is each entry $P_{i,j}$ of $P = D^{-1/2}AD^{-1/2}$ (and edge (v_i, v_j)) is normalized by the amount of flow in and out of the nodes v_i and v_j of its corresponding edge (v_i, v_j).

Example: Normalized Laplacian

The *normalized Laplacian matrix* of our example graph is $L = I - D^{-1/2}AD^{-1/2} =$

$$\begin{pmatrix} 1 & -0.408 & -0.333 & -0.333 & 0 & 0 & 0 & 0 \\ -0.408 & 1 & 0 & -0.408 & 0 & 0 & 0 & 0 \\ -0.333 & 0 & 1 & -0.333 & -0.333 & 0 & 0 & 0 \\ -0.333 & -0.408 & -0.333 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & -0.333 & 0 & 1 & -0.333 & -0.408 & 0 \\ 0 & 0 & 0 & 0 & -0.333 & 1 & -0.408 & -0.577 \\ 0 & 0 & 0 & 0 & -0.408 & -0.408 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & -0.577 & 0 & 1 \end{pmatrix}.$$

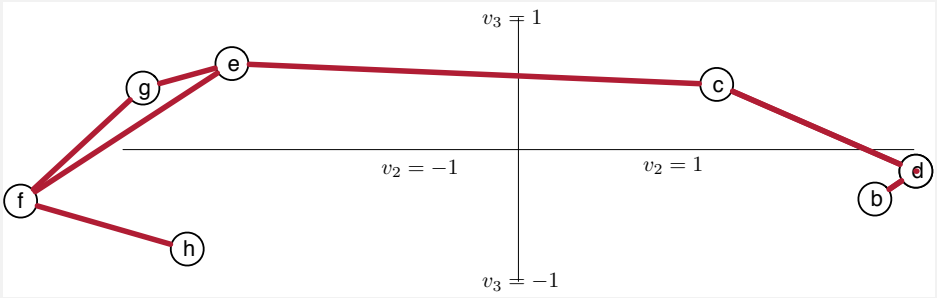
Using that the normalized Laplacian is mechanically the same as using the (unnormalized) Laplacian. The second eigenvector provides a sorted order of the vertices, and the best normalized cut can be found among the subsets according to this sorted order. Then to complete the spectral clustering, this procedure is recursively repeated on each subset until some designed resolution has been reached.

Example: Normalized Laplacian-based Embedding

The following table shows the eigenvalues $\lambda_1, \ldots, \lambda_8$ and eigenvectors $u_1, \ldots, u_8$ of the normalized Laplacian L for our example graph.

λ	0	**0.125**	0.724	1.00	1.33	1.42	1.66	1.73
V	$-.39$	0.38	$-.09$	0.00	0.71	0.26	$-.32$	0.16
	$-.32$	0.36	$-.27$	0.50	0.00	$-.51$	0.38	$-.18$
	$-.39$	0.18	0.36	$-.61$	0.00	0.03	0.47	$-.29$
	$-.39$	0.38	$-.09$	0.00	$-.71$	0.26	$-.32$	0.16
	$-.39$	$-.28$	0.48	0.00	0.00	$-.57$	0.31	0.33
	$-.39$	$-.48$	$-.29$	0.00	0.00	0.05	$-.31$	$-.65$
	$-.31$	$-.36$	0.27	0.50	0.00	0.51	0.38	$-.18$
	$-.22$	$-.32$	$-.61$	$-.35$	0.00	$-.07$	0.27	0.51

Again we can plot the vertices using the implied 2-dimensional representation from $(u_2/\sqrt{\lambda_2})$ and $(u_3/\sqrt{\lambda_3})$. Again the drawing keeps edge-connected vertices nearby and does not distinguish points a and d.



Compared to the plot based on L_0, this one is even more dramatically stretched out along u_2. Also, note that while the suggested cut along u_2 is still at $\tau = 0$, the direction of the orientation is flipped. The vertices $\{a, b, c, d\}$ are now all positive, while the others are negative. The is because the eigen-decomposition is not unique in the choice of signs of the eigenvectors, even if eigenvalues are distinct. Its also worth observing that in the second coordinate defined by u_3, vertex f now has different sign that vertices g and e, because the normalized Laplacian values larger cardinality cuts more than the unnormalized Laplacian.

10.4 Communities in Graphs

Finding relationships and communities from a social network has been a holy grail for the internet data explosion since the late 90s; before Facebook, and even before MySpace. This information is important for targeted advertising, for identifying influential people, and for predicting trends before they happen, or spurring them to make them happen.

At the most general, a social network is a large directed graph $G = (V, E)$. For decades, psychologists and others studied small scale networks (100 friends, seniors in a high school). Anything larger was too hard to collect and work with.

Also mathematicians studied large graphs, famously as properties of random graphs. For instance, the Erdös-Rényi model assumed that each pair of vertices had an edge with probability $p \in (0, 1)$. As p

increased as a function $|V|$, they could study properties of the connectedness of the graph: one large connected component forms, then the entire graph is connected, then cycles and more complex structures appear at greater rates.

Ethical Questions with collecting large social network graphs

There are many large active and influential social networks (Facebook, Twitter, Instagram). These are influential because they are an important source of information and news for their users. But in many cases the full versions of these networks are closely guarded secrets by the companies. These networks are expensive to host and maintain, and most of these hosting companies derive value and profit by applying targeted advertising to the users.

Various example networks are available through inventive and either sanctioned or unsanctioned information retrieval and scraping techniques. Is it ethical to try to scrape public or semi-private posts of users for either academic or entrepreneurial reasons?

Some social media companies attempted to quantify the happiness derived by users based on how they interacted with the social network. This included some approaches by which the study potentially decreased the happiness of the users (e.g., showing only negative posts). Was it useful to test these experiments, and how what could go wrong with this ability?

Example: Why do people join groups?

Before large social networks on the internet it was a painstaking and noisy process to collect networks. And tracking them over time was even more difficult. Now it is possible to study the formation of these structures, and collect this data at large scale on the fly. This allows for more quantitative studies of many questions. As an example anthropological question: *Why do people join groups?*

Consider a group of people C that have tight connections (say in a social network). Consider two people X (Xavier) and Y (Yolonda). Who is more likely to join group C?

- X has three friends in C, all connected.
- Y has three friends in C, none connected.

Before witnessing large social networks, both sides had viable theories with reasonable arguments. Arguments for X were that there was safety and trust in friends who know each other. Arguments for Y were that there was independent support for joining the group; hearing about it from completely different reasons might be more convincing than from a single group of the same size. For static network data, it is probability impossible to distinguish empirically between these cases. But this can be verified by just seeing which scenario is more common as networks form, and users decided to add pre-defined labels indicating a group.

It turns out the answer is: X Vertices tend to form tightly connected subsets of graphs.

10.4.1 Preferential Attachment

How do edges form in an evolving graph? An important concept is called *preferential attachment*: If edge (a, b) and (a, c) exist in a graph, then it is more likely than random for (b, c) to exist. Reasons include

- b and c somehow already trust each other (through a).
- a may have incentive to bring b and c together.

- All edges may result from common phenomenon (e.g. church group).

This has been widely observed in large networks in many settings. Although the exact model which realizes this, and how the probability is adjusted is debated, the general property is accepted as a key principle. Notably, this is *not* compatible with the famous mathematical model of Erdös-Rényi graphs which assumes all edges are equally likely. And so the rich mathematics associated with that model needs to be avoided or re-worked to apply to most typical large networks.

A key consequence of this is that the number of triangles in the graph becomes an important parameter in judging the health of the network as a whole or its components.

10.4.2 Betweenness

One way of defining communities of a graph, is a procedure through an importance labeling on the graphs vertices or edges. For instance, it could be the PageRank importance assigned to vertices. Once an importance has been assigned, these objects may hold a universal quality or connection, and the parts which are connected despite these objects make up communities. Thus, we can remove all nodes or edges weighted above a threshold, and the remaining connected components may defined the true communities of the graph.

A powerful definition of importance on edges which is especially effective in this paradigm is the *betweenness* score. Specifically, the betweenness of an edge (a, b) is defined as

$$\mathsf{betw}(a, b) = \text{fraction of shortest paths that use edge } (a, b).$$

A large score may indicate an edge does *not* represent a central part of a community. If to get between two communities you need to take this edge, its betweenness score will be high. A "facilitator edge", but not a community edge.

Similarly, the *betweenness of a vertex* is the number of shortest paths that go through this vertex.

Calculating $\mathsf{betw}(a, b)$ for all edges can be time consuming. It typically requires for each vertex, computing shortest paths to all other vertices (the all pairs shortest path problem). Then for each vertex, its effect on every edges can be calculated bt running a careful dynamic programming on the DAG (directed acyclic graph) defined by its shortest path.

10.4.3 Modularity

Communities can be defined and formed without providing an entire partition of the vertices, as in a clustering. The common alternative approach is to define a score on each potential community $C \subset V$, and then search for subsets with large score. The most common score for this approach is called *modularity* and, at a high level it is

$$Q(C) = (\text{fraction of edges in group}) - (\text{expected fraction of edges in group}).$$

So the higher the more tightly packed the community is.

More precisely, we use the adjacency matrix A to denote existing edges, where $A_{i,j}$ as 1 if edge (i, j) exists, and 0 otherwise. The expected value of an edge for two nodes i and j with degree d_i and d_j, respectively, is $E_{i,j} = d_i d_j / 2|E|$. Note this allows self edges. Now formally,

$$Q(\mathcal{C}) = \frac{1}{4|E|} \left[\sum_{C \in \mathcal{C}} \sum_{i,j \in C} (A_{i,j} - E_{i,j}) \right].$$

This score can obtain values $Q(\mathcal{C}) \in [-1/2, 1]$. If the number of edges exceeds the expectation defined this way, then it is positive. Typically when $Q \in [0.3, 0.7]$ this is would be deemed a significant group.

Finding high modularity subsets can be a challenging search problem. Spectral clustering can efficiently provide good initial guesses – although the goal of this approach was to not rely on such methods. Once an initial guess is formed, or starting from a single vertex community C, then they can be incrementally updated to look for high modularity communities. Either, one can add the one vertex at a time that most increases the score, or all vertices which individually increase the score. Alternatively, a random walk, similar to the metropolis algorithm can be used to explore the space of communities.

Exercises

Q10.1: Consider the following probability transition matrix

$$M = \begin{bmatrix} .2 & 0.5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ .8 & 0.2 & 0 & 0 & 0 & 0 & 0 & 0.1 & 0 & 0 \\ 0 & 0.3 & 0 & 0.3 & 0.2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.5 & 0 & 0.2 & 0.5 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.5 & 0.3 & 0 & 0.5 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.4 & 0.4 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.2 & 0 & 0 & 0.5 & 0.4 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0.8 & 0 & 0 & 0.3 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0.2 & 0 & 0 & 0.7 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.4 & 0.6 & 0 \end{bmatrix}$$

We will consider four ways to find $q_* = M^t q_0$ as $t \to \infty$.

- (Matrix Power:) Choose some large enough value t, and create M^t. Then apply $q_* = (M^t)q_0$. There are two ways to create M^t, first we can just let $M^{i+1} = M^i * M$, repeating this process $t-1$ times. Alternatively, (for simplicity assume t is a power of 2), then in $\log_2 t$ steps create $M^{2i} = M^i * M^i$.
- (State Propagation:) Iterate $q_{i+1} = M * q_i$ for some large enough number t iterations.
- (Random Walk:) Starting with a fixed state $q_0 = [0, 0, \ldots, 1, \ldots, 0, 0]^T$ where there is only a 1 at the ith entry, and then transition to a new state with only a 1 in the jth entry by choosing a new location proportional to the values in the ith column of M. Iterate this some large number t_0 of steps to get state q_0'. (This is the *burn in period*.)

  Now make t new step starting at q_0' and record the location after each step. Keep track of how many times you have recorded each location and estimate q_* as the normalized version (recall $\|q_*\|_1 = 1$) of the vector of these counts.
- (Eigen-Analysis:) Compute `eig(M)` and take the first eigenvector after it has been normalized.

1. Run each method (with $t = 1024$, $q_0 = [1, 0, 0, \ldots, 0]^T$ and $t_0 = 100$ when needed) and report the answers.
2. Rerun the Matrix Power and State Propagation techniques with $q_0 = [0.1, 0.1, \ldots, 0.1]^T$. For what value of t is required to get as close to the true answer as the older initial state?
3. Explain at least one **Pro** and one **Con** of each approach. The **Pro** should explain a situation when it is the best option to use. The **Con** should explain why another approach may be better for some situation.
4. Is the Markov chain *ergodic*? Explain why or why not.
5. Each matrix M row and column represents a node of the graph, label these from 1 to 10 starting from the top and from the left. What nodes can be reached from node 4 in one step, and with what probabilities?