Interpretable Machine Learning

A Guide for Making Black Box Models Explainable

@ChristophMolnar

Interpretable Machine Learning

A Guide for Making Black Box Models Explainable

Christoph Molnar

This book is for sale at http://leanpub.com/interpretable-machine-learning

This version was published on 2019-02-21

Contents

Preface

Machine learning has great potential for improving products, processes and research. But **computers usually do not explain their predictions** which is a barrier to the adoption of machine learning. This book is about making machine learning models and their decisions interpretable.

After exploring the concepts of interpretability, you will learn about simple, **interpretable models** such as decision trees, decision rules and linear regression. Later chapters focus on general model-agnostic methods for **interpreting black box models** like feature importance and accumulated local effects and explaining individual predictions with Shapley values and LIME.

All interpretation methods are explained in depth and discussed critically. How do they work under the hood? What are their strengths and weaknesses? How can their outputs be interpreted? This book will enable you to select and correctly apply the interpretation method that is most suitable for your machine learning project.

The book focuses on machine learning models for tabular data (also called relational or structured data) and less on computer vision and natural language processing tasks. Reading the book is recommended for machine learning practitioners, data scientists, statisticians, and anyone else interested in making machine learning models interpretable.

About me: My name is Christoph Molnar, I'm a statistician and a machine learner. My goal is to make machine learning interpretable. If you are interested in improving the interpretability of your machine learning models, do not hesitate to contact me!

Mail: christoph.molnar.ai@gmail.com

Website: https://christophm.github.io/[1]

Follow me on Twitter! @ChristophMolnar[2]

Cover by @YvonneDoinel[3]

[1] https://christophm.github.io/
[2] https://twitter.com/ChristophMolnar
[3] https://twitter.com/YvonneDoinel

Introduction

This book explains to you how to make (supervised) machine learning models interpretable. The chapters contain some mathematical formulas, but you should be able to understand the ideas behind the methods even without the formulas. This book is not for people trying to learn machine learning from scratch. If you are new to machine learning, there are a lot of books and other resources to learn the basics. I recommend the book "The Elements of Statistical Learning" by Hastie, Tibshirani, and Friedman (2009) [4] and Andrew Ng's "Machine Learning" online course[5] on the online learning platform coursera.com to start with machine learning. Both the book and the course are available free of charge!

New methods for the interpretation of machine learning models are published at breakneck speed. To keep up with everything that is published would be madness and simply impossible. That is why you will not find the most novel and fancy methods in this book, but established methods and basic concepts of machine learning interpretability. These basics prepare you for making machine learning models interpretable. Internalizing the basic concepts also empowers you to better understand and evaluate any new paper on interpretability published on arxiv.org[6] in the last 5 minutes since you began reading this book (I might be exaggerating the publication rate).

This book starts with some (dystopian) short stories that are not needed to understand the book, but hopefully will entertain and make you think. Then the book explores the concepts of machine learning interpretability. We will discuss when interpretability is important and what different types of explanations there are. Terms used throughout the book can be looked up in the Terminology chapter. Most of the models and methods explained are presented using real data examples which are described in the Data chapter. One way to make machine learning interpretable is to use interpretable models, such as linear models or decision trees. The other option is the use of model-agnostic interpretation tools that can be applied to any supervised machine learning model. The Model-Agnostic Methods chapter deals with methods such as partial dependence plots and permutation feature importance. Model-agnostic methods work by changing the input of the machine learning model and measuring changes in the prediction output. Model-agnostic methods that return data instances as explanations are discussed in the chapter Example Based Explanations. All model-agnostic methods can be further differentiated based on whether they explain global model behavior across all data instances or individual predictions. The following methods explain the overall behavior of the model: Partial Dependence Plots, Accumulated Local Effects, Feature Interaction, Feature Importance, Global Surrogate Models and Prototypes and Criticisms. To explain individual predictions we have Local Surrogate Models, Shapley Value Explanations, Counterfactual Explanations (and closely related: Adversarial Examples). Some methods can be used to explain both

[4] Friedman, Jerome, Trevor Hastie, and Robert Tibshirani. "The elements of statistical learning". www.web.stanford.edu/~hastie/ElemStatLearn/ (2009).

[5] https://www.coursera.org/learn/machine-learning

[6] https://arxiv.org/

aspects of global model behavior and individual predictions: Individual Conditional Expectation and Influential Instances.

The book ends with an optimistic outlook on what the future of interpretable machine learning might look like.

You can either read the book from beginning to end or jump directly to the methods that interest you.

I hope you will enjoy the read!

Story Time

We will start with some short stories. Each story is an admittedly exaggerated call for interpretable machine learning. If you are in a hurry, you can skip the stories. If you want to be entertained and (de-)motivated, read on!

The format is inspired by Jack Clark's Tech Tales in his Import AI Newsletter[7]. If you like this kind of stories or if you are interested in AI, I recommend that you sign up.

Lightning Never Strikes Twice

2030: A medical lab in Switzerland

"It's definitely not the worst way to die!" Tom summarised, trying to find something positive in the tragedy. He removed the pump from the intravenous pole.
"He just died for the wrong reasons," Lena added.
"And certainly with the wrong morphine pump! Just creating more work for us!" Tom complained while unscrewing the back plate of the pump. After removing all the screws, he lifted the plate and put it aside. He plugged a cable into the diagnostic port.
"You didn't just complain about having a job, did you?" Lena gave him a mocking smile.
"Of course not. Never!" he exclaimed with a sarcastic undertone.

He booted the pump's computer.
Lena plugged the other end of the cable into her tablet. "All right, diagnostics are running," she announced. "I am really curious about what went wrong."

[7] https://jack-clark.net/

"It certainly shot our John Doe into Nirvana. That high concentration of this morphine stuff. Man. I mean … that's a first, right? Normally a broken pump gives off too little of the sweet stuff or nothing at all. But never, you know, like that crazy shot," Tom explained.
"I know. You don't have to convince me … Hey, look at that." Lena held up her tablet. "Do you see this peak here? That's the potency of the painkillers mix. Look! This line shows the reference level. The poor guy had a mixture of painkillers in his blood system that could kill him 17 times over. Injected by our pump here. And here …" she swiped, "here you can see the moment of the patient's demise."
"So, any idea what happened, boss?" Tom asked his supervisor.
"Hm … The sensors seem to be fine. Heart rate, oxygen levels, glucose, … The data were collected as expected. Some missing values in the blood oxygen data, but that's not unusual. Look here. The sensors have also detected the patient's slowing heart rate and extremely low cortisol levels caused by the morphine derivate and other pain blocking agents." She continued to swipe through the diagnostics report.
Tom stared captivated at the screen. It was his first investigation of a real device failure.

"Ok, here is our first piece of the puzzle. The system failed to send a warning to the hospital's communication channel. The warning was triggered, but rejected at protocol level. It could be our fault, but it could also be the fault of the hospital. Please send the logs over to the IT team," Lena told Tom.
Tom nodded with his eyes still fixed on the screen.
Lena continued: "It's odd. The warning should also have caused the pump to shut down. But it obviously failed to do so. That must be a bug. Something the quality team missed. Something really bad. Maybe it's related to the protocol issue."
"So, the emergency system of the pump somehow broke down, but why did the pump go full bananas and inject so much painkiller into John Doe?" Tom wondered.
"Good question. You are right. Protocol emergency failure aside, the pump shouldn't have administered that amount of medication at all. The algorithm should have stopped much earlier on its own, given the low level of cortisol and other warning signs," Lena explained.
"Maybe some bad luck, like a one in a million thing, like being hit by a lightning?" Tom asked her.
"No, Tom. If you had read the documentation I sent you, you would have known that the pump was first trained in animal experiments, then later on humans, to learn to inject the perfect amount of painkillers based on the sensory input. The algorithm of the pump might be opaque and complex, but it's not random. That means that in the same situation the pump would behave exactly the same way again. Our patient would die again. A combination or undesired interaction of the sensory inputs must have triggered the erroneous behavior of the pump. That is why we have to dig deeper and find out what happened here," Lena explained.

"I see …," Tom replied, lost in thought. "Wasn't the patient going to die soon anyway? Because of cancer or something?"
Lena nodded while she read the analysis report.
Tom got up and went to the window. He looked outside, his eyes fixed on a point in the distance. "Maybe the machine did him a favor, you know, in freeing him from the pain. No more suffering. Maybe it just did the right thing. Like a lightning, but, you know, a good one. I mean like the lottery,

but not random. But for a reason. If I were the pump, I would have done the same."
She finally lifted her head and looked at him.
He kept looking at something outside.
Both were silent for a few moments.
Lena lowered her head again and continued the analysis. "No, Tom. It's a bug… Just a damn bug."

Trust Fall

2050: A subway station in Singapore

She rushed to the Bishan subway station. With her thoughts she was already at work. The tests for the new neural architecture should be completed by now. She led the redesign of the government's "Tax Affinity Prediction System for Individual Entities", which predicts whether a person will hide money from the tax office. Her team has come up with an elegant piece of engineering. If successful, the system would not only serve the tax office, but also feed into other systems such as the counter-terrorism alarm system and the commercial registry. One day, the government could even integrate the predictions into the Civic Trust Score. The Civic Trust Score estimates how trustworthy a person is. The estimate affects every part of your daily life, such as getting a loan or how long you have to wait for a new passport. As she descended the escalator, she imagined how an integration of her team's system into the Civic Trust Score System might look like.

She routinely wiped her hand over the RFID reader without reducing her walking speed. Her mind was occupied, but a dissonance of sensory expectations and reality rang alarm bells in her brain.

Too late.

Nose first she ran into the subway entrance gate and fell with her butt first to the ground. The door was supposed to open, … but it did not. Dumbfounded, she stood up and looked at the screen next

to the gate. “Please try another time,” suggested a friendly looking smiley on the screen. A person passed by and, ignoring her, wiped his hand over the reader. The door opened and he went through. The door closed again. She wiped her nose. It hurt, but at least it did not bleed. She tried to open the door, but was rejected again. It was strange. Maybe her public transport account did not have sufficient tokens. She looked at her smartwatch to check the account balance.

“Login denied. Please contact your Citizens Advice Bureau!” her watch informed her.

A feeling of nausea hit her like a fist to the stomach. She suspected what had happened. To confirm her theory, she started the mobile game “Sniper Guild”, an ego shooter. The app was directly closed again automatically, which confirmed her theory. She became dizzy and sat down on the floor again.

There was only one possible explanation: Her Civic Trust Score had dropped. Substantially. A small drop meant minor inconveniences, such as not getting first class flights or having to wait a little longer for official documents. A low trust score was rare and meant that you were classified as a threat to society. One measure in dealing with these people was to keep them away from public places such as the subway. The government restricted the financial transactions of subjects with low Civic Trust Scores. They also began to actively monitor your behavior on social media and even went as far as to restrict certain content, such as violent games. It became exponentially more difficult to increase your Civic Trust Score the lower it was. People with a very low score usually never recovered.

She could not think of any reason why her score should have fallen. The score was based on machine learning. The Civic Trust Score System worked like a well-oiled engine that ran society. The performance of the Trust Score System was always closely monitored. Machine learning had become much better since the beginning of the century. It had become so efficient that decisions made by the Trust Score System could no longer be disputed. An infallible system.

She laughed in despair. Infallible system. If only. The system has rarely failed. But it failed. She must be one of those special cases; an error of the system; from now on an outcast. Nobody dared to question the system. It was too integrated into the government, into society itself, to be questioned. In the few remaining democratic countries it was forbidden to form anti-democratic movements, not because they where inherently malicious, but because they would destabilize the current system. The same logic applied to the now more common algocraties. Critique in the algorithms was forbidden because of the danger to the status quo.

Algorithmic trust was the fabric of the social order. For the common good, rare false trust scorings were tacitly accepted. Hundreds of other prediction systems and databases fed into the score, making it impossible to know what caused the drop in her score. She felt like a big dark hole was opening in and under her. With horror she looked into the void.

Her tax affinity system was eventually integrated into the Civic Trust Score System, but she never got to know it.

Fermi’s Paperclips

Year 612 AMS (after Mars settlement): A museum on Mars

“History is boring,” Xola whispered to her friend. Xola, a blue-haired girl, was lazily chasing one of the projector drones humming in the room with her left hand. “History is important,” the teacher said with an upset voice, looking at the girls. Xola blushed. She did not expect her teacher to overhear her.

“Xola, what did you just learn?” the teacher asked her. “That the ancient people used up all resources from Earther Planet and then died?” she asked carefully. “No. They made the climate hot and it wasn’t people, it was computers and machines. And it’s Planet Earth, not Earther Planet,” added another girl named Lin. Xola nodded in agreement. With a touch of pride, the teacher smiled and nodded. “You are both right. Do you know why it happened?” “Because people were short-sighted and greedy?” Xola asked. “People could not stop their machines!” Lin blurted out.

“Again, you are both right,” the teacher decided, “but it’s much more complicated than that. Most people at the time were not aware of what was happening. Some saw the drastic changes, but could not reverse them. The most famous piece from this period is a poem by an anonymous author. It best captures what happened at that time. Listen carefully!”

The teacher started the poem. A dozen of the small drones repositioned themselves in front of the children and began to project the video directly into their eyes. It showed a person in a suit standing in a forest with only tree stumps left. He began to talk:

The machines compute; the machines predict.

We march on as we are part of it.

We chase an optimum as trained.

The optimum is one-dimensional, local and unconstrained.

Silicon and flesh, chasing exponentiality.

Growth is our mentality.

When all rewards are collected,

and side-effects neglected;

When all the coins are mined,

and nature has fallen behind;

We will be in trouble,

After all, exponential growth is a bubble.

The tragedy of the commons unfolding,

Exploding,

Before our eyes.

Cold calculations and icy greed,

Fill the earth with heat.

Everything is dying,

And we are complying.

Like horses with blinders we race the race of our own creation,

Towards the Great Filter of civilization.

And so we march on relentlessly.

As we are part of the machine.

Embracing entropy.

"A dark memory," the teacher said to break the silence in the room. "It will be uploaded to your library. Your homework is to memorise it until next week." Xola sighed. She managed to catch one of the little drones. The drone was warm from the CPU and the engines. Xola liked how it warmed her hands.

What Is Machine Learning?

Machine learning is a set of methods that computers use to make and improve predictions or behaviors based on data.

For example, to predict the value of a house, the computer would learn patterns from past house sales. The book focuses on supervised machine learning, which covers all prediction problems where we have a dataset for which we already know the outcome of interest (e.g. past house prices) and want to learn to predict the outcome for new data. Excluded from supervised learning are for example clustering tasks (= unsupervised learning) where we do not have a specific outcome of interest, but want to find clusters of data points. Also excluded are things like reinforcement learning, where an agent learns to optimize a certain reward by acting in an environment (e.g. a computer playing Tetris). The goal of supervised learning is to learn a predictive model that maps features of the data (e.g. house size, location, floor type, ...) to an output (e.g. house price). If the output is categorical, the task is called classification, and if it is numerical, it is called regression. The machine learning algorithm learns a model by estimating parameters (like weights) or learning structures (like trees). The algorithm is guided by a score or loss function that is minimized. In the house value example, the machine minimizes the difference between the estimated house price and the predicted price. A fully trained machine learning model can then be used to make predictions for new instances.

Estimation of house prices, product recommendations, street sign detection, credit default prediction and fraud detection: All these examples have in common that they can be solved by machine learning. The tasks are different, but the approach is the same:
Step 1: Data collection. The more, the better. The data must contain the outcome you want to predict and additional information from which to make the prediction. For a street sign detector ("Is there a street sign in the image?"), you would collect street images and label whether a street sign is visible or not. For a credit default predictor, you need past data on actual loans, information on whether the customers were in default with their loans, and data that will help you make predictions, such as income, past credit defaults, and so on. For an automatic house value estimator program, you could collect data from past house sales and information about the real estate such as size, location, and so on.
Step 2: Enter this information into a machine learning algorithm that generates a sign detector model, a credit rating model or a house value estimator.
Step 3: Use model with new data. Integrate the model into a product or process, such as a self-driving car, a credit application process or a real estate marketplace website.

Machines surpass humans in many tasks, such as playing chess (or more recently Go) or predicting the weather. Even if the machine is as good as a human or a bit worse at a task, there remain great advantages in terms of speed, reproducibility and scaling. A once implemented machine learning model can complete a task much faster than humans, reliably delivers consistent results and can be copied infinitely. Replicating a machine learning model on another machine is fast and cheap. The training of a human for a task can take decades (especially when they are young) and is very costly. A major disadvantage of using machine learning is that insights about the data and the task the machine solves is hidden in increasingly complex models. You need millions of numbers to

describe a deep neural network, and there is no way to understand the model in its entirety. Other models, such as the random forest, consist of hundreds of decision trees that "vote" for predictions. To understand how the decision was made, you would have to look into the votes and structures of each of the hundreds of trees. That just does not work no matter how clever you are or how good your working memory is. The best performing models are often blends of several models (also called ensembles) that cannot be interpreted, even if each single model could be interpreted. If you focus only on performance, you will automatically get more and more opaque models. Just take a look at interviews with winners on the kaggle.com machine learning competition platform[8]: The winning models were mostly ensembles of models or very complex models such as boosted trees or deep neural networks.

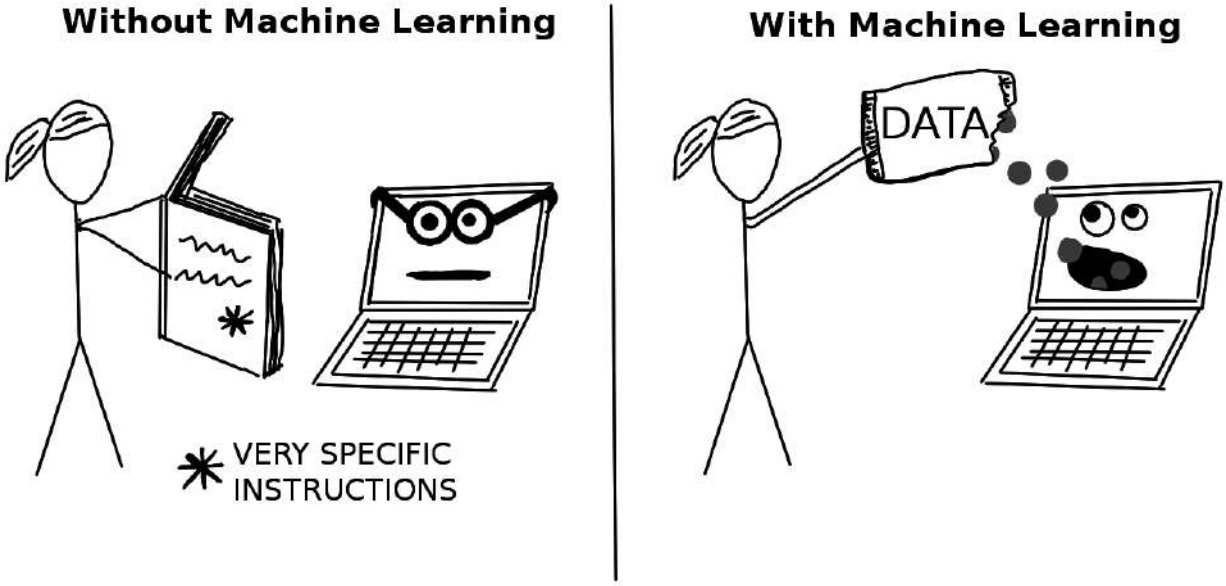
[8] http://blog.kaggle.com/

Terminology

To avoid confusion due to ambiguity, here are some definitions of terms used in this book:

An **Algorithm** is a set of rules that a machine follows to achieve a particular goal[9]. An algorithm can be considered as a recipe that defines the inputs, the output and all the steps needed to get from the inputs to the output. Cooking recipes are algorithms where the ingredients are the inputs, the cooked food is the output, and the preparation and cooking steps are the algorithm instructions.

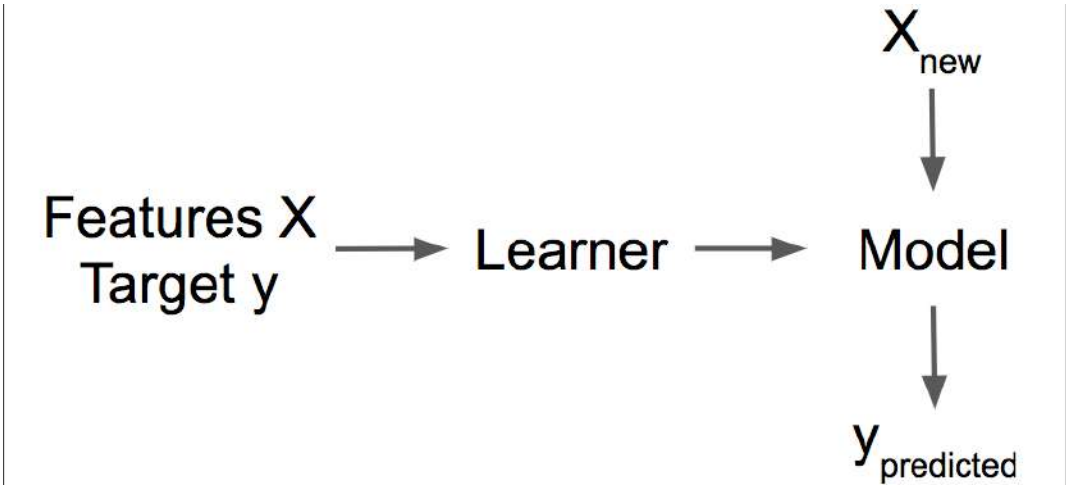
Machine Learning is a set of methods that allow computers to learn from data to make and improve predictions (for example cancer, weekly sales, credit default). Machine learning is a paradigm shift from "normal programming" where all instructions must be explicitly given to the computer to "indirect programming" that takes place through providing data.



A **Learner** or **Machine Learning Algorithm** is the program used to learn a machine learning model from data. Another name is "inducer" (e.g. "tree inducer").

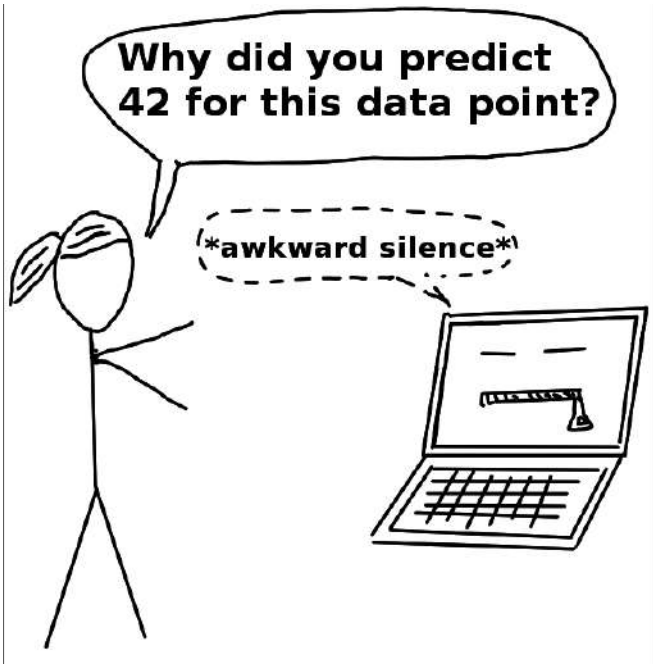
A **Machine Learning Model** is the learned program that maps inputs to predictions. This can be a set of weights for a linear model or for a neural network. Other names for the rather unspecific word "model" are "predictor" or - depending on the task - "classifier" or "regression model". In formulas, the trained machine learning model is called $\hat{f}$ or $\hat{f}(x)$.

[9]"Definition of Algorithm." https://www.merriam-webster.com/dictionary/algorithm. (2017).



A learner learns a model from labeled training data. The model is used to make predictions.

A **Black Box Model** is a system that does not reveal its internal mechanisms. In machine learning, "black box" describes models that cannot be understood by looking at their parameters (e.g. a neural network). The opposite of a black box is sometimes referred to as **White Box**, and is referred to in this book as interpretable model. Model-agnostic methods for interpretability treat machine learning models as black boxes, even if they are not.



Interpretable Machine Learning refers to methods and models that make the behavior and predictions of machine learning systems understandable to humans.

A **Dataset** is a table with the data from which the machine learns. The dataset contains the features and the target to predict. When used to induce a model, the dataset is called training data.

An **Instance** is a row in the dataset. Other names for 'instance' are: (data) point, example, observation. An instance consists of the feature values $x^{(i)}$ and, if known, the target outcome y_i.

The **Features** are the inputs used for prediction or classification. A feature is a column in the dataset. Throughout the book, features are assumed to be interpretable, meaning it is easy to understand what they mean, like the temperature on a given day or the height of a person. The interpretability of the features is a big assumption. But if it is hard to understand the input features, it is even harder to understand what the model does. The matrix with all features is called X and $x^{(i)}$ for a single instance. The vector of a single feature for all instances is x_j and the value for the feature j and instance i is $x_j^{(i)}$.

The **Target** is the information the machine learns to predict. In mathematical formulas, the target is usually called y or y_i for a single instance.

A **Machine Learning Task** is the combination of a dataset with features and a target. Depending on the type of the target, the task can be for example classification, regression, survival analysis, clustering, or outlier detection.

The **Prediction** is what the machine learning model "guesses" what the target value should be based on the given features. In this book, the model prediction is denoted by $\hat{f}(x^{(i)})$ or $\hat{y}$.

Interpretability

There is no mathematical definition of interpretability. A (non-mathematical) definition I like by Miller (2017)[10] is: **Interpretability is the degree to which a human can understand the cause of a decision**. Another one is: **Interpretability is the degree to which a human can consistently predict the model's result** [11]. The higher the interpretability of a machine learning model, the easier it is for someone to comprehend why certain decisions or predictions have been made. A model is better interpretable than another model if its decisions are easier for a human to comprehend than decisions from the other model. I will use both the terms interpretable and explainable interchangeably. Like Miller (2017), I think it makes sense to distinguish between the terms interpretability/explainability and explanation. I will use "explanation" for explanations of individual predictions. See the section about explanations to learn what we humans see as a good explanation.

Importance of Interpretability

If a machine learning model performs well, **why do not we just trust the model** and ignore **why** it made a certain decision? "The problem is that a single metric, such as classification accuracy, is an incomplete description of most real-world tasks." (Doshi-Velez and Kim 2017 [12])

Let us dive deeper into the reasons why interpretability is so important. When it comes to predictive modeling, you have to make a trade-off: Do you just want to know **what** is predicted? For example, the probability that a customer will churn or how effective some drug will be for a patient. Or do you want to know **why** the prediction was made and possibly pay for the interpretability with a drop in predictive performance? In some cases, you do not care why a decision was made, it is enough to know that the predictive performance on a test dataset was good. But in other cases, knowing the 'why' can help you learn more about the problem, the data and the reason why a model might fail. Some models may not require explanations because they are used in a low-risk environment, meaning a mistake will not have serious consequences, (e.g. a movie recommender system) or the method has already been extensively studied and evaluated (e.g. optical character recognition). The need for interpretability arises from an incompleteness in problem formalization (Doshi-Velez and Kim 2017), which means that for certain problems or tasks it is not enough to get the prediction (the **what**). The model must also explain how it came to the prediction (the **why**), because a correct prediction only partially solves your original problem. The following reasons drive the demand for interpretability and explanations (Doshi-Velez and Kim 2017 and Miller 2017).

[10]Miller, Tim. "Explanation in artificial intelligence: Insights from the social sciences." arXiv Preprint arXiv:1706.07269. (2017).

[11]Kim, Been, Rajiv Khanna, and Oluwasanmi O. Koyejo. "Examples are not enough, learn to criticize! Criticism for interpretability." Advances in Neural Information Processing Systems (2016).

[12]Doshi-Velez, Finale, and Been Kim. "Towards a rigorous science of interpretable machine learning," no. Ml: 1–13. http://arxiv.org/abs/1702.08608 (2017).

Human curiosity and learning: Humans have a mental model of their environment that is updated when something unexpected happens. This update is performed by finding an explanation for the unexpected event. For example, a human feels unexpectedly sick and asks, "Why do I feel so sick?". He learns that he gets sick every time he eats those red berries. He updates his mental model and decides that the berries caused the sickness and should therefore be avoided. When opaque machine learning models are used in research, scientific findings remain completely hidden if the model only gives predictions without explanations. To facilitate learning and satisfy curiosity as to why certain predictions or behaviors are created by machines, interpretability and explanations are crucial. Of course, humans do not need explanations for everything that happens. For most people it is okay that they do not understand how a computer works. Unexpected events makes us curious. For example: Why is my computer shutting down unexpectedly?

Closely related to learning is the human desire to **find meaning in the world**. We want to harmonize contradictions or inconsistencies between elements of our knowledge structures. "Why did my dog bite me even though it has never done so before?" a human might ask. There is a contradiction between the knowledge of the dog's past behavior and the newly made, unpleasant experience of the bite. The vet's explanation reconciles the dog owner's contradiction: "The dog was under stress and bit." The more a machine's decision affects a person's life, the more important it is for the machine to explain its behavior. If a machine learning model rejects a loan application, this may be completely unexpected for the applicants. They can only reconcile this inconsistency between expectation and reality with some kind of explanation. The explanations do not actually have to fully explain the situation, but should address a main cause. Another example is algorithmic product recommendation. Personally, I always think about why certain products or movies have been algorithmically recommended to me. Often it is quite clear: Advertising follows me on the Internet because I recently bought a washing machine, and I know that in the next days I will be followed by advertisements for washing machines. Yes, it makes sense to suggest gloves if I already have a winter hat in my shopping cart. The algorithm recommends this movie, because users who liked other movies I liked also enjoyed the recommended movie. Increasingly, Internet companies are adding explanations to their recommendations. A good example is the Amazon product recommendation, which is based on frequently purchased product combinations:

Recommended products when buying some paint from [Amazon](https://www.amazon.com/Colore-Acrylic-Paint-Set-12/dp/B014UMGA5W/). Visited on December 5th 2012.

In many scientific disciplines there is a change from qualitative to quantitative methods (e.g. sociology, psychology), and also towards machine learning (biology, genomics). The **goal of science** is to gain knowledge, but many problems are solved with big datasets and black box machine learning models. The model itself becomes the source of knowledge instead of the data. Interpretability makes it possible to extract this additional knowledge captured by the model.

Machine learning models take on real-world tasks that require **safety measures** and testing. Imagine a self-driving car automatically detects cyclists based on a deep learning system. You want to be 100% sure that the abstraction the system has learned is error-free, because running over cyclists is quite bad. An explanation might reveal that the most important learned feature is to recognize the two wheels of a bicycle, and this explanation helps you think about edge cases like bicycles with side bags that partially cover the wheels.

By default, machine learning models pick up biases from the training data. This can turn your machine learning models into racists that discriminate against protected groups. Interpretability is a useful debugging tool for **detecting bias** in machine learning models. It might happen that the machine learning model you have trained for automatic approval or rejection of credit applications discriminates against a minority. Your main goal is to grant loans only to people who will eventually repay them. The incompleteness of the problem formulation in this case lies in the fact that you not only want to minimize loan defaults, but are also obliged not to discriminate on the basis of certain demographics. This is an additional constraint that is part of your problem formulation (granting loans in a low-risk and compliant way) that is not covered by the loss function the machine learning model was optimized for.

The process of integrating machines and algorithms into our daily lives requires interpretability to increase **social acceptance**. People attribute beliefs, desires, intentions and so on to objects. In a famous experiment, Heider and Simmel (1944) [13] showed participants videos of shapes in which a

[13]Heider, Fritz, and Marianne Simmel. "An experimental study of apparent behavior." The American Journal of Psychology 57 (2). JSTOR: 243–59. (1944).

circle opened a "door" to enter a "room" (which was simply a rectangle). The participants described the actions of the shapes as they would describe the actions of a human agent, assigning intentions and even emotions and personality traits to the shapes. Robots are a good example, like my vacuum cleaner, which I named "Doge". If Doge gets stuck, I think: "Doge wants to keep cleaning, but asks me for help because it got stuck." Later, when Doge finishes cleaning and searches the home base to recharge, I think: "Doge has a desire to recharge and intends to find the home base." I also attribute personality traits: "Doge is a bit dumb, but in a cute way." These are my thoughts, especially when I find out that Doge has knocked over a plant while dutifully vacuuming the house. A machine or algorithm that explains its predictions will find more acceptance. See also the chapter on explanations, which argues that explanations are a social process.

Explanations are used to **manage social interactions**. By creating a shared meaning of something, the explainer influences the actions, emotions and beliefs of the recipient of the explanation. For a machine to interact with us, it may need to shape our emotions and beliefs. Machines have to "persuade" us, so that they can achieve their intended goal. I would not fully accept my robot vacuum cleaner if it did not explain its behavior to some degree. The vacuum cleaner creates a shared meaning of, for example, an "accident" (like getting stuck on the bathroom carpet ... again) by explaining that it got stuck instead of simply stopping to work without comment. Interestingly, there may be a misalignment between the goal of the explaining machine (create trust) and the goal of the recipient (understand the prediction or behavior). Perhaps the full explanation for why Doge got stuck could be that the battery was very low, that one of the wheels is not working properly and that there is a bug that makes the robot go to the same spot over and over again even though there was an obstacle. These reasons (and a few more) caused the robot to get stuck, but it only explained that something was in the way, and that was enough for me to trust its behavior and get a shared meaning of that accident. By the way, Doge got stuck in the bathroom again. We have to remove the carpets each time before we let Doge vacuum.

Doge, our vacuum cleaner, got stuck. As an explanation for the accident, Doge told us that it needs to be on an even surface.

Machine learning models can only be **debugged and audited** when they can be interpreted. Even in low risk environments, such as movie recommendations, the ability to interpret is valuable in the research and development phase as well as after deployment. Later, when a model is used in a product, things can go wrong. An interpretation for an erroneous prediction helps to understand the cause of the error. It delivers a direction for how to fix the system. Consider an example of a husky versus wolf classifier that misclassifies some huskies as wolves. Using interpretable machine learning methods, you would find that the misclassification was due to the snow on the image. The classifier learned to use snow as a feature for classifying images as "wolf", which might make sense in terms of separating wolves from huskies in the training dataset, but not in real-world use.

If you can ensure that the machine learning model can explain decisions, you can also check the following traits more easily (Doshi-Velez and Kim 2017):

- Fairness: Ensuring that predictions are unbiased and do not implicitly or explicitly discriminate against protected groups. An interpretable model can tell you why it has decided that a certain person should not get a loan, and it becomes easier for a human to judge whether the decision is based on a learned demographic (e.g. racial) bias.
- Privacy: Ensuring that sensitive information in the data is protected.
- Reliability or Robustness: Ensuring that small changes in the input do not lead to large changes in the prediction.
- Causality: Check that only causal relationships are picked up.

- Trust: It is easier for humans to trust a system that explains its decisions compared to a black box.

When we do not need interpretability.

The following scenarios illustrate when we do not need or even do not want interpretability of machine learning models.

Interpretability is not required if the model **has no significant impact**. Imagine someone named Mike working on a machine learning side project to predict where his friends will go for their next holidays based on Facebook data. Mike just likes to surprise his friends with educated guesses where they will be going on holidays. There is no real problem if the model is wrong (at worst just a little embarrassment for Mike), nor is there a problem if Mike cannot explain the output of his model. It is perfectly fine not to have interpretability in this case. The situation would change if Mike started building a business around these holiday destination predictions. If the model is wrong, the business could lose money, or the model may work worse for some people because of learned racial bias. As soon as the model has a significant impact, be it financial or social, interpretability becomes relevant.

Interpretability is not required when the **problem is well studied**. Some applications have been sufficiently well studied so that there is enough practical experience with the model and problems with the model have been solved over time. A good example is a machine learning model for optical character recognition that processes images from envelopes and extracts addresses. There is years of experience with these systems and it is clear that they work. In addition, we are not really interested in gaining additional insights about the task at hand.

Interpretability might enable people or programs to **manipulate the system**. Problems with users who deceive a system result from a mismatch between the goals of the creator and the user of a model. Credit scoring is such a system because banks want to ensure that loans are only given to applicants who are likely to return them, and applicants aim to get the loan even if the bank does not want to give them one. This mismatch between the goals introduces incentives for applicants to game the system to increase their chances of getting a loan. If an applicant knows that having more than two credit cards negatively affects his score, he simply returns his third credit card to improve his score, and organizes a new card after the loan has been approved. While his score improved, the actual probability of repaying the loan remained unchanged. The system can only be gamed if the inputs are proxies for a causal feature, but do not actually cause the outcome. Whenever possible, proxy features should be avoided as they make models gameable. For example, Google developed a system called Google Flu Trends to predict flu outbreaks. The system correlated Google searches with flu outbreaks – and it has performed poorly. The distribution of search queries changed and Google Flu Trends missed many flu outbreaks. Google searches do not cause the flu. When people search for symptoms like “fever” it is merely a correlation with actual flu outbreaks. Ideally, models would only use causal features because they would not be gameable.

Taxonomy of Interpretability Methods

Methods for machine learning interpretability can be classified according to various criteria.

Intrinsic or post hoc? This criteria distinguishes whether interpretability is achieved by restricting the complexity of the machine learning model (intrinsic) or by applying methods that analyze the model after training (post hoc). Intrinsic interpretability refers to machine learning models that are considered interpretable due to their simple structure, such as short decision trees or sparse linear models. Post hoc interpretability refers to the application of interpretation methods after model training. Permutation feature importance is, for example, a post hoc interpretation method. Post hoc methods can also be applied to intrinsically interpretable models. For example, permutation feature importance can be computed for decision trees. The organization of the chapters in this book is determined by the distinction between intrinsically interpretable models and post hoc (and model-agnostic) interpretation methods.

Result of the interpretation method The various interpretation methods can be roughly differentiated according to their results.

- **Feature summary statistic**: Many interpretation methods provide summary statistics for each feature. Some methods return a single number per feature, such as feature importance, or a more complex result, such as the pairwise feature interaction strengths, which consist of a number for each feature pair.
- **Feature summary visualization**: Most of the feature summary statistics can also be visualized. Some feature summaries are actually only meaningful if they are visualized and a table would be a wrong choice. The partial dependence of a feature is such a case. Partial dependence plots are curves that show a feature and the average predicted outcome. The best way to present partial dependences is to actually draw the curve instead of printing the coordinates.
- **Model internals (e.g. learned weights)**: The interpretation of intrinsically interpretable models falls into this category. Examples are the weights in linear models or the learned tree structure (the features and thresholds used for the splits) of decision trees. The lines are blurred between model internals and feature summary statistic in, for example, linear models, because the weights are both model internals and summary statistics for the features at the same time. Another method that outputs model internals is the visualization of feature detectors learned in convolutional neural networks. Interpretability methods that output model internals are by definition model-specific (see next criterion).
- **Data point**: This category includes all methods that return data points (already existent or newly created) to make a model interpretable. One method is called counterfactual explanations. To explain the prediction of a data instance, the method finds a similar data point by changing some of the features for which the predicted outcome changes in a relevant way (e.g. a flip in the predicted class). Another example is the identification of prototypes of predicted classes. To be useful, interpretation methods that output new data points require that the data points themselves can be interpreted. This works well for images and texts, but is less useful for tabular data with hundreds of features.

- **Intrinsically interpretable model**: One solution to interpreting black box models is to approximate them (either globally or locally) with an interpretable model. The interpretable model itself is interpreted by looking at internal model parameters or feature summary statistics.

Model-specific or model-agnostic? Model-specific interpretation tools are limited to specific model classes. The interpretation of regression weights in a linear model is a model-specific interpretation, since – by definition – the interpretation of intrinsically interpretable models is always model-specific. Tools that only work for the interpretation of e.g. neural networks are model-specific. Model-agnostic tools can be used on any machine learning model and are applied after the model has been trained (post hoc). These agnostic methods usually work by analyzing feature input and output pairs. By definition, these methods cannot have access to model internals such as weights or structural information.

Local or global? Does the interpretation method explain an individual prediction or the entire model behavior? Or is the scope somewhere in between? Read more about the scope criterion in the next section.

Scope of Interpretability

An algorithm trains a model that produces the predictions. Each step can be evaluated in terms of transparency or interpretability.

Algorithm Transparency

How does the algorithm create the model?

Algorithm transparency is about how the algorithm learns a model from the data and what kind of relationships it can learn. If you use convolutional neural networks to classify images, you can explain that the algorithm learns edge detectors and filters on the lowest layers. This is an understanding of how the algorithm works, but not for the specific model that is learned in the end, and not for how individual predictions are made. Algorithm transparency only requires knowledge of the algorithm and not of the data or learned model. This book focuses on model interpretability and not algorithm transparency. Algorithms such as the least squares method for linear models are well studied and understood. They are characterized by a high transparency. Deep learning approaches (pushing a gradient through a network with millions of weights) are less well understood and the inner workings are the focus of ongoing research. They are considered less transparent.

Global, Holistic Model Interpretability

How does the trained model make predictions?

You could describe a model as interpretable if you can comprehend the entire model at once (Lipton 2016[14]). To explain the global model output, you need the trained model, knowledge of the algorithm and the data. This level of interpretability is about understanding how the model makes decisions, based on a holistic view of its features and each of the learned components such as weights, other parameters, and structures. Which features are important and what kind of interactions between them take place? Global model interpretability helps to understand the distribution of your target outcome based on the features. Global model interpretability is very difficult to achieve in practice. Any model that exceeds a handful of parameters or weights is unlikely to fit into the short-term memory of the average human. I argue that you cannot really imagine a linear model with 5 features, because it would mean drawing the estimated hyperplane mentally in a 5-dimensional space. Any feature space with more than 3 dimensions is simply inconceivable for humans. Usually, when people try to comprehend a model, they consider only parts of it, such as the weights in linear models.

Global Model Interpretability on a Modular Level

How do parts of the model affect predictions?

[14]Lipton, Zachary C. "The mythos of model interpretability." arXiv preprint arXiv:1606.03490, (2016).

A Naive Bayes model with many hundreds of features would be too big for me and you to keep in our working memory. And even if we manage to memorize all the weights, we would not be able to quickly make predictions for new data points. In addition, you need to have the joint distribution of all features in your head to estimate the importance of each feature and how the features affect the predictions on average. An impossible task. But you can easily understand a single weight. While global model interpretability is usually out of reach, there is a good chance of understanding at least some models on a modular level. Not all models are interpretable at a parameter level. For linear models, the interpretable parts are the weights, for trees it would be the splits (selected features plus cut-off points) and leaf node predictions. Linear models, for example, look like as if they could be perfectly interpreted on a modular level, but the interpretation of a single weight is interlocked with all other weights. The interpretation of a single weight always comes with the footnote that the other input features remain at the same value, which is not the case with many real applications. A linear model that predicts the value of a house, that takes into account both the size of the house and the number of rooms, can have a negative weight for the room feature. It can happen because there is already the highly correlated house size feature. In a market where people prefer larger rooms, a house with fewer rooms could be worth more than a house with more rooms if both have the same size. The weights only make sense in the context of the other features in the model. But the weights in a linear model can still be interpreted better than the weights of a deep neural network.

Local Interpretability for a Single Prediction

Why did the model make a certain prediction for an instance?

You can zoom in on a single instance and examine what the model predicts for this input, and explain why. If you look at an individual prediction, the behavior of the otherwise complex model might behave more pleasantly. Locally, the prediction might only depend linearly or monotonously on some features, rather than having a complex dependence on them. For example, the value of a house may depend nonlinearly on its size. But if you are looking at only one particular 100 square meters house, there is a possibility that for that data subset, your model prediction depends linearly on the size. You can find this out by simulating how the predicted price changes when you increase or decrease the size by 10 square meters. Local explanations can therefore be more accurate than global explanations. This book presents methods that can make individual predictions more interpretable in the section on model-agnostic methods.

Local Interpretability for a Group of Predictions

Why did the model make specific predictions for a group of instances?

Model predictions for multiple instances can be explained either with global model interpretation methods (on a modular level) or with explanations of individual instances. The global methods can be applied by taking the group of instances, treating them as if the group were the complete dataset, and using the global methods with this subset. The individual explanation methods can be used on each instance and then listed or aggregated for the entire group.

Evaluation of Interpretability

There is no real consensus about what interpretability is in machine learning. Nor is it clear how to measure it. But there is some initial research on this and an attempt to formulate some approaches for evaluation, as described in the following section.

Doshi-Velez and Kim (2017) propose three main levels for the evaluation of interpretability:

Application level evaluation (real task): Put the explanation into the product and have it tested by the end user. Imagine fracture detection software with a machine learning component that locates and marks fractures in X-rays. At the application level, radiologists would test the fracture detection software directly to evaluate the model. This requires a good experimental setup and an understanding of how to assess quality. A good baseline for this is always how good a human would be at explaining the same decision.

Human level evaluation (simple task) is a simplified application level evaluation. The difference is that these experiments are not carried out with the domain experts, but with laypersons. This makes experiments cheaper (especially if the domain experts are radiologists) and it is easier to find more testers. An example would be to show a user different explanations and the user would choose the best one.

Function level evaluation (proxy task) does not require humans. This works best when the class of model used has already been evaluated by someone else in a human level evaluation. For example, it might be known that the end users understand decision trees. In this case, a proxy for explanation quality may be the depth of the tree. Shorter trees would get a better explainability score. It would make sense to add the constraint that the predictive performance of the tree remains good and does not decrease too much compared to a larger tree.

The next chapter focuses on the evaluation of explanations for individual predictions on the function level. What are the relevant properties of explanations that we would consider for their evaluation?

Properties of Explanations

We want to explain the predictions of a machine learning model. To achieve this, we rely on some explanation method, which is an algorithm that generates explanations. **An explanation usually relates the feature values of an instance to its model prediction in a humanly understandable way**. Other types of explanations consist of a set of data instances (e.g in the case of the k-nearest neighbor model). For example, we could predict cancer risk using a support vector machine and explain predictions using the local surrogate method, which generates decision trees as explanations. Or we could use a linear regression model instead of a support vector machine. The linear regression model is already equipped with an explanation method (interpretation of the weights).

We take a closer look at the properties of explanation methods and explanations (Robnik-Sikonja and Bohanec, 2018[15]). These properties can be used to judge how good an explanation method or explanation is. It is not clear for all these properties how to measure them correctly, so one of the challenges is to formalize how they could be calculated.

Properties of Explanation Methods

- **Expressive Power** is the "language" or structure of the explanations the method is able to generate. An explanation method could generate IF-THEN rules, decision trees, a weighted sum, natural language or something else.
- **Translucency** describes how much the explanation method relies on looking into the machine learning model, like its parameters. For example, explanation methods relying on intrinsically interpretable models like the linear regression model (model-specific) are highly translucent. Methods only relying on manipulating inputs and observing the predictions have zero translucency. Depending on the scenario, different levels of translucency might be desirable. The advantage of high translucency is that the method can rely on more information to generate explanations. The advantage of low translucency is that the explanation method is more portable.
- **Portability** describes the range of machine learning models with which the explanation method can be used. Methods with a low translucency have a higher portability because they treat the machine learning model as a black box. Surrogate models might be the explanation method with the highest portability. Methods that only work for e.g. recurrent neural networks have low portability.
- **Algorithmic Complexity** describes the computational complexity of the method that generates the explanation. This property is important to consider when computation time is a bottleneck in generating explanations.

Properties of Individual Explanations

- **Accuracy**: How well does an explanation predict unseen data? High accuracy is especially important if the explanation is used for predictions in place of the machine learning model.

[15]Robnik-Sikonja, Marko, and Marko Bohanec. "Perturbation-based explanations of prediction models." Human and Machine Learning. Springer, Cham. 159-175. (2018).

Low accuracy can be fine if the accuracy of the machine learning model is also low, and if the goal is to explain what the black box model does. In this case, only fidelity is important.

- **Fidelity**: How well does the explanation approximate the prediction of the black box model? High fidelity is one of the most important properties of an explanation, because an explanation with low fidelity is useless to explain the machine learning model. Accuracy and fidelity are closely related. If the black box model has high accuracy and the explanation has high fidelity, the explanation also has high accuracy. Some explanations offer only local fidelity, meaning the explanation only approximates well to the model prediction for a subset of the data (e.g. local surrogate models) or even for only an individual data instance (e.g. Shapley Values).
- **Consistency**: How much does an explanation differ between models that have been trained on the same task and that produce similar predictions? For example, I train a support vector machine and a linear regression model on the same task and both produce very similar predictions. I compute explanations using a method of my choice and analyze how different the explanations are. If the explanations are very similar, the explanations are highly consistent. I find this property somewhat tricky, since the two models could use different features, but get similar predictions (also called "Rashomon Effect"[16]). In this case a high consistency is not desirable because the explanations have to be very different. High consistency is desirable if the models really rely on similar relationships.
- **Stability**: How similar are the explanations for similar instances? While consistency compares explanations between models, stability compares explanations between similar instances for a fixed model. High stability means that slight variations in the features of an instance do not substantially change the explanation (unless these slight variations also strongly change the prediction). A lack of stability can be the result of a high variance of the explanation method. In other words, the explanation method is strongly affected by slight changes of the feature values of the instance to be explained. A lack of stability can also be caused by non-deterministic components of the explanation method, such as a data sampling step, like the local surrogate method uses. High stability is always desirable.
- **Comprehensibility**: How well do humans understand the explanations? This looks just like one more property among many, but it is the elephant in the room. Difficult to define and measure, but extremely important to get right. Many people agree that comprehensibility depends on the audience. Ideas for measuring comprehensibility include measuring the size of the explanation (number of features with a non-zero weight in a linear model, number of decision rules, ...) or testing how well people can predict the behavior of the machine learning model from the explanations. The comprehensibility of the features used in the explanation should also be considered. A complex transformation of features might be less comprehensible than the original features.
- **Certainty**: Does the explanation reflect the certainty of the machine learning model? Many machine learning models only give predictions without a statement about the models confidence that the prediction is correct. If the model predicts a 4% probability of cancer for one patient, is it as certain as the 4% probability that another patient, with different feature values, received? An explanation that includes the model's certainty is very useful.

[16] https://en.wikipedia.org/wiki/Rashomon_effect

- **Degree of Importance**: How well does the explanation reflect the importance of features or parts of the explanation? For example, if a decision rule is generated as an explanation for an individual prediction, is it clear which of the conditions of the rule was the most important?
- **Novelty**: Does the explanation reflect whether a data instance to be explained comes from a "new" region far removed from the distribution of training data? In such cases, the model may be inaccurate and the explanation may be useless. The concept of novelty is related to the concept of certainty. The higher the novelty, the more likely it is that the model will have low certainty due to lack of data.
- **Representativeness**: How many instances does an explanation cover? Explanations can cover the entire model (e.g. interpretation of weights in a linear regression model) or represent only an individual prediction (e.g. Shapley Values).

Human-friendly Explanations

Let us dig deeper and discover what we humans see as "good" explanations and what the implications are for interpretable machine learning. Humanities research can help us find out. Miller (2017) has conducted a huge survey of publications on explanations, and this chapter builds on his summary.

In this chapter, I want to convince you of the following: As an explanation for an event, humans prefer short explanations (only 1 or 2 causes) that contrast the current situation with a situation in which the event would not have occurred. Especially abnormal causes provide good explanations. Explanations are social interactions between the explainer and the explainee (recipient of the explanation) and therefore the social context has a great influence on the actual content of the explanation.

When you need explanations with ALL factors for a particular prediction or behavior, you do not want a human-friendly explanation, but a complete causal attribution. You probably want a causal attribution if you are legally required to specify all influencing features or if you debug the machine learning model. In this case, ignore the following points. In all other cases, where lay people or people with little time are the recipients of the explanation, the following sections should be interesting to you.

What Is an Explanation?

An explanation is the **answer to a why-question** (Miller 2017).

- Why did not the treatment work on the patient?
- Why was my loan rejected?
- Why have we not been contacted by alien life yet?

The first two questions can be answered with an "everyday"-explanation, while the third one comes from the category "More general scientific phenomena and philosophical questions". We focus on the "everyday"-type explanations, because those are relevant to interpretable machine learning. Questions that start with "how" can usually be rephrased as "why" questions: "How was my loan rejected?" can be turned into "Why was my loan rejected?".

In the following, the term "explanation" refers to the social and cognitive process of explaining, but also to the product of these processes. The explainer can be a human being or a machine.

What Is a Good Explanation?

This section further condenses Miller's summary on "good" explanations and adds concrete implications for interpretable machine learning.

Explanations are contrastive (Lipton 1990[17]): Humans usually do not ask why a certain prediction was made, but why this prediction was made *instead of another prediction*. We tend to think in counterfactual cases, i.e. "How would the prediction have been if input X had been different?". For a house price prediction, the house owner might be interested in why the predicted price was high compared to the lower price they had expected. If my loan application is rejected, I do not care to hear all the factors that generally speak for or against a rejection. I am interested in the factors in my application that would need to change to get the loan. I want to know the contrast between my application and the would-be-accepted version of my application. The recognition that contrasting explanations matter is an important finding for explainable machine learning. From most interpretable models, you can extract an explanation that implicitly contrasts a prediction of an instance with the prediction of an artificial data instance or an average of instances. Physicians might ask: "Why did the drug not work for my patient?". And they might want an explanation that contrasts their patient with a patient for whom the drug worked and who is similar to the non-responding patient. Contrastive explanations are easier to understand than complete explanations. A complete explanation of the physician's question why the drug does not work might include: The patient has had the disease for 10 years, 11 genes are over-expressed, the patients body is very quick in breaking the drug down into ineffective chemicals, … A contrastive explanation might be much simpler: In contrast to the responding patient, the non-responding patient has a certain combination of genes that make the drug less effective. The best explanation is the one that highlights the greatest difference between the object of interest and the reference object. **What it means for interpretable machine learning**:
Humans do not want a complete explanation for a prediction, but want to compare what the differences were to another instance's prediction (can be an artificial one). Creating contrastive explanations is application-dependent because it requires a point of reference for comparison. And this may depend on the data point to be explained, but also on the user receiving the explanation. A user of a house price prediction website might want to have an explanation of a house price prediction contrastive to their own house or maybe to another house on the website or maybe to an average house in the neighborhood. The solution for the automated creation of contrastive explanations might also involve finding prototypes or archetypes in the data.

Explanations are selected: People do not expect explanations that cover the actual and complete list of causes of an event. We are used to selecting one or two causes from a variety of possible causes as THE explanation. As proof, turn on the TV news: "The decline in stock prices is blamed on a growing backlash against the company's product due to problems with the latest software update."
"Tsubasa and his team lost the match because of a weak defense: they gave their opponents too much room to play out their strategy."
"The increasing distrust of established institutions and our government are the main factors that have reduced voter turnout."
The fact that an event can be explained by various causes is called the Rashomon Effect. Rashomon is a Japanese movie that tells alternative, contradictory stories (explanations) about the death of a samurai. For machine learning models, it is advantageous if a good prediction can be made from different features. Ensemble methods that combine multiple models with different features (different

[17]Lipton, Peter. "Contrastive explanation." Royal Institute of Philosophy Supplements 27 (1990): 247-266.

explanations) usually perform well because averaging over those "stories" makes the predictions more robust and accurate. But it also means that there is more than one selective explanation why a certain prediction was made. **What it means for interpretable machine learning**:
Make the explanation very short, give only 1 to 3 reasons, even if the world is more complex. The LIME method does a good job with this.

Explanations are social: They are part of a conversation or interaction between the explainer and the receiver of the explanation. The social context determines the content and nature of the explanations. If I wanted to explain to a technical person why digital cryptocurrencies are worth so much, I would say things like: "The decentralized, distributed, blockchain-based ledger, which cannot be controlled by a central entity, resonates with people who want to secure their wealth, which explains the high demand and price." But to my grandmother I would say: "Look, Grandma: Cryptocurrencies are a bit like computer gold. People like and pay a lot for gold, and young people like and pay a lot for computer gold." **What it means for interpretable machine learning**:
Pay attention to the social environment of your machine learning application and the target audience. Getting the social part of the machine learning model right depends entirely on your specific application. Find experts from the humanities (e.g. psychologists and sociologists) to help you.

Explanations focus on the abnormal. People focus more on abnormal causes to explain events (Kahnemann and Tversky, 1981[18]). These are causes that had a small probability but nevertheless happened. The elimination of these abnormal causes would have greatly changed the outcome (counterfactual explanation). Humans consider these kinds of "abnormal" causes as good explanations. An example from Štrumbelj and Kononenko (2011)[19] is: Assume we have a dataset of test situations between teachers and students. Students attend a course and pass the course directly after successfully giving a presentation. The teacher has the option to additionally ask the student questions to test their knowledge. Students who cannot answer these questions will fail the course. Students can have different levels of preparation, which translates into different probabilities for correctly answering the teacher's questions (if they decide to test the student). We want to predict whether a student will pass the course and explain our prediction. The chance of passing is 100% if the teacher does not ask any additional questions, otherwise the probability of passing depends on the student's level of preparation and the resulting probability of answering the questions correctly.
Scenario 1: The teacher usually asks the students additional questions (e.g. 95 out of 100 times). A student who did not study (10% chance to pass the question part) was not one of the lucky ones and gets additional questions that he fails to answer correctly. Why did the student fail the course? I would say that it was the student's fault to not study.
Scenario 2: The teacher rarely asks additional questions (e.g. 2 out of 100 times). For a student who has not studied for the questions, we would predict a high probability of passing the course because questions are unlikely. Of course, one of the students did not prepare for the questions, which gives him a 10% chance of passing the questions. He is unlucky and the teacher asks additional questions that the student cannot answer and he fails the course. What is the reason for the failure? I would

[18]Kahneman, Daniel, and Amos Tversky. "The Simulation Heuristic." Stanford Univ CA Dept of Psychology. (1981).
[19]Štrumbelj, Erik, and Igor Kononenko. "A general method for visualizing and explaining black-box regression models." In International Conference on Adaptive and Natural Computing Algorithms, 21–30. Springer. (2011).

argue that now, the better explanation is "because the teacher tested the student". It was unlikely that the teacher would test, so the teacher behaved abnormally. **What it means for interpretable machine learning**:
If one of the input features for a prediction was abnormal in any sense (like a rare category of a categorical feature) and the feature influenced the prediction, it should be included in an explanation, even if other 'normal' features have the same influence on the prediction as the abnormal one. An abnormal feature in our house price prediction example might be that a rather expensive house has two balconies. Even if some attribution method finds that the two balconies contribute as much to the price difference as the above average house size, the good neighborhood or the recent renovation, the abnormal feature "two balconies" might be the best explanation for why the house is so expensive.

Explanations are truthful. Good explanations prove to be true in reality (i.e. in other situations). But disturbingly, this is not the most important factor for a "good" explanation. For example, selectiveness seems to be more important than truthfulness. An explanation that selects only one or two possible causes rarely covers the entire list of relevant causes. Selectivity omits part of the truth. It is not true that only one or two factors, for example, have caused a stock market crash, but the truth is that there are millions of causes that influence millions of people to act in such a way that in the end a crash was caused. **What it means for interpretable machine learning**:
The explanation should predict the event as truthfully as possible, which in machine learning is sometimes called **fidelity**. So if we say that a second balcony increases the price of a house, then that also should apply to other houses (or at least to similar houses). For humans, fidelity of an explanation is not as important as its selectivity, its contrast and its social aspect.

Good explanations are consistent with prior beliefs of the explainee. Humans tend to ignore information that is inconsistent with their prior beliefs. This effect is called confirmation bias (Nickerson 1998[20]). Explanations are not spared by this kind of bias. People will tend to devalue or ignore explanations that do not agree with their beliefs. The set of beliefs varies from person to person, but there are also group-based prior beliefs such as political worldviews. **What it means for interpretable machine learning**:
Good explanations are consistent with prior beliefs. This is difficult to integrate into machine learning and would probably drastically compromise predictive performance. Our prior belief for the effect of house size on predicted price is that the larger the house, the higher the price. Let us assume that a model also shows a negative effect of house size on the predicted price for a few houses. The model has learned this because it improves predictive performance (due to some complex interactions), but this behavior strongly contradicts our prior beliefs. You can enforce monotonicity constraints (a feature can only affect the prediction in one direction) or use something like a linear model that has this property.

Good explanations are general and probable. A cause that can explain many events is very general and could be considered a good explanation. Note that this contradicts the claim that abnormal causes make good explanations. As I see it, abnormal causes beat general causes. Abnormal causes are by definition rare in the given scenario. In the absence of an abnormal event, a general explanation is considered a good explanation. Also remember that people tend to misjudge

[20]Nickerson, Raymond S. "Confirmation Bias: A ubiquitous phenomenon in many guises." Review of General Psychology 2 (2). Educational Publishing Foundation: 175. (1998).

probabilities of joint events. (Joe is a librarian. Is he more likely to be a shy person or to be a shy person who likes to read books?) A good example is "The house is expensive because it is big", which is a very general, good explanation of why houses are expensive or cheap. **What it means for interpretable machine learning**:
Generality can easily be measured by the feature's support, which is the number of instances to which the explanation applies divided by the total number of instances.

Datasets

Throughout the book, all models and techniques are applied to real datasets that are freely available online. We will use different datasets for different tasks: Classification, regression and text classification.

Bike Rentals (Regression)

This dataset contains daily counts of rented bicycles from the bicycle rental company Capital-Bikeshare[21] in Washington D.C., along with weather and seasonal information. The data was kindly made openly available by Capital-Bikeshare. Fanaee-T and Gama (2013)[22] added weather data and season information. The goal is to predict how many bikes will be rented depending on the weather and the day. The data can be downloaded from the UCI Machine Learning Repository[23].

New features were added to the dataset and not all original features were used for the examples in this book. Here is the list of features that were used:

- Count of bicycles including both casual and registered users. The count is used as the target in the regression task.
- The season, either spring, summer, fall or winter.
- Indicator whether the day was a holiday or not.
- The year, either 2011 or 2012.
- Number of days since the 01.01.2011 (the first day in the dataset). This feature was introduced to take account of the trend over time.
- Indicator whether the day was a working day or weekend.
- The weather situation on that day. One of:
  - clear, few clouds, partly cloudy, cloudy
  - mist + clouds, mist + broken clouds, mist + few clouds, mist
  - light snow, light rain + thunderstorm + scattered clouds, light rain + scattered clouds
  - heavy rain + ice pallets + thunderstorm + mist, snow + mist
- Temperature in degrees Celsius.
- Relative humidity in percent (0 to 100).
- Wind speed in km per hour.

For the examples in this book, the data has been slightly processed. You can find the processing R-script in the book's Github repository[24] together with the final RData file[25].

[21] https://www.capitalbikeshare.com/

[22] Fanaee-T, Hadi, and Joao Gama. "Event labeling combining ensemble detectors and background knowledge." Progress in Artificial Intelligence. Springer Berlin Heidelberg, 1–15. doi:10.1007/s13748-013-0040-3. (2013).

[23] http://archive.ics.uci.edu/ml/datasets/Bike+Sharing+Dataset

[24] https://github.com/christophM/interpretable-ml-book/blob/master/R/get-bike-sharing-dataset.R

[25] https://github.com/christophM/interpretable-ml-book/blob/master/data/bike.RData

YouTube Spam Comments (Text Classification)

As an example for text classification we work with 1956 comments from 5 different YouTube videos. Thankfully, the authors who used this dataset in an article on spam classification made the data freely available[26] (Alberto, Lochter, and Almeida (2015)[27]).

The comments were collected via the YouTube API from five of the ten most viewed videos on YouTube in the first half of 2015. All 5 are music videos. One of them is "Gangnam Style" by Korean artist Psy. The other artists were Katy Perry, LMFAO, Eminem, and Shakira.

Checkout some of the comments. The comments were manually labeled as spam or legitimate. Spam was coded with a "1" and legitimate comments with a "0".

CONTENT	CLASS
Huh, anyway check out this you[tube] channel: kobyoshi02	1
Hey guys check out my new channel and our first vid THIS IS US THE MONKEYS!!! I'm the monkey in the white shirt,please leave a like comment and please subscribe!!!!	1
just for test I have to say murdev.com	1
me shaking my sexy ass on my channel enjoy _	1
watch?v=vtaRGgvGtWQ Check this out .	1
Hey, check out my new website!! This site is about kids stuff. kidsmediausa . com	1
Subscribe to my channel	1
i turned it on mute as soon is i came on i just wanted to check the views…	0
You should check my channel for Funny VIDEOS!!	1
and u should.d check my channel and tell me what I should do next!	1

You can also go to YouTube and take a look at the comment section. But please do not get caught in YouTube hell and end up watching videos of monkeys stealing and drinking cocktails from tourists on the beach. The Google Spam detector has also probably changed a lot since 2015.

Watch the view-record breaking video "Gangnam Style" here[28].

If you want to play around with the data, you can find the RData file[29] along with the R-script[30] with some convenience functions in the book's Github repository.

[26] http://dcomp.sor.ufscar.br/talmeida/youtubespamcollection/

[27] Alberto, Túlio C, Johannes V Lochter, and Tiago A Almeida. "Tubespam: comment spam filtering on YouTube." In Machine Learning and Applications (Icmla), Ieee 14th International Conference on, 138–43. IEEE. (2015).

[28] https://www.youtube.com/watch?v=9bZkp7q19f0&feature=player_embedded

[29] https://github.com/christophM/interpretable-ml-book/blob/master/data/ycomments.RData

[30] https://github.com/christophM/interpretable-ml-book/blob/master/R/get-SpamTube-dataset.R

Risk Factors for Cervical Cancer (Classification)

The cervical cancer dataset contains indicators and risk factors for predicting whether a woman will get cervical cancer. The features include demographic data (such as age), lifestyle, and medical history. The data can be downloaded from the UCI Machine Learning repository[31] and is described by Fernandes, Cardoso, and Fernandes (2017)[32].

The subset of data features used in the book's examples are:

- Age in years
- Number of sexual partners
- First sexual intercourse (age in years)
- Number of pregnancies
- Smoking yes or no
- Smoking (in years)
- Hormonal contraceptives yes or no
- Hormonal contraceptives (in years)
- Intrauterine device yes or no (IUD)
- Number of years with an intrauterine device (IUD)
- Has patient ever had a sexually transmitted disease (STD) yes or no
- Number of STD diagnoses
- Time since first STD diagnosis
- Time since last STD diagnosis
- The biopsy results "Healthy" or "Cancer". Target outcome.

The biopsy serves as the gold standard for diagnosing cervical cancer. For the examples in this book, the biopsy outcome was used as the target. Missing values for each column were imputed by the mode (most frequent value), which is probably a bad solution, since the true answer could be correlated with the probability that a value is missing. There is probably a bias because the questions are of a very private nature. But this is not a book about missing data imputation, so the mode imputation will have to suffice for the examples.

To reproduce the examples of this book with this dataset, find the preprocessing R-script[33] and the final RData file[34] in the book's Github repository.

[31] https://archive.ics.uci.edu/ml/datasets/Cervical+cancer+%28Risk+Factors%29

[32] Fernandes, Kelwin, Jaime S Cardoso, and Jessica Fernandes. "Transfer learning with partial observability applied to cervical cancer screening." In Iberian Conference on Pattern Recognition and Image Analysis, 243–50. Springer. (2017).

[33] https://github.com/christophM/interpretable-ml-book/blob/master/R/get-cervical-cancer-dataset.R

[34] https://github.com/christophM/interpretable-ml-book/blob/master/data/cervical.RData

Interpretable Models

The easiest way to achieve interpretability is to use only a subset of algorithms that create interpretable models. Linear regression, logistic regression and the decision tree are commonly used interpretable models.

In the following chapters we will talk about these models. Not in detail, only the basics, because there is already a ton of books, videos, tutorials, papers and more material available. We will focus on how to interpret the models. The book discusses linear regression, logistic regression, other linear regression extensions, decision trees, decision rules and the RuleFit algorithm in more detail. It also lists other interpretable models.

All interpretable models explained in this book are interpretable on a modular level, with the exception of the k-nearest neighbors method. The following table gives an overview of the interpretable model types and their properties. A model is linear if the association between features and target is modelled linearly. A model with monotonicity constraints ensures that the relationship between a feature and the target outcome always goes in the same direction over the entire range of the feature: An increase in the feature value either always leads to an increase or always to a decrease in the target outcome. Monotonicity is useful for the interpretation of a model because it makes it easier to understand a relationship. Some models can automatically include interactions between features to predict the target outcome. You can include interactions in any type of model by manually creating interaction features. Interactions can improve predictive performance, but too many or too complex interactions can hurt interpretability. Some models handle only regression, some only classification, and still others both.

From this table, you can select a suitable interpretable model for your task, either regression (regr) or classification (class):

Algorithm	Linear	Monotone	Interaction	Task
Linear regression	Yes	Yes	No	regr
Logistic regression	No	Yes	No	class
Decision trees	No	Some	Yes	class,regr
RuleFit	Yes	No	Yes	class,regr
Naive Bayes	No	Yes	No	class
k-nearest neighbors	No	No	No	class,regr

Linear Regression

A linear regression model predicts the target as a weighted sum of the feature inputs. The linearity of the learned relationship makes the interpretation easy. Linear regression models have long been used by statisticians, computer scientists and other people who tackle quantitative problems.

Linear models can be used to model the dependence of a regression target y on some features x. The learned relationships are linear and can be written for a single instance i as follows:

$$y = \beta_0 + \beta_1 x_1 + \ldots + \beta_p x_p + \epsilon$$

The predicted outcome of an instance is a weighted sum of its p features. The betas (β_j) represent the learned feature weights or coefficients. The first weight in the sum (β_0) is called the intercept and is not multiplied with a feature. The epsilon (ϵ) is the error we still make, i.e. the difference between the prediction and the actual outcome. These errors are assumed to follow a Gaussian distribution, which means that we make errors in both negative and positive directions and make many small errors and few large errors.

Various methods can be used to estimate the optimal weight. The ordinary least squares method is usually used to find the weights that minimize the squared differences between the actual and the estimated outcomes:

$$\hat{\boldsymbol{\beta}} = \arg\min_{\beta_0,\ldots,\beta_p} \sum_{i=1}^{n} \left(y^{(i)} - \left(\beta_0 + \sum_{j=1}^{p} \beta_j x_j^{(i)} \right) \right)^2$$

We will not discuss in detail how the optimal weights can be found, but if you are interested, you can read chapter 3.2 of the book "The Elements of Statistical Learning" (Friedman, Hastie and Tibshirani 2009)[35] or one of the other online resources on linear regression models.

The biggest advantage of linear regression models is linearity: It makes the estimation procedure simple and, most importantly, these linear equations have an easy to understand interpretation on a modular level (i.e. the weights). This is one of the main reasons why the linear model and all similar models are so widespread in academic fields such as medicine, sociology, psychology, and many other quantitative research fields. For example, in the medical field, it is not only important to predict the clinical outcome of a patient, but also to quantify the influence of the drug and at the same time take sex, age, and other features into account in an interpretable way.

Estimated weights come with confidence intervals. A confidence interval is a range for the weight estimate that covers the "true" weight with a certain confidence. For example, a 95% confidence interval for a weight of 2 could range from 1 to 3. The interpretation of this interval would be: If we repeated the estimation 100 times with newly sampled data, the confidence interval would include

[35] Friedman, Jerome, Trevor Hastie, and Robert Tibshirani. "The elements of statistical learning". www.web.stanford.edu/~hastie/ElemStatLearn/ (2009).

the true weight in 95 out of 100 cases, given that the linear regression model is the correct model for the data.

Whether the model is the "correct" model depends on whether the relationships in the data meet certain assumptions, which are linearity, normality, homoscedasticity, independence, fixed features, and absence of multicollinearity.

Linearity
The linear regression model forces the prediction to be a linear combination of features, which is both its greatest strength and its greatest limitation. Linearity leads to interpretable models. Linear effects are easy to quantify and describe. They are additive, so it is easy to separate the effects. If you suspect feature interactions or a nonlinear association of a feature with the target value, you can add interaction terms or use regression splines.

Normality
It is assumed that the target outcome given the features follows a normal distribution. If this assumption is violated, the estimated confidence intervals of the feature weights are invalid.

Homoscedasticity (constant variance)
The variance of the error terms is assumed to be constant over the entire feature space. Suppose you want to predict the value of a house given the living area in square meters. You estimate a linear model that assumes that, regardless of the size of the house, the error around the predicted response has the same variance. This assumption is often violated in reality. In the house example, it is plausible that the variance of error terms around the predicted price is higher for larger houses, since prices are higher and there is more room for price fluctuations. Suppose the average error (difference between predicted and actual price) in your linear regression model is 50,000 Euros. If you assume homoscedasticity, you assume that the average error of 50,000 is the same for houses that cost 1 million and for houses that cost only 40,000. This is unreasonable because it would mean that we can expect negative house prices.

Independence
It is assumed that each instance is independent of any other instance. If you perform repeated measurements, such as multiple blood tests per patient, the data points are not independent. For dependent data you need special linear regression models, such as mixed effect models or GEEs. If you use the "normal" linear regression model, you might draw wrong conclusions from the model.

Fixed features
The input features are considered "fixed". Fixed means that they are treated as "given constants" and not as statistical variables. This implies that they are free of measurement errors. This is a rather unrealistic assumption. Without that assumption, however, you would have to fit very complex measurement error models that account for the measurement errors of your input features. And usually you do not want to do that.

Absence of multicollinearity
You do not want strongly correlated features, because this messes up the estimation of the weights. In a situation where two features are strongly correlated, it becomes problematic to estimate the

weights because the feature effects are additive and it becomes indeterminable to which of the correlated features to attribute the effects.

Interpretation

The interpretation of a weight in the linear regression model depends on the type of the corresponding feature.

- Numerical feature: Increasing the numerical feature by one unit changes the estimated outcome by its weight. An example of a numerical feature is the size of a house.
- Binary feature: A feature that takes one of two possible values for each instance. An example is the feature "House comes with a garden". One of the values counts as the reference category (in some programming languages encoded with 0), such as "No garden". Changing the feature from the reference category to the other category changes the estimated outcome by the feature's weight.
- Categorical feature with multiple categories: A feature with a fixed number of possible values. An example is the feature "floor type", with possible categories "carpet", "laminate" and "parquet". A solution to deal with many categories is the one-hot-encoding, meaning that each category has its own binary column. For a categorical feature with L categories, you only need L-1 columns, because the L-th column would have redundant information (e.g. when columns 1 to L-1 all have value 0 for one instance, we know that the categorical feature of this instance takes on category L). The interpretation for each category is then the same as the interpretation for binary features. Some languages, such as R, allow you to encode categorical features in various ways, as described later in this chapter.
- Intercept β_0: The intercept is the feature weight for the "constant feature", which is always 1 for all instances. Most software packages automatically add this "1"-feature to estimate the intercept. The interpretation is: For an instance with all numerical feature values at zero and the categorical feature values at the reference categories, the model prediction is the intercept weight. The interpretation of the intercept is usually not relevant because instances with all features values at zero often make no sense. The interpretation is only meaningful when the features have been standardised (mean of zero, standard deviation of one). Then the intercept reflects the predicted outcome of an instance where all features are at their mean value.

The interpretation of the features in the linear regression model can be automated by using following text templates.

Interpretation of a Numerical Feature

An increase of feature x_k by one unit increases the prediction for y by β_k units when all other feature values remain fixed.

Interpretation of a Categorical Feature

Changing feature x_k from the reference category to the other category increases the prediction for y by β_k when all other features remain fixed.

Another important measurement for interpreting linear models is the R-squared measurement. R-squared tells you how much of the total variance of your target outcome is explained by the model. The higher R-squared, the better your model explains the data. The formula for calculating R-squared is:

$$R^2 = 1 - SSE/SST$$

SSE is the squared sum of the error terms:

$$SSE = \sum_{i=1}^{n}(y^{(i)} - \hat{y}^{(i)})^2$$

SST is the squared sum of the data variance:

$$SST = \sum_{i=1}^{n}(y^{(i)} - \bar{y})^2$$

The SSE tells you how much variance remains after fitting the linear model, which is measured by the squared differences between the predicted and actual target values. SST is the total variance of the target outcome. R-squared tells you how much of your variance can be explained by the linear model. R-squared ranges between 0 for models where the model does not explain the data at all and 1 for models that explain all of the variance in your data.

There is a catch, because R-squared increases with the number of features in the model, even if they do not contain any information about the target value at all. Therefore, it is better to use the adjusted R-squared, which accounts for the number of features used in the model. Its calculation is:

$$\bar{R}^2 = R^2 - (1 - R^2)\frac{p}{n - p - 1}$$

where p is the number of features and n the number of instances.

It is not meaningful to interpret a model with very low (adjusted) R-squared, because such a model basically does not explain much of the variance. Any interpretation of the weights would not be meaningful.

Feature Importance

The importance of a feature in a linear regression model can be measured by the absolute value of its t-statistic. The t-statistic is the estimated weight scaled with its standard error.

$$t_{\hat{\beta}_j} = \frac{\hat{\beta}_j}{SE(\hat{\beta}_j)}$$

Let us examine what this formula tells us: The importance of a feature increases with increasing

weight. This makes sense. The more variance the estimated weight has (= the less certain we are about the correct value), the less important the feature is. This also makes sense.

Example

In this example, we use the linear regression model to predict the number of rented bikes on a particular day, given weather and calendar information. For the interpretation, we examine the estimated regression weights. The features consist of numerical and categorical features. For each feature, the table shows the estimated weight, the standard error of the estimate (SE), and the absolute value of the t-statistic (|t|).

	Weight	SE	t
(Intercept)	2399.4	238.3	10.1
seasonSUMMER	899.3	122.3	7.4
seasonFALL	138.2	161.7	0.9
seasonWINTER	425.6	110.8	3.8
holidayHOLIDAY	-686.1	203.3	3.4
workingdayWORKING DAY	124.9	73.3	1.7
weathersitMISTY	-379.4	87.6	4.3
weathersitRAIN/SNOW/STORM	-1901.5	223.6	8.5
temp	110.7	7.0	15.7
hum	-17.4	3.2	5.5
windspeed	-42.5	6.9	6.2
days_since_2011	4.9	0.2	28.5

Interpretation of a numerical feature (temperature): An increase of the temperature by 1 degree Celsius increases the predicted number of bicycles by 110.7, when all other features remain fixed.

Interpretation of a categorical feature ("weathersit"): The estimated number of bicycles is -1901.5 lower when it is raining, snowing or stormy, compared to good weather – again assuming that all other features do not change. When the weather is misty, the predicted number of bicycles is -379.4 lower compared to good weather, given all other features remain the same.

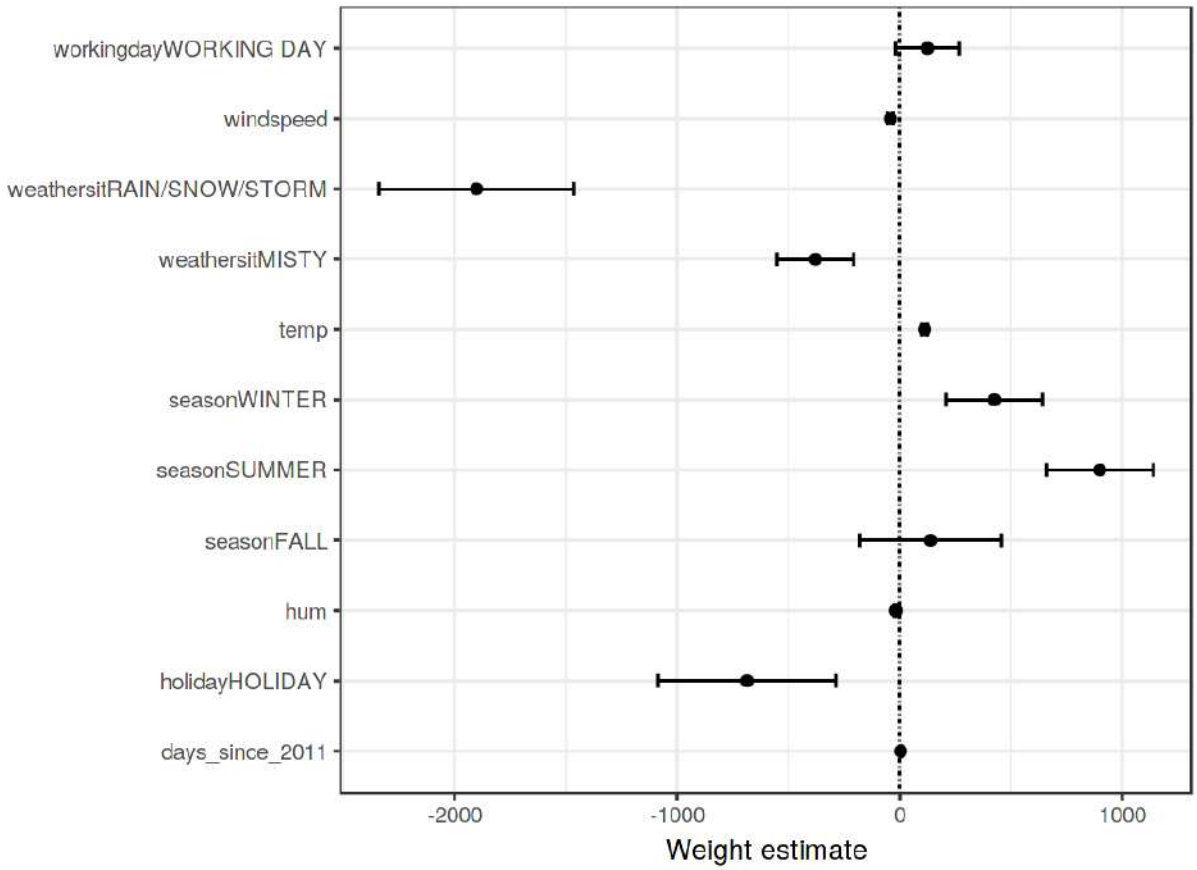
All the interpretations always come with the footnote that "all other features remain the same". This is because of the nature of linear regression models. The predicted target is a linear combination of the weighted features. The estimated linear equation is a hyperplane in the feature/target space (a simple line in the case of a single feature). The weights specify the slope (gradient) of the hyperplane in each direction. The good side is that the additivity isolates the interpretation of an individual feature effect from all other features. That is possible because all the feature effects (= weight times feature value) in the equation are combined with a plus. On the bad side of things, the interpretation ignores the joint distribution of the features. Increasing one feature, but not changing another, can lead to unrealistic or at least unlikely data points. For example increasing the number of rooms might be unrealistic without also increasing the size of a house.

Visual Interpretation

Various visualizations make the linear regression model easy and quick to grasp for humans.

Weight Plot

The information of the weight table (weight and variance estimates) can be visualized in a weight plot. The following plot shows the results from the previous linear regression model.



Weights are displayed as points and the 95% confidence intervals as lines.

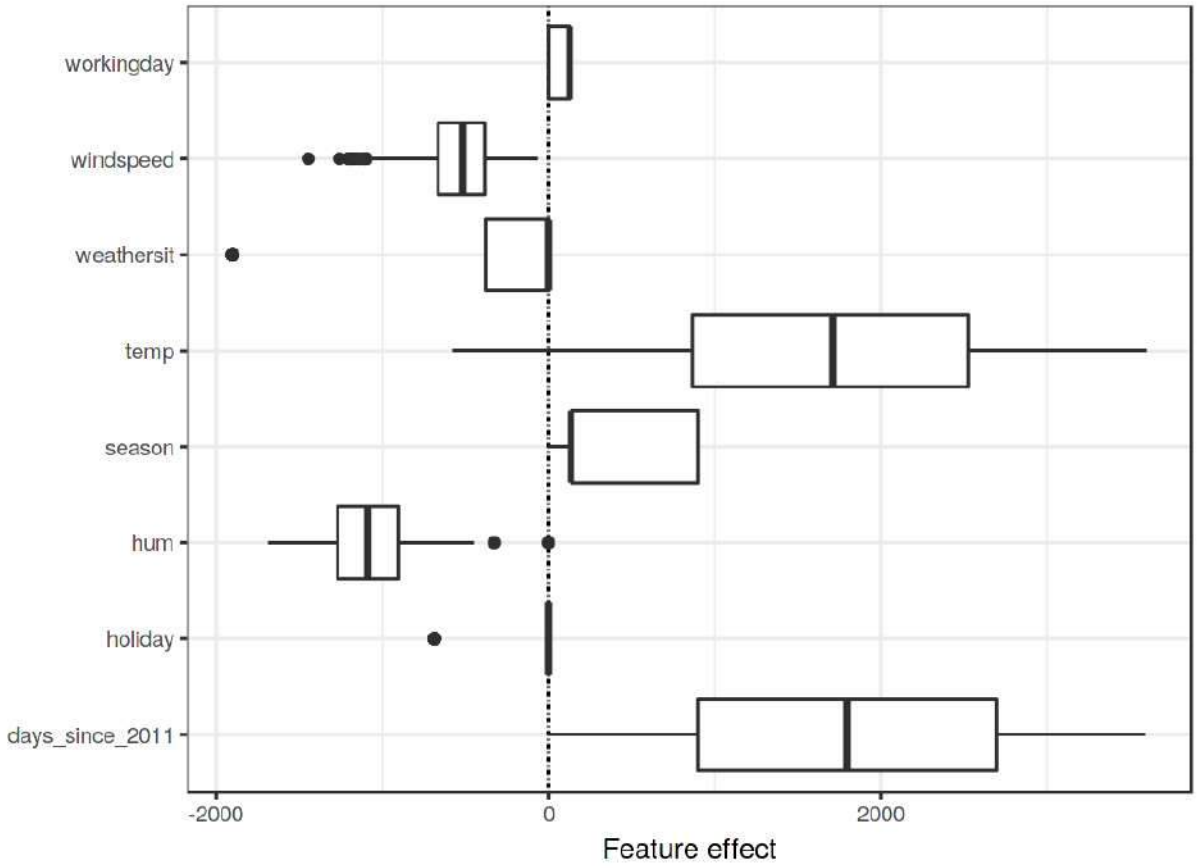
The weight plot shows that rainy/snowy/stormy weather has a strong negative effect on the predicted number of bikes. The weight of the working day feature is close to zero and zero is included in the 95% interval, which means that the effect is not statistically significant. Some confidence intervals are very short and the estimates are close to zero, yet the feature effects were statistically significant. Temperature is one such candidate. The problem with the weight plot is that the features are measured on different scales. While for the weather the estimated weight reflects the difference between good and rainy/stormy/snowy weather, for temperature it only reflects an increase of 1 degree Celsius. You can make the estimated weights more comparable by scaling the features (zero mean and standard deviation of one) before fitting the linear model.

Effect Plot

The weights of the linear regression model can be more meaningfully analyzed when they are multiplied by the actual feature values. The weights depend on the scale of the features and will be different if you have a feature that measures e.g. a person's height and you switch from meter to centimeter. The weight will change, but the actual effects in your data will not. It is also important to know the distribution of your feature in the data, because if you have a very low variance, it means that almost all instances have similar contribution from this feature. The effect plot can help you understand how much the combination of weight and feature contributes to the predictions in your data. Start by calculating the effects, which is the weight per feature times the feature value of an instance:

$$\text{effect}_j^{(i)} = w_j x_j^{(i)}$$

The effects can be visualized with boxplots. A box in a boxplot contains the effect range for half of your data (25% to 75% effect quantiles). The vertical line in the box is the median effect, i.e. 50% of the instances have a lower and the other half a higher effect on the prediction. The horizontal lines extend to $\pm 1.58 \text{IQR}/\sqrt{n}$, with IQR being the inter quartile range (75% quantile minus 25% quantile). The dots are outliers. The categorical feature effects can be summarized in a single boxplot, compared to the weight plot, where each category has its own row.



The feature effect plot shows the distribution of effects (= feature value times feature weight) across the data per feature.

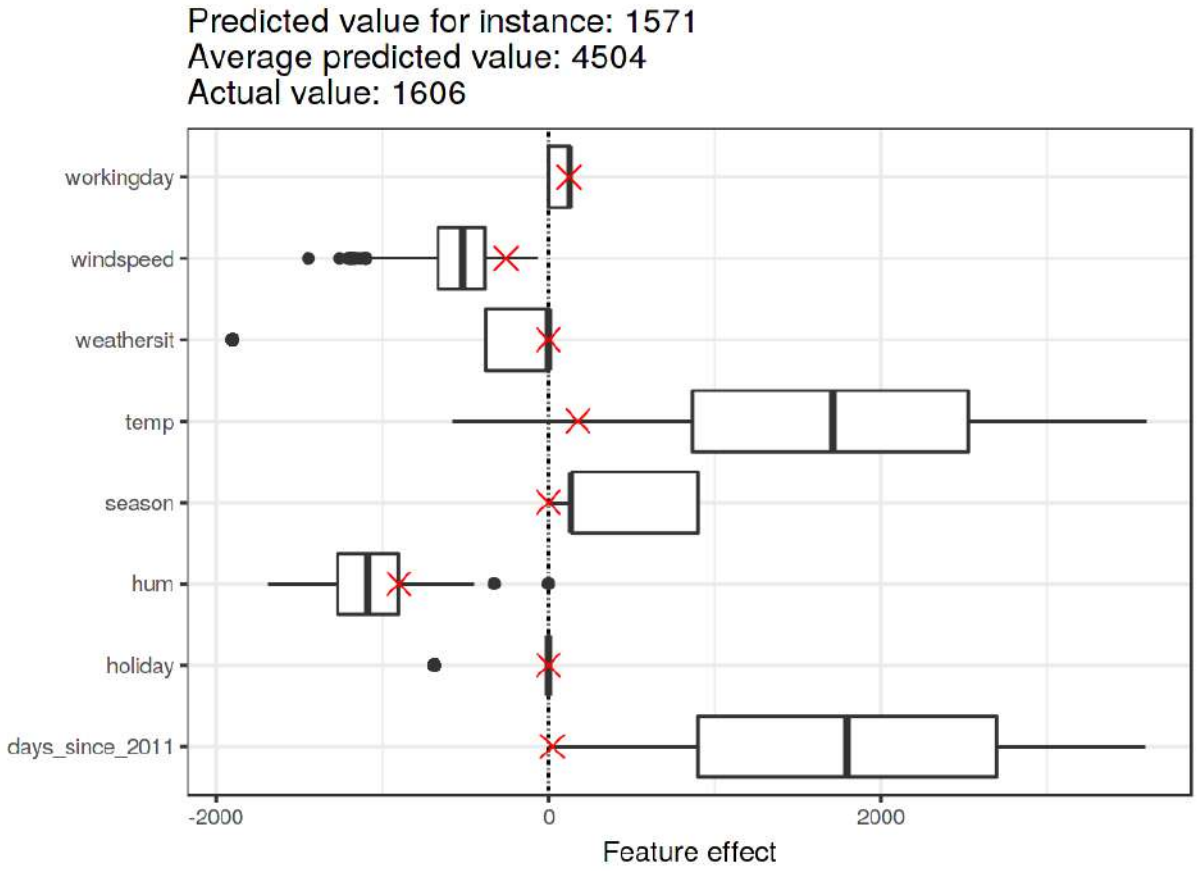
The largest contributions to the expected number of rented bicycles comes from the temperature feature and the days feature, which captures the trend of bike rentals over time. The temperature has a broad range of how much it contributes to the prediction. The day trend feature goes from zero to large positive contributions, because the first day in the dataset (01.01.2011) has a very small trend effect and the estimated weight for this feature is positive (4.93). This means that the effect increases with each day and is highest for the last day in the dataset (31.12.2012). Note that for effects with a negative weight, the instances with a positive effect are those that have a negative feature value. For example, days with a high negative effect of windspeed are the ones with high wind speeds.

Explain Individual Predictions

How much has each feature of an instance contributed to the prediction? This can be answered by computing the effects for this instance. An interpretation of instance-specific effects only makes sense in comparison to the distribution of the effect for each feature. We want to explain the prediction of the linear model for the 6-th instance from the bicycle dataset. The instance has the following feature values.

Feature	Value
season	SPRING
yr	2011
mnth	JAN
holiday	NO HOLIDAY
weekday	THU
workingday	WORKING DAY
weathersit	GOOD
temp	1.604356
hum	51.8261
windspeed	6.000868
cnt	1606
days_since_2011	5

To obtain the feature effects of this instance, we have to multiply its feature values by the corresponding weights from the linear regression model. For the value "WORKING DAY" of feature "workingday", the effect is, 124.9. For a temperature of 1.6 degrees Celsius, the effect is 177.6. We add these individual effects as crosses to the effect plot, which shows us the distribution of the effects in the data. This allows us to compare the individual effects with the distribution of effects in the data.



The effect plot for one instance shows the effect distribution and highlights the effects of the instance of interest.

If we average the predictions for the training data instances, we get an average of 4504. In comparison, the prediction of the 6-th instance is small, since only 1571 bicycle rents are predicted. The effect plot reveals the reason why. The boxplots show the distributions of the effects for all instances of the dataset, the red crosses show the effects for the 6-th instance. The 6-th instance has a low temperature effect because on this day the temperature was 2 degrees, which is low compared to most other days (and remember that the weight of the temperature feature is positive). Also, the effect of the trend feature "days_since_2011" is small compared to the other data instances because this instance is from early 2011 (5 days) and the trend feature also has a positive weight.

Encoding of Categorical Features

There are several ways to encode a categorical feature, and the choice influences the interpretation of the weights.

The standard in linear regression models is treatment coding, which is sufficient in most cases. Using different encodings boils down to creating different (design) matrices from a single column with the categorical feature. This section presents three different encodings, but there are many more. The example used has six instances and a categorical feature with three categories. For the first two

instances, the feature takes category A; for instances three and four, category B; and for the last two instances, category C.

Treatment coding

In treatment coding, the weight per category is the estimated difference in the prediction between the corresponding category and the reference category. The intercept of the linear model is the mean of the reference category (when all other features remain the same). The first column of the design matrix is the intercept, which is always 1. Column two indicates whether instance i is in category B, column three indicates whether it is in category C. There is no need for a column for category A, because then the linear equation would be overspecified and no unique solution for the weights can be found. It is sufficient to know that an instance is neither in category B or C.

Feature matrix: $\begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \\ 1 & 0 & 1 \end{pmatrix}$

Effect coding

The weight per category is the estimated y-difference from the corresponding category to the overall mean (given all other features are zero or the reference category). The first column is used to estimate the intercept. The weight β_0 associated with the intercept represents the overall mean and β_1, the weight for column two, is the difference between the overall mean and category B. The total effect of category B is $\beta_0 + \beta_1$. The interpretation for category C is equivalent. For the reference category A, $-(\beta_1 + \beta_2)$ is the difference to the overall mean and $\beta_0 - (\beta_1 + \beta_2)$ the overall effect.

Feature matrix: $\begin{pmatrix} 1 & -1 & -1 \\ 1 & -1 & -1 \\ 1 & 1 & 0 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \\ 1 & 0 & 1 \end{pmatrix}$

Dummy coding

The β per category is the estimated mean value of y for each category (given all other feature values are zero or the reference category). Note that the intercept has been omitted here so that a unique solution can be found for the linear model weights.

Feature matrix: $\begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix}$

If you want to dive a little deeper into the different encodings of categorical features, checkout this overview webpage[36] and this blog post[37].

Do Linear Models Create Good Explanations?

Judging by the attributes that constitute a good explanation, as presented in the Human-Friendly Explanations chapter, linear models do not create the best explanations. They are contrastive, but the reference instance is a data point where all numerical features are zero and the categorical features are at their reference categories. This is usually an artificial, meaningless instance that is unlikely to occur in your data or reality. There is an exception: If all numerical features are mean centered (feature minus mean of feature) and all categorical features are effect coded, the reference instance is the data point where all the features take on the mean feature value. This might also be a non-existent data point, but it might at least be more likely or more meaningful. In this case, the weights times the feature values (feature effects) explain the contribution to the predicted outcome contrastive to the "mean-instance". Another aspect of a good explanation is selectivity, which can be achieved in linear models by using less features or by training sparse linear models. But by default, linear models do not create selective explanations. Linear models create truthful explanations, as long as the linear equation is an appropriate model for the relationship between features and outcome. The more non-linearities and interactions there are, the less accurate the linear model will be and the less truthful the explanations become. Linearity makes the explanations more general and simpler. The linear nature of the model, I believe, is the main factor why people use linear models for explaining relationships.

Sparse Linear Models

The examples of the linear models that I have chosen all look nice and neat, do they not? But in reality you might not have just a handful of features, but hundreds or thousands. And your linear regression models? Interpretability goes downhill. You might even find yourself in a situation where there are more features than instances, and you cannot fit a standard linear model at all. The good news is that there are ways to introduce sparsity (= few features) into linear models.

Lasso

Lasso is an automatic and convenient way to introduce sparsity into the linear regression model. Lasso stands for "least absolute shrinkage and selection operator" and, when applied in a linear regression model, performs feature selection and regularization of the selected feature weights. Let us consider the minimization problem that the weights optimize:

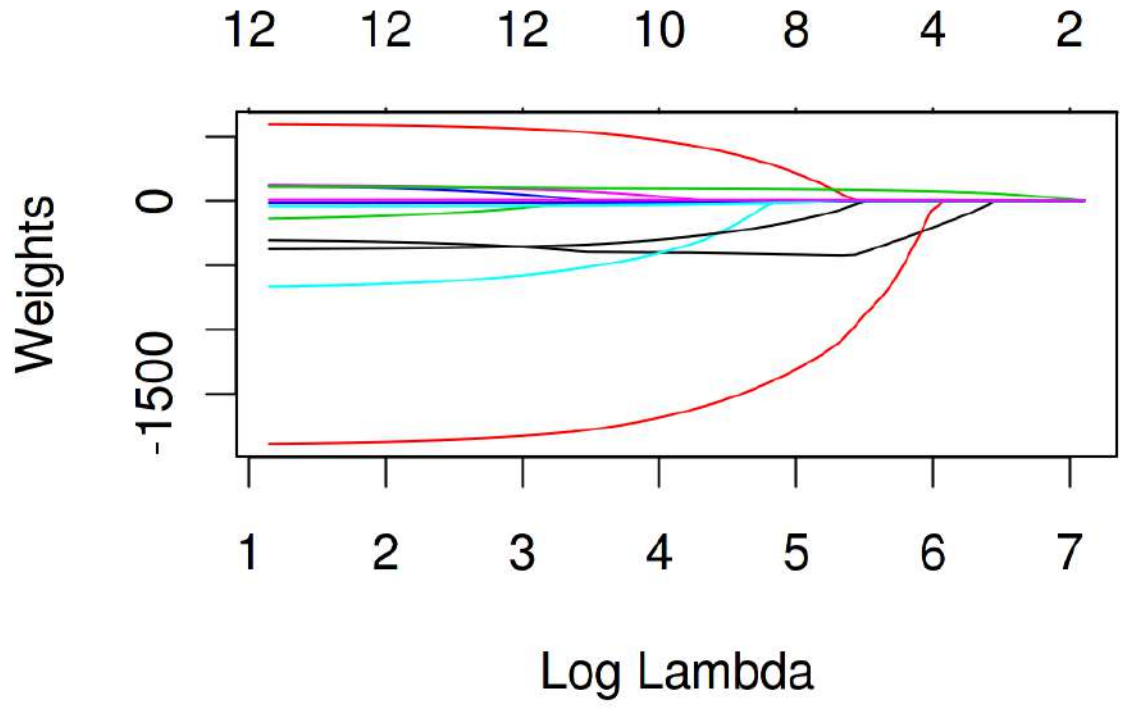
$$min_{\boldsymbol{\beta}} \left(\frac{1}{n} \sum_{i=1}^{n} (y^{(i)} - x_i^T \boldsymbol{\beta})^2 \right)$$

[36] http://stats.idre.ucla.edu/r/library/r-library-contrast-coding-systems-for-categorical-variables/
[37] http://heidiseibold.github.io/page7/

Lasso adds a term to this optimization problem.

$$min_{\boldsymbol{\beta}} \left(\frac{1}{n} \sum_{i=1}^{n} (y^{(i)} - x_i^T \boldsymbol{\beta})^2 + \lambda ||\boldsymbol{\beta}||_1 \right)$$

The term $||\boldsymbol{\beta}||_1$, the L1-norm of the feature vector, leads to a penalization of large weights. Since the L1-norm is used, many of the weights receive an estimate of 0 and the others are shrunk. The parameter lambda (λ) controls the strength of the regularizing effect and is usually tuned by cross-validation. Especially when lambda is large, many weights become 0. The feature weights can be visualized as a function of the penalty term lambda. Each feature weight is represented by a curve in the following figure.



With increasing penalty of the weights, fewer and fewer features receive a non-zero weight estimate. These curves are also called regularization paths. The number above the plot is the number of non-zero weights.

What value should we choose for lambda? If you see the penalization term as a tuning parameter, then you can find the lambda that minimizes the model error with cross-validation. You can also consider lambda as a parameter to control the interpretability of the model. The larger the penalization, the fewer features are present in the model (because their weights are zero) and the better the model can be interpreted.

Example with Lasso

We will predict bicycle rentals using Lasso. We set the number of features we want to have in the model beforehand. Let us first set the number to 2 features:

	Weight
seasonSPRING	0.00
seasonSUMMER	0.00
seasonFALL	0.00
seasonWINTER	0.00
holidayHOLIDAY	0.00
workingdayWORKING DAY	0.00
weathersitMISTY	0.00
weathersitRAIN/SNOW/STORM	0.00
temp	52.33
hum	0.00
windspeed	0.00
days_since_2011	2.15

The first two features with non-zero weights in the Lasso path are temperature ("temp") and the time trend ("days_since_2011").

Now, let us select 5 features:

	Weight
seasonSPRING	-389.99
seasonSUMMER	0.00
seasonFALL	0.00
seasonWINTER	0.00
holidayHOLIDAY	0.00
workingdayWORKING DAY	0.00
weathersitMISTY	0.00
weathersitRAIN/SNOW/STORM	-862.27
temp	85.58
hum	-3.04
windspeed	0.00
days_since_2011	3.82

Note that the weights for "temp" and "days_since_2011" differ from the model with two features. The reason for this is that by decreasing lambda even features that are already "in" the model are penalized less and may get a larger absolute weight. The interpretation of the Lasso weights corresponds to the interpretation of the weights in the linear regression model. You only need to pay attention to whether the features are standardized or not, because this affects the weights. In this example, the features were standardized by the software, but the weights were automatically transformed back for us to match the original feature scales.

Other methods for sparsity in linear models

A wide spectrum of methods can be used to reduce the number of features in a linear model.

Pre-processing methods:

- Manually selected features: You can always use expert knowledge to select or discard some features. The big drawback is that it cannot be automated and you need to have access to someone who understands the data.
- Univariate selection: An example is the correlation coefficient. You only consider features that exceed a certain threshold of correlation between the feature and the target. The disadvantage is that it only considers the features individually. Some features might not show a correlation until the linear model has accounted for some other features. Those ones you will miss with univariate selection methods.

Step-wise methods:

- Forward selection: Fit the linear model with one feature. Do this with each feature. Select the model that works best (e.g. highest R-squared). Now again, for the remaining features, fit different versions of your model by adding each feature to your current best model. Select the one that performs best. Continue until some criterion is reached, such as the maximum number of features in the model.
- Backward selection: Similar to forward selection. But instead of adding features, start with the model that contains all features and try out which feature you have to remove to get the highest performance increase. Repeat this until some stopping criterion is reached.

I recommend using Lasso, because it can be automated, considers all features simultaneously, and can be controlled via lambda. It also works for the logistic regression model for classification.

Advantages

The modeling of the predictions as a **weighted sum** makes it transparent how predictions are produced. And with Lasso we can ensure that the number of features used remains small.

Many people use linear regression models. This means that in many places it is **accepted** for predictive modeling and doing inference. There is a **high level of collective experience and expertise**, including teaching materials on linear regression models and software implementations. Linear regression can be found in R, Python, Java, Julia, Scala, Javascript, ...

Mathematically, it is straightforward to estimate the weights and you have a **guarantee to find optimal weights** (given all assumptions of the linear regression model are met by the data).

Together with the weights you get confidence intervals, tests, and solid statistical theory. There are also many extensions of the linear regression model (see chapter on GLM, GAM and more).

Disadvantages

Linear regression models can only represent linear relationships, i.e. a weighted sum of the input features. Each **nonlinearity or interaction has to be hand-crafted** and explicitly given to the model as an input feature.

Linear models are also often **not that good regarding predictive performance**, because the relationships that can be learned are so restricted and usually oversimplify how complex reality is.

The interpretation of a weight **can be unintuitive** because it depends on all other features. A feature with high positive correlation with the outcome y and another feature might get a negative weight in the linear model, because, given the other correlated feature, it is negatively correlated with y in the high-dimensional space. Completely correlated features make it even impossible to find a unique solution for the linear equation. An example: You have a model to predict the value of a house and have features like number of rooms and size of the house. House size and number of rooms are highly correlated: the bigger a house is, the more rooms it has. If you take both features into a linear model, it might happen, that the size of the house is the better predictor and gets a large positive weight. The number of rooms might end up getting a negative weight, because, given that a house has the same size, increasing the number of rooms could make it less valuable or the linear equation becomes less stable, when the correlation is too strong.

Logistic Regression

Logistic regression models the probabilities for classification problems with two possible outcomes. It's an extension of the linear regression model for classification problems.

What is Wrong with Linear Regression for Classification?

The linear regression model can work well for regression, but fails for classification. Why is that? In case of two classes, you could label one of the classes with 0 and the other with 1 and use linear regression. Technically it works and most linear model programs will spit out weights for you. But there are a few problems with this approach:

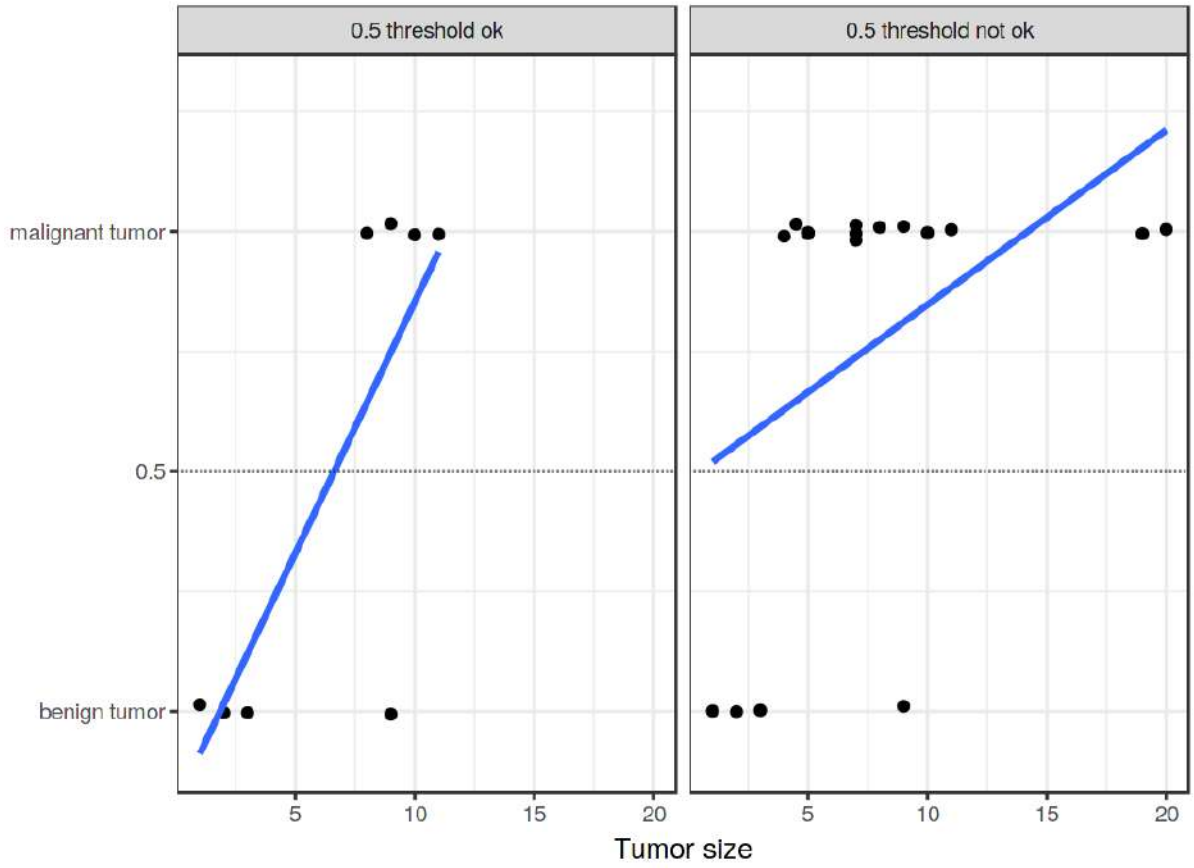
A linear model does not output probabilities, but it treats the classes as numbers (0 and 1) and fits the best hyperplane (for a single feature, it is a line) that minimizes the distances between the points and the hyperplane. So it simply interpolates between the points, and you cannot interpret it as probabilities.

A linear model also extrapolates and gives you values below zero and above one. This is a good sign that there might be a smarter approach to classification.

Since the predicted outcome is not a probability, but a linear interpolation between points, there is no meaningful threshold at which you can distinguish one class from the other. A good illustration of this issue has been given on Stackoverflow[38].

Linear models do not extend to classification problems with multiple classes. You would have to start labeling the next class with 2, then 3, and so on. The classes might not have any meaningful order, but the linear model would force a weird structure on the relationship between the features and your class predictions. The higher the value of a feature with a positive weight, the more it contributes to the prediction of a class with a higher number, even if classes that happen to get a similar number are not closer than other classes.

[38] https://stats.stackexchange.com/questions/22381/why-not-approach-classification-through-regression



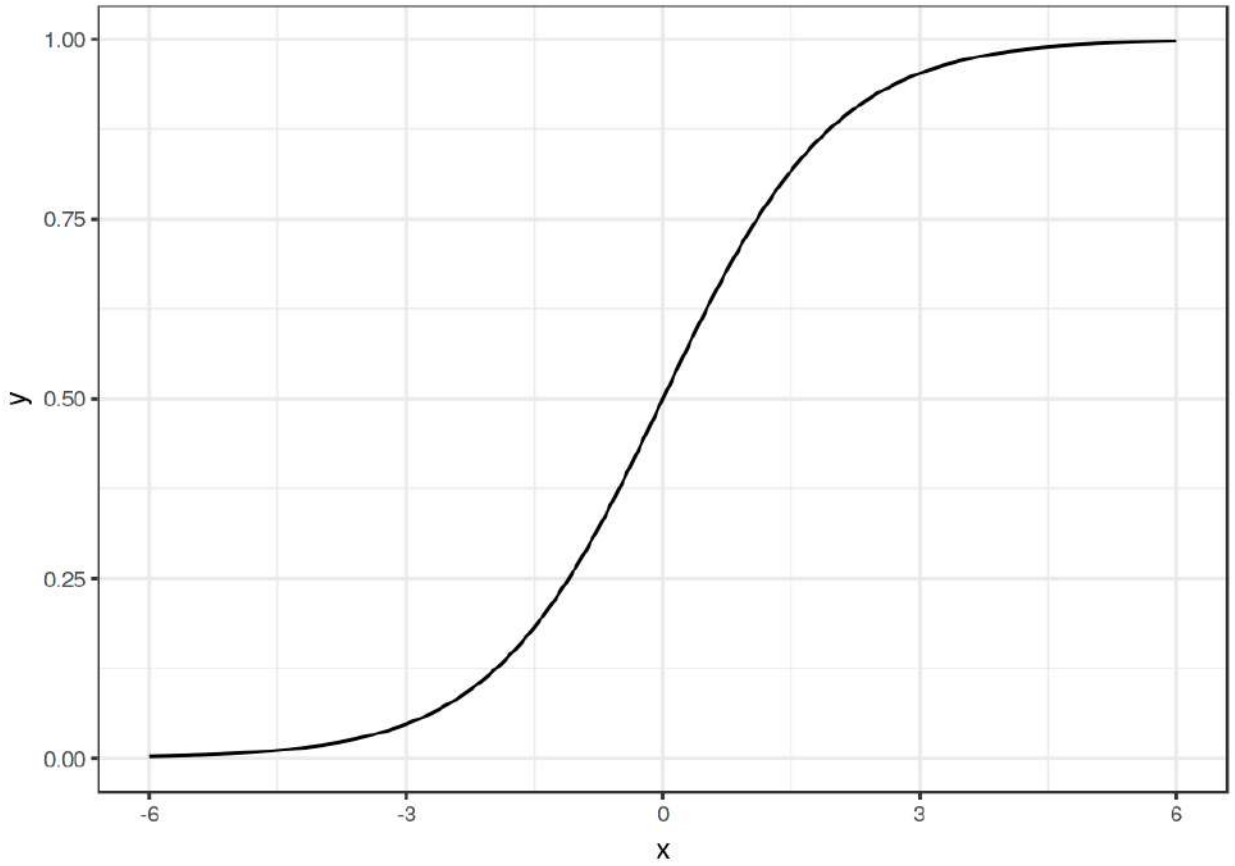
A linear model classifies tumors as malignant (1) or benign (0) given their size. The lines show the prediction of the linear model. For the data on the left, we can use 0.5 as classification threshold. After introducing a few more malignant tumor cases, the regression line shifts and a threshold of 0.5 no longer separates the classes. Points are slightly jittered to reduce over-plotting.

Theory

A solution for classification is logistic regression. Instead of fitting a straight line or hyperplane, the logistic regression model uses the logistic function to squeeze the output of a linear equation between 0 and 1. The logistic function is defined as:

$$\text{logistic}(\eta) = \frac{1}{1 + exp(-\eta)}$$

And it looks like this:



The logistic function. It outputs numbers between 0 and 1. At input 0, it outputs 0.5.

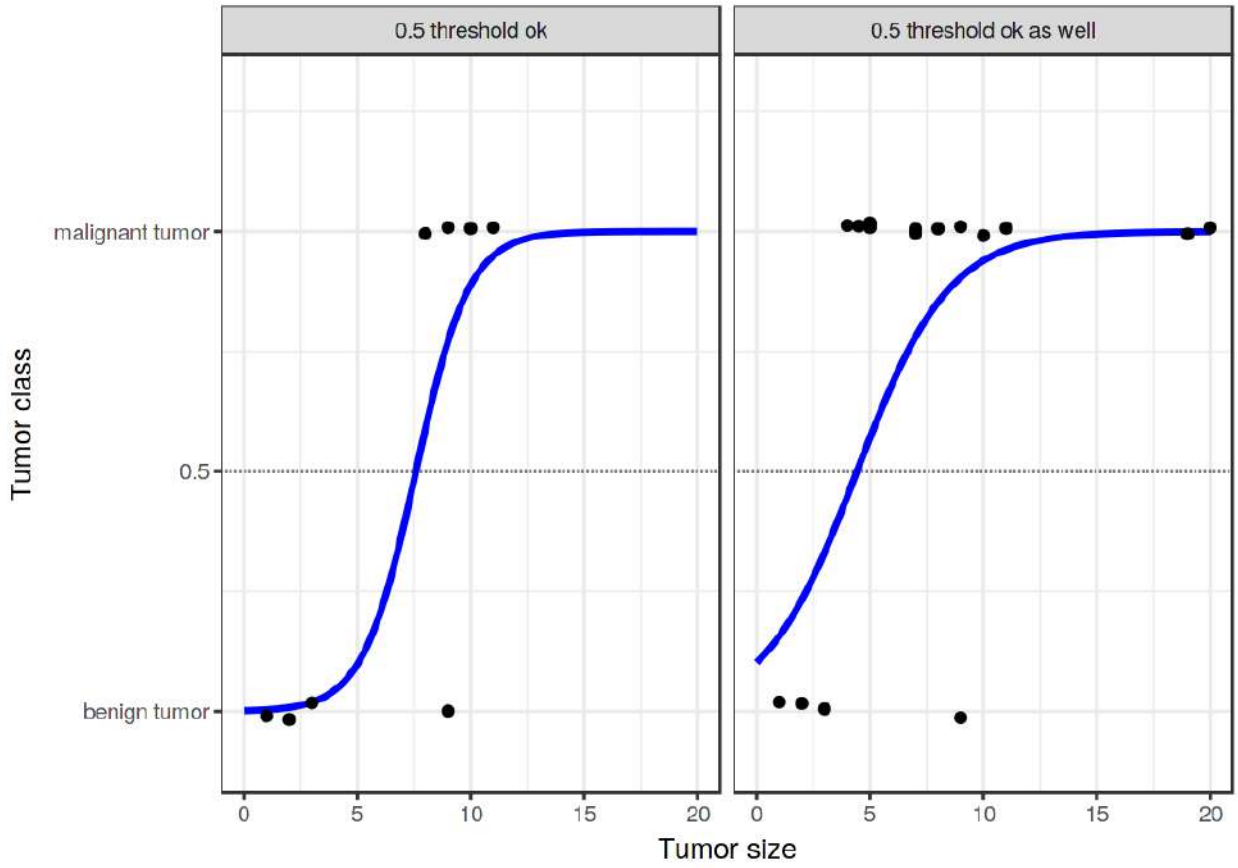
The step from linear regression to logistic regression is kind of straightforward. In the linear regression model, we have modelled the relationship between outcome and features with a linear equation:

$$\hat{y}^{(i)} = \beta_0 + \beta_1 x_1^{(i)} + \ldots + \beta_p x_p^{(i)}$$

For classification, we prefer probabilities between 0 and 1, so we wrap the right side of the equation into the logistic function. This forces the output to assume only values between 0 and 1.

$$P(y^{(i)} = 1) = \frac{1}{1 + exp(-(\beta_0 + \beta_1 x_1^{(i)} + \ldots + \beta_p x_p^{(i)}))}$$

Let us revisit the tumor size example again. But instead of the linear regression model, we use the logistic regression model:



The logistic regression model finds the correct decision boundary between malignant and benign depending on tumor size. The blue line is the logistic function shifted and squeezed to fit the data.

Classification works better with logistic regression and we can use 0.5 as a threshold in both cases. The inclusion of additional points does not really affect the estimated curve.

Interpretation

The interpretation of the weights in logistic regression differs from the interpretation of the weights in linear regression, since the outcome in logistic regression is a probability between 0 and 1. The weights do not influence the probability linearly any longer. The weighted sum is transformed by the logistic function to a probability. Therefore we need to reformulate the equation for the interpretation so that only the linear term is on the right side of the formula.

$$log\left(\frac{P(y=1)}{1-P(y=1)}\right) = log\left(\frac{P(y=1)}{P(y=0)}\right) = \beta_0 + \beta_1 x_1 + \ldots + \beta_p x_p$$

We call the term in the log() function "odds" (probability of event divided by probability of no event) and wrapped in the logarithm it is called log odds.

This formula shows that the logistic regression model is a linear model for the log odds. Great! That does not sound helpful! With a little shuffling of the terms, you can figure out how the prediction changes when one of the features x_j is changed by 1 unit. To do this, we can first apply the exp() function to both sides of the equation:

$$\frac{P(y=1)}{1-P(y=1)} = odds = exp\left(\beta_0 + \beta_1 x_1 + \ldots + \beta_p x_p\right)$$

Then we compare what happens when we increase one of the feature values by 1. But instead of looking at the difference, we look at the ratio of the two predictions:

$$\frac{odds_{x_j+1}}{odds} = \frac{exp\left(\beta_0 + \beta_1 x_1 + \ldots + \beta_j(x_j+1) + \ldots + \beta_p x_p\right)}{exp\left(\beta_0 + \beta_1 x_1 + \ldots + \beta_j x_j + \ldots + \beta_p x_p\right)}$$

We apply the following rule:

$$\frac{exp(a)}{exp(b)} = exp(a-b)$$

And we remove many terms:

$$\frac{odds_{x_j+1}}{odds} = exp\left(\beta_j(x_j+1) - \beta_j x_j\right) = exp\left(\beta_j\right)$$

In the end, we have something as simple as exp() of a feature weight. A change in a feature by one unit changes the odds ratio (multiplicative) by a factor of exp(β_j). We could also interpret it this way: A change in x_j by one unit increases the log odds ratio by the value of the corresponding weight. Most people interpret the odds ratio because thinking about the log() of something is known to be hard on the brain. Interpreting the odds ratio already requires some getting used to. For example, if you have odds of 2, it means that the probability for y=1 is twice as high as y=0. If you have a weight (= log odds ratio) of 0.7, then increasing the respective feature by one unit multiplies the odds by exp(0.7) (approximately 2) and the odds change to 4. But usually you do not deal with the odds and interpret the weights only as the odds ratios. Because for actually calculating the odds you would need to set a value for each feature, which only makes sense if you want to look at one specific instance of your dataset.

These are the interpretations for the logistic regression model with different feature types:

- Numerical feature: If you increase the value of feature x_j by one unit, the estimated odds change by a factor of exp(β_j)
- Binary categorical feature: One of the two values of the feature is the reference category (in some languages, the one encoded in 0). Changing the feature x_j from the reference category to the other category changes the estimated odds by a factor of exp(β_j).
- Categorical feature with more than two categories: One solution to deal with multiple categories is one-hot-encoding, meaning that each category has its own column. You only need

L-1 columns for a categorical feature with L categories, otherwise it is over-parameterized. The L-th category is then the reference category. You can use any other encoding that can be used in linear regression. The interpretation for each category then is equivalent to the interpretation of binary features.

- Intercept β_0: When all numerical features are zero and the categorical features are at the reference category, the estimated odds are $\exp(\beta_0)$. The interpretation of the intercept weight is usually not relevant.

Example

We use the logistic regression model to predict cervical cancer based on some risk factors. The following table shows the estimate weights, the associated odds ratios, and the standard error of the estimates.

	Weight	Odds ratio	Std. Error
Intercept	2.91	18.36	0.32
Hormonal contraceptives y/n	0.12	1.12	0.30
Smokes y/n	-0.26	0.77	0.37
Num. of pregnancies	-0.04	0.96	0.10
Num. of diagnosed STDs	-0.82	0.44	0.33
Intrauterine device y/n	-0.62	0.54	0.40

Interpretation of a numerical feature ("Num. of diagnosed STDs"): An increase in the number of diagnosed STDs (sexually transmitted diseases) changes (decreases) the odds of cancer vs. no cancer by a factor of 0.44, when all other features remain the same. Keep in mind that correlation does not imply causation. No recommendation here to get STDs.

Interpretation of a categorical feature ("Hormonal contraceptives y/n"): For women using hormonal contraceptives, the odds for cancer vs. no cancer are by a factor of 1.12 higher, compared to women without hormonal contraceptives, given all other features stay the same.

Like in the linear model, the interpretations always come with the clause that 'all other features stay the same'.

Advantages and Disadvantages

Many of the pros and cons of the linear regression model also apply to the logistic regression model. Logistic regression has been widely used by many different people, but it struggles with its restrictive expressiveness (e.g. interactions must be added manually) and other models may have better predictive performance.

Another disadvantage of the logistic regression model is that the interpretation is more difficult because the interpretation of the weights is multiplicative and not additive.

Logistic regression can suffer from **complete separation**. If there is a feature that would perfectly separate the two classes, the logistic regression model can no longer be trained. This is because

the weight for that feature would not converge, because the optimal weight would be infinite. This is really a bit unfortunate, because such a feature is really useful. But you do not need machine learning if you have a simple rule that separates both classes. The problem of complete separation can be solved by introducing penalization of the weights or defining a prior probability distribution of weights.

On the good side, the logistic regression model is not only a classification model, but also gives you probabilities. This is a big advantage over models that can only provide the final classification. Knowing that an instance has a 99% probability for a class compared to 51% makes a big difference.

Logistic regression can also be extended from binary classification to multi-class classification. Then it is called Multinomial Regression.

Software

I used the `glm` function in R for all examples. You can find logistic regression in any programming language that can be used for performing data analysis, such as Python, Java, Stata, Matlab, …

GLM, GAM and more

The biggest strength but also the biggest weakness of the linear regression model is that the prediction is modeled as a weighted sum of the features. In addition, the linear model comes with many other assumptions. The bad news is (well, not really news) that all those assumptions are often violated in reality: The outcome given the features might have a non-Gaussian distribution, the features might interact and the relationship between the features and the outcome might be nonlinear. The good news is that the statistics community has developed a variety of modifications that transform the linear regression model from a simple blade into a Swiss knife.

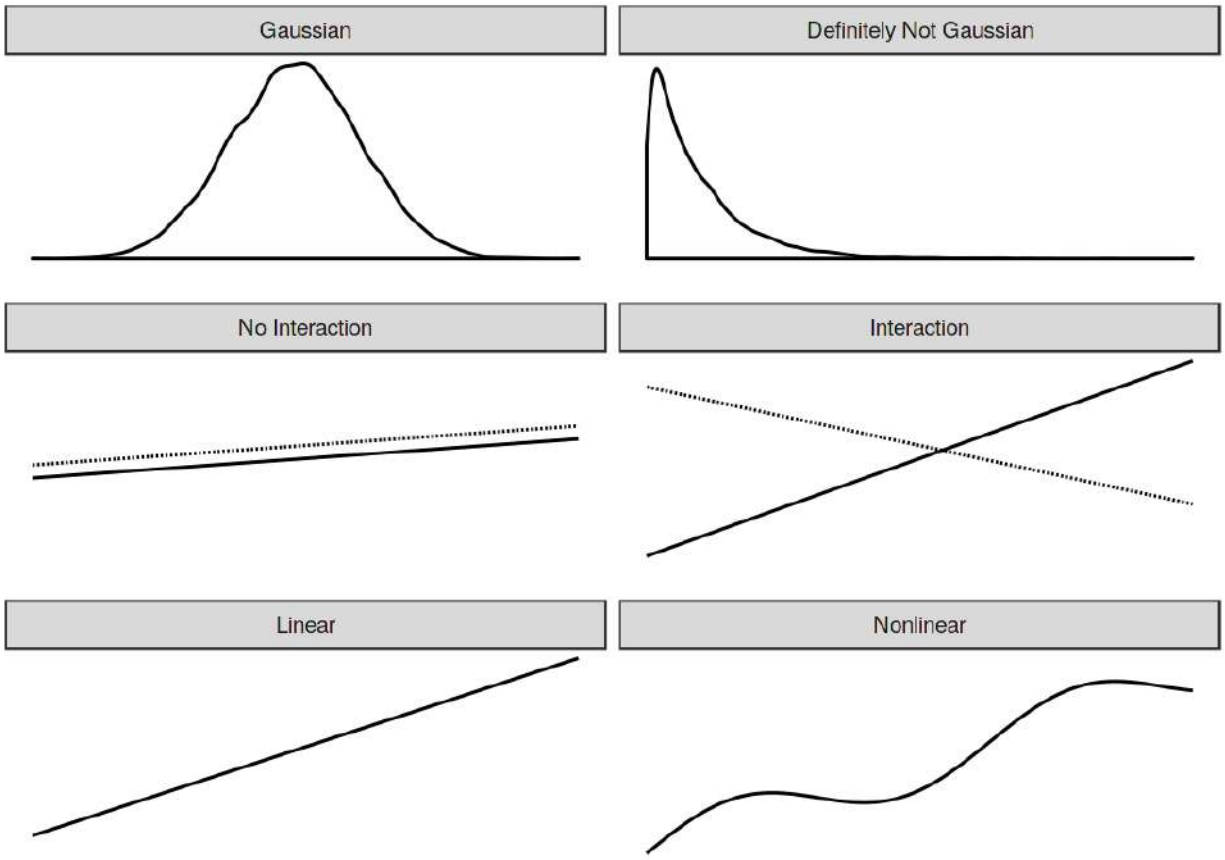
This chapter is definitely not your definite guide to extending linear models. Rather, it serves as an overview of extensions such as Generalized Linear Models (GLMs) and Generalized Additive Models (GAMs) and gives you a little intuition. After reading, you should have a solid overview of how to extend linear models. If you want to learn more about the linear regression model first, I suggest you read the chapter on linear regression models, if you have not already.

Let us remember the formula of a linear regression model:

$$y = \beta_0 + \beta_1 x_1 + \ldots + \beta_p x_p + \epsilon$$

The linear regression model assumes that the outcome y of an instance can be expressed by a weighted sum of its p features with an individual error ϵ that follows a Gaussian distribution. By forcing the data into this corset of a formula, we obtain a lot of model interpretability. The feature effects are additive, meaning no interactions, and the relationship is linear, meaning an increase of a feature by one unit can be directly translated into an increase/decrease of the predicted outcome. The linear model allows us to compress the relationship between a feature and the expected outcome into a single number, namely the estimated weight.

But a simple weighted sum is too restrictive for many real world prediction problems. In this chapter we will learn about three problems of the classical linear regression model and how to solve them. There are many more problems with possibly violated assumptions, but we will focus on the three shown in the following figure:



Three assumptions of the linear model (left side): Gaussian distribution of the outcome given the features, additivity (= no interactions) and linear relationship. Reality usually does not adhere to those assumptions (right side): Outcomes might have non-Gaussian distributions, features might interact and the relationship might be nonlinear.

There is a solution to all these problems:

Problem: The target outcome y given the features does not follow a Gaussian distribution.
Example: Suppose I want to predict how many minutes I will ride my bike on a given day. As features I have the type of day, the weather and so on. If I use a linear model, it could predict negative minutes because it assumes a Gaussian distribution which does not stop at 0 minutes. Also if I want to predict probabilities with a linear model, I can get probabilities that are negative or greater than 1.
Solution: Generalized Linear Models (GLMs).

Problem: The features interact.
Example: On average, light rain has a slight negative effect on my desire to go cycling. But in summer, during rush hour, I welcome rain, because then all the fair-weather cyclists stay at home and I have the bicycle paths for myself! This is an interaction between time and weather that cannot be captured by a purely additive model.
Solution: Adding interactions manually.

Problem: The true relationship between the features and y is not linear.
Example: Between 0 and 25 degrees Celsius, the influence of the temperature on my desire to ride

a bike could be linear, which means that an increase from 0 to 1 degree causes the same increase in cycling desire as an increase from 20 to 21. But at higher temperatures my motivation to cycle levels off and even decreases - I do not like to bike when it is too hot.
Solutions: Generalized Additive Models (GAMs); transformation of features.

The solutions to these three problems are presented in this chapter. Many further extensions of the linear model are omitted. If I attempted to cover everything here, the chapter would quickly turn into a book within a book about a topic that is already covered in many other books. But since you are already here, I have made a little problem plus solution overview for linear model extensions, which you can find at the end of the chapter. The name of the solution is meant to serve as a starting point for a search.

Non-Gaussian Outcomes - GLMs

The linear regression model assumes that the outcome given the input features follows a Gaussian distribution. This assumption excludes many cases: The outcome can also be a category (cancer vs. healthy), a count (number of children), the time to the occurrence of an event (time to failure of a machine) or a very skewed outcome with a few very high values (household income). The linear regression model can be extended to model all these types of outcomes. This extension is called **Generalized Linear Models** or **GLMs** for short. Throughout this chapter, I will use the name GLM for both the general framework and for particular models from that framework. The core concept of any GLM is: Keep the weighted sum of the features, but allow non-Gaussian outcome distributions and connect the expected mean of this distribution and the weighted sum through a possibly nonlinear function. For example, the logistic regression model assumes a Bernoulli distribution for the outcome and links the expected mean and the weighted sum using the logistic function.

The GLM mathematically links the weighted sum of the features with the mean value of the assumed distribution using the link function g, which can be chosen flexibly depending on the type of outcome.

$$g(E_Y(y|x)) = \beta_0 + \beta_1 x_1 + \ldots \beta_p x_p$$

GLMs consist of three components: The link function g, the weighted sum $X^T\beta$ (sometimes called linear predictor) and a probability distribution from the exponential family that defines E_Y.

The exponential family is a set of distributions that can be written with the same (parameterized) formula that includes an exponent, the mean and variance of the distribution and some other parameters. I will not go into the mathematical details because this is a very big universe of its own that I do not want to enter. Wikipedia has a neat list of distributions from the exponential family[39]. Any distribution from this list can be chosen for your GLM. Based on the type of the outcome you want to predict, choose a suitable distribution. Is the outcome a count of something (e.g. number of children living in a household)? Then the Poisson distribution could be a good choice. Is the outcome

[39] https://en.wikipedia.org/wiki/Exponential_family#Table_of_distributions

always positive (e.g. time between two events)? Then the exponential distribution could be a good choice.

Let us consider the classic linear model as a special case of a GLM. The link function for the Gaussian distribution in the classic linear model is simply the identity function. The Gaussian distribution is parameterized by the mean and the variance parameters. The mean describes the value that we expect on average and the variance describes how much the values vary around this mean. In the linear model, the link function links the weighted sum of the features to the mean of the Gaussian distribution.

Under the GLM framework, this concept generalizes to any distribution (from the exponential family) and arbitrary link functions. If y is a count of something, such as the number of coffees someone drinks on a certain day, we could model it with a GLM with a Poisson distribution and the natural logarithm as the link function:

$$ln(E_Y(y|x) = x^T\beta$$

The logistic regression model is also a GLM that assumes a Bernoulli distribution and uses the logistic function as the link function. The mean of the binomial distribution used in logistic regression is the probability that y is 1.

$$x^T\beta = ln\left(\frac{E_Y(y|x)}{1-E_Y(y|x)}\right) = ln\left(\frac{P(y=1|x)}{1-P(y=1|x)}\right)$$

And if we solve this equation to have P(y=1) on one side, we get the logistic regression formula:

$$P(y=1) = \frac{1}{1+exp(-x^T\beta)}$$

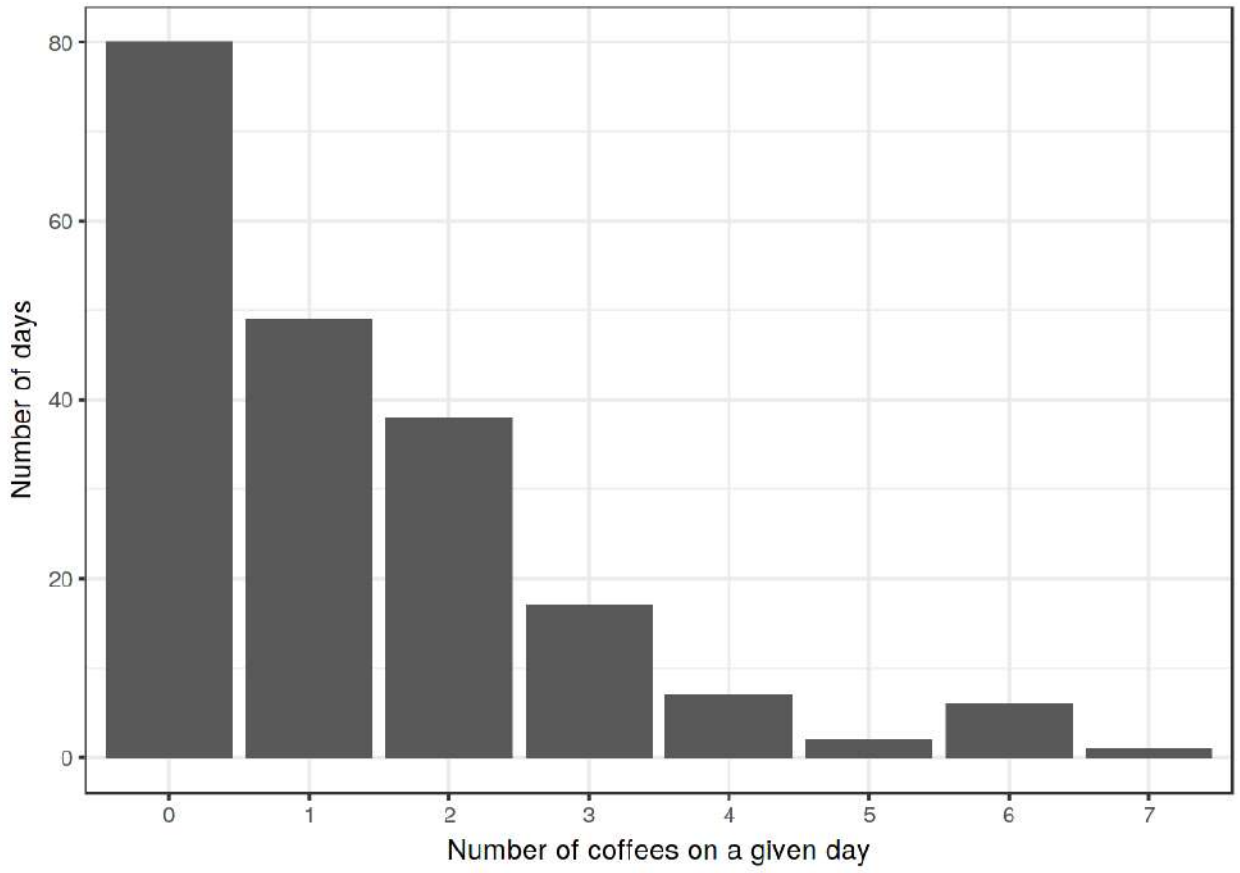
Each distribution from the exponential family has a canonical link function that can be derived mathematically from the distribution. The GLM framework makes it possible to choose the link function independently of the distribution. How to choose the right link function? There is no perfect recipe. You take into account knowledge about the distribution of your target, but also theoretical considerations and how well the model fits your actual data. For some distributions the canonical link function can lead to values that are invalid for that distribution. In the case of the exponential distribution, the canonical link function is the negative inverse, which can lead to negative predictions that are outside the domain of the exponential distribution. Since you can choose any link function, the simple solution is to choose another function that respects the domain of the distribution.

Examples

I have simulated a dataset on coffee drinking behavior to highlight the need for GLMs. Suppose you have collected data about your daily coffee drinking behavior. If you do not like coffee, pretend it is about tea. Along with number of cups, you record your current stress level on a scale of 1 to 10,

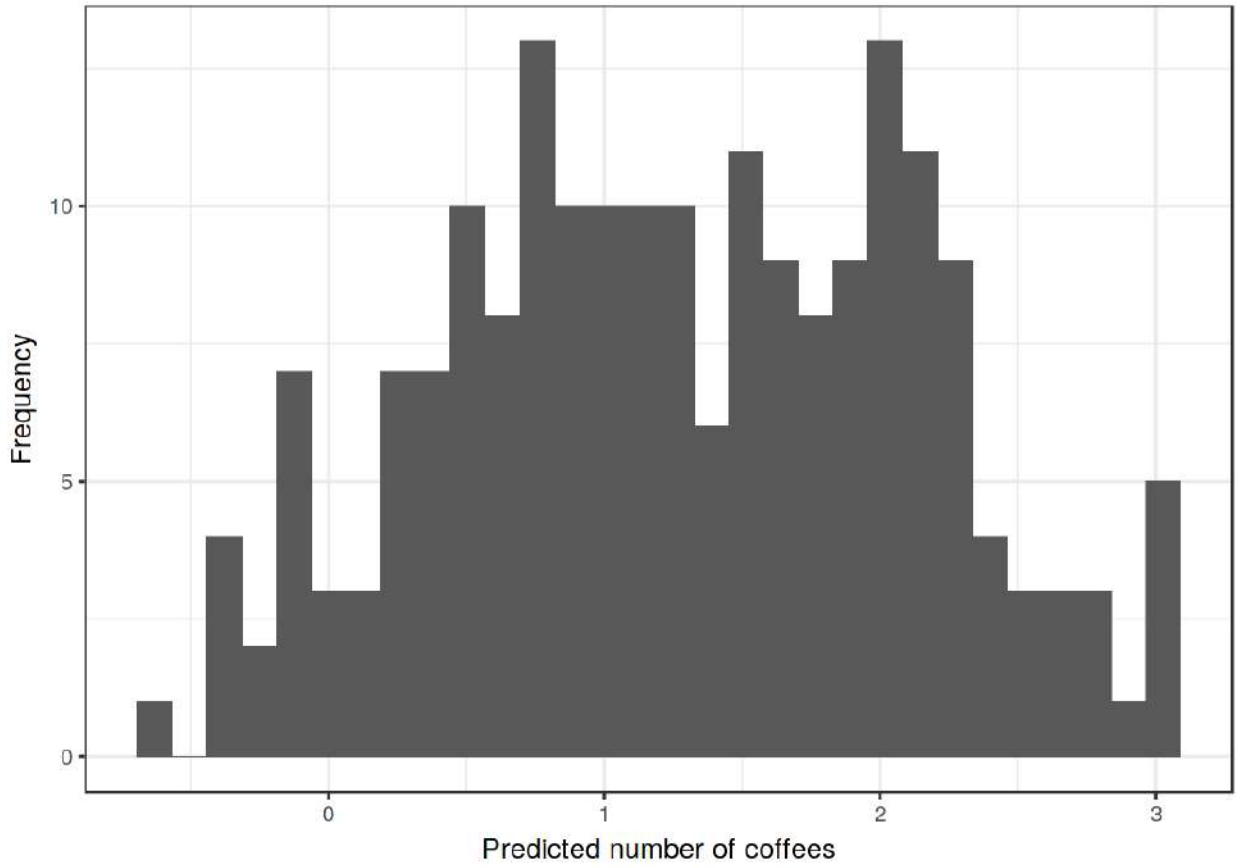
how well you slept the night before on a scale of 1 to 10 and whether you had to work on that day. The goal is to predict the number of coffees given the features stress, sleep and work. I simulated data for 200 days. Stress and sleep were drawn uniformly between 1 and 10 and work yes/no was drawn with a 50/50 chance (what a life!). For each day, the number of coffees was then drawn from a Poisson distribution, modelling the intensity λ (which is also the expected value of the Poisson distribution) as a function of the features sleep, stress and work. You can guess where this story will lead: *"Hey, let us model this data with a linear model ... Oh it does not work ... Let us try a GLM with Poisson distribution ... SURPRISE! Now it works!"*. I hope I did not spoil the story too much for you.

Let us look at the distribution of the target variable, the number of coffees on a given day:



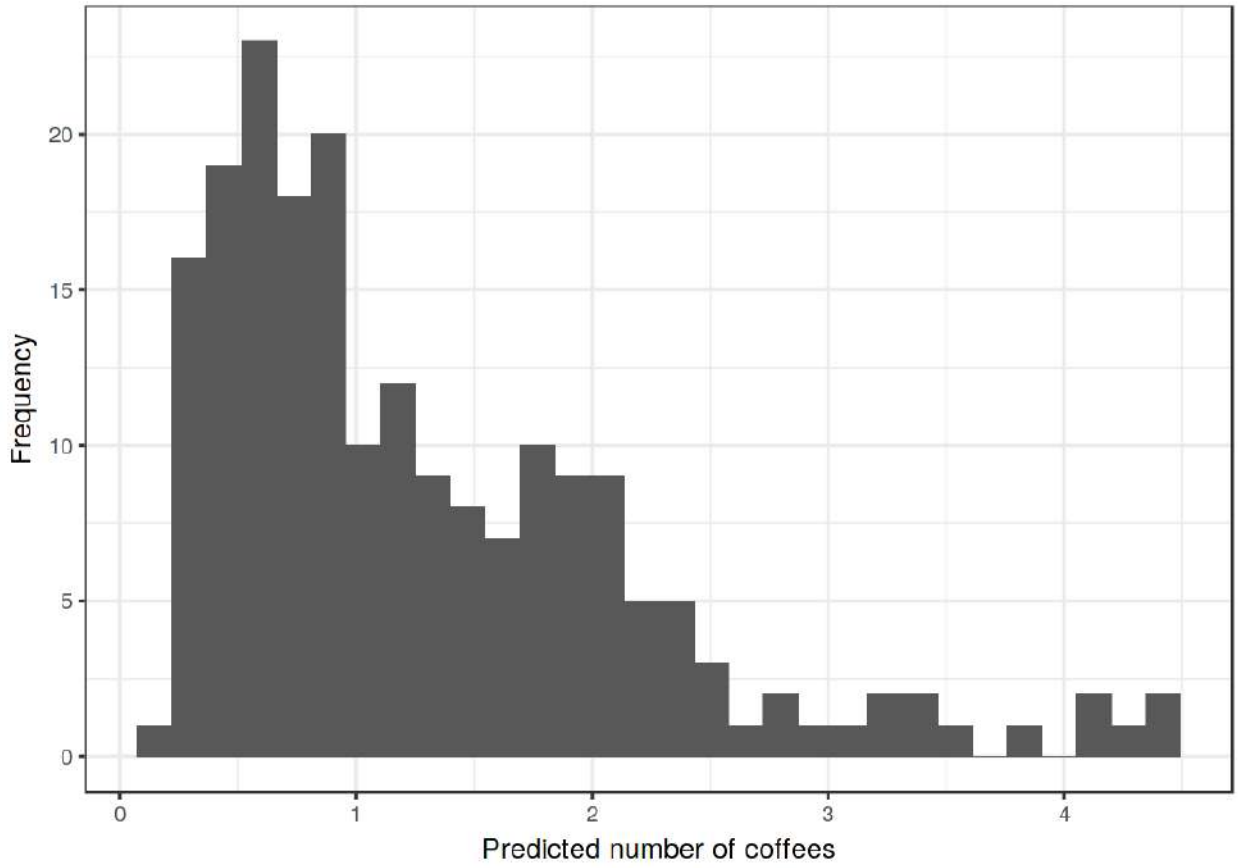
Simulated distribution of number of daily coffees for 200 days.

On 80 of the 200 days you had no coffee at all and on the most extreme day you had 7. Let us naively use a linear model to predict the number of coffees using sleep level, stress level and work yes/no as features. What can go wrong when we falsely assume a Gaussian distribution? A wrong assumption can invalidate the estimates, especially the confidence intervals of the weights. A more obvious problem is that the predictions do not match the "allowed" domain of the true outcome, as the following figure shows.



Predicted number of coffees dependent on stress, sleep and work. The linear model predicts negative values.

The linear model does not make sense, because it predicts negative number of coffees. This problem can be solved with Generalized Linear Models (GLMs). We can change the link function and the assumed distribution. One possibility is to keep the Gaussian distribution and use a link function that always leads to positive predictions such as the log-link (the inverse is the exp-function) instead of the identity function. Even better: We choose a distribution that corresponds to the data generating process and an appropriate link function. Since the outcome is a count, the Poisson distribution is a natural choice, along with the logarithm as link function. In this case, the data was even generated with the Poisson distribution, so the Poisson GLM is the perfect choice. The fitted Poisson GLM leads to the following distribution of predicted values:



Predicted number of coffees dependent on stress, sleep and work. The GLM with Poisson assumption and log link is an appropriate model for this dataset.

No negative amounts of coffees, looks much better now.

Interpretation of GLM weights

The assumed distribution together with the link function determines how the estimated feature weights are interpreted. In the coffee count example, I used a GLM with Poisson distribution and log link, which implies the following relationship between the features and the expected outcome.

$$ln(E(\text{coffees}|\text{stress}, \text{sleep}, \text{work})) = \beta_0 + \beta_{\text{stress}}x_{\text{stress}} + \beta_{\text{sleep}}x_{\text{sleep}} + \beta_{\text{work}}x_{\text{work}}$$

To interpret the weights we invert the link function so that we can interpret the effect of the features on the expected outcome and not on the logarithm of the expected outcome.

$$E(\text{coffees}|\text{stress}, \text{sleep}, \text{work}) = exp(\beta_0 + \beta_{\text{stress}}x_{\text{stress}} + \beta_{\text{sleep}}x_{\text{sleep}} + \beta_{\text{work}}x_{\text{work}})$$

Since all the weights are in the exponential function, the effect interpretation is not additive, but multiplicative, because exp(a + b) is exp(a) times exp(b). The last ingredient for the interpretation

is the actual weights of the toy example. The following table lists the estimated weights and exp(weights) together with the 95% confidence interval:

	weight	exp(weight) [2.5%, 97.5%]
(Intercept)	-0.12	0.89 [0.56, 1.38]
stress	0.11	1.11 [1.06, 1.17]
sleep	-0.16	0.85 [0.81, 0.89]
workYES	0.88	2.42 [1.87, 3.16]

Increasing the stress level by one point multiplies the expected number of coffees by the factor 1.11. Increasing the sleep quality by one point multiplies the expected number of coffees by the factor 0.85. The predicted number of coffees on a work day is on average 2.42 times the number of coffees on a day off. In summary, the more stress, the less sleep and the more work, the more coffee is consumed.

In this section you learned a little about Generalized Linear Models that are useful when the target does not follow a Gaussian distribution. Next, we look at how to integrate interactions between two features into the linear regression model.

Interactions

The linear regression model assumes that the effect of one feature is the same regardless of the values of the other features (= no interactions). But often there are interactions in the data. To predict the number of bicycles rented, there may be an interaction between temperature and whether it is a working day or not. Perhaps, when people have to work, the temperature does not influence the number of rented bikes much, because people will ride the rented bike to work no matter what happens. On days off, many people ride for pleasure, but only when it is warm enough. When it comes to rental bicycles, you might expect an interaction between temperature and working day.

How can we get the linear model to include interactions? Before you fit the linear model, add a column to the feature matrix that represents the interaction between the features and fit the model as usual. The solution is elegant in a way, since it does not require any change of the linear model, only additional columns in the data. In the working day and temperature example, we would add a new feature that has zeros for no-work days, otherwise it has the value of the temperature feature, assuming that working day is the reference category. Suppose our data looks like this:

work	temp
Y	25
N	12
N	30
Y	5

The data matrix used by the linear model looks slightly different. The following table shows what the data prepared for the model looks like if we do not specify any interactions. Normally, this transformation is performed automatically by any statistical software.

Intercept	workY	temp
1	1	25
1	0	12
1	0	30
1	1	5

The first column is the intercept term. The second column encodes the categorical feature, with 0 for the reference category and 1 for the other. The third column contains the temperature.

If we want the linear model to consider the interaction between temperature and the workingday feature, we have to add a column for the interaction:

Intercept	workY	temp	workY.temp
1	1	25	25
1	0	12	0
1	0	30	0
1	1	5	5

The new column "workY.temp" captures the interaction between the features working day (work) and temperature (temp). This new feature column is zero for an instance if the work feature is at the reference category ("N" for no working day), otherwise it assumes the values of the instances temperature feature. With this type of encoding, the linear model can learn a different linear effect of temperature for both types of days. This is the interaction effect between the two features. Without an interaction term, the combined effect of a categorical and a numerical feature can be described by a line that is vertically shifted for the different categories. If we include the interaction, we allow the effect of the numerical features (the slope) to have a different value in each category.

The interaction of two categorical features works similarly. We create additional features which represent combinations of categories. Here is some artificial data containing working day (work) and a categorical weather feature (wthr):

work	wthr
Y	Good
N	Bad
N	Ok
Y	Good

Next, we include interaction terms:

Intercept	workY	wthrGood	wthrOk	workY.wthrGood	workY.wthrOk
1	1	1	0	1	0
1	0	0	0	0	0
1	0	0	1	0	0
1	1	1	0	1	0

The first column serves to estimate the intercept. The second column is the encoded work feature. Columns three and four are for the weather feature, which requires two columns because you

need two weights to capture the effect for three categories, one of which is the reference category. The rest of the columns capture the interactions. For each category of both features (except for the reference categories), we create a new feature column that is 1 if both features have a certain category, otherwise 0.

For two numerical features, the interaction column is even easier to construct: We simply multiply both numerical features.

There are approaches to automatically detect and add interaction terms. One of them can be found in the RuleFit chapter. The RuleFit algorithm first mines interaction terms and then estimates a linear regression model including interactions.

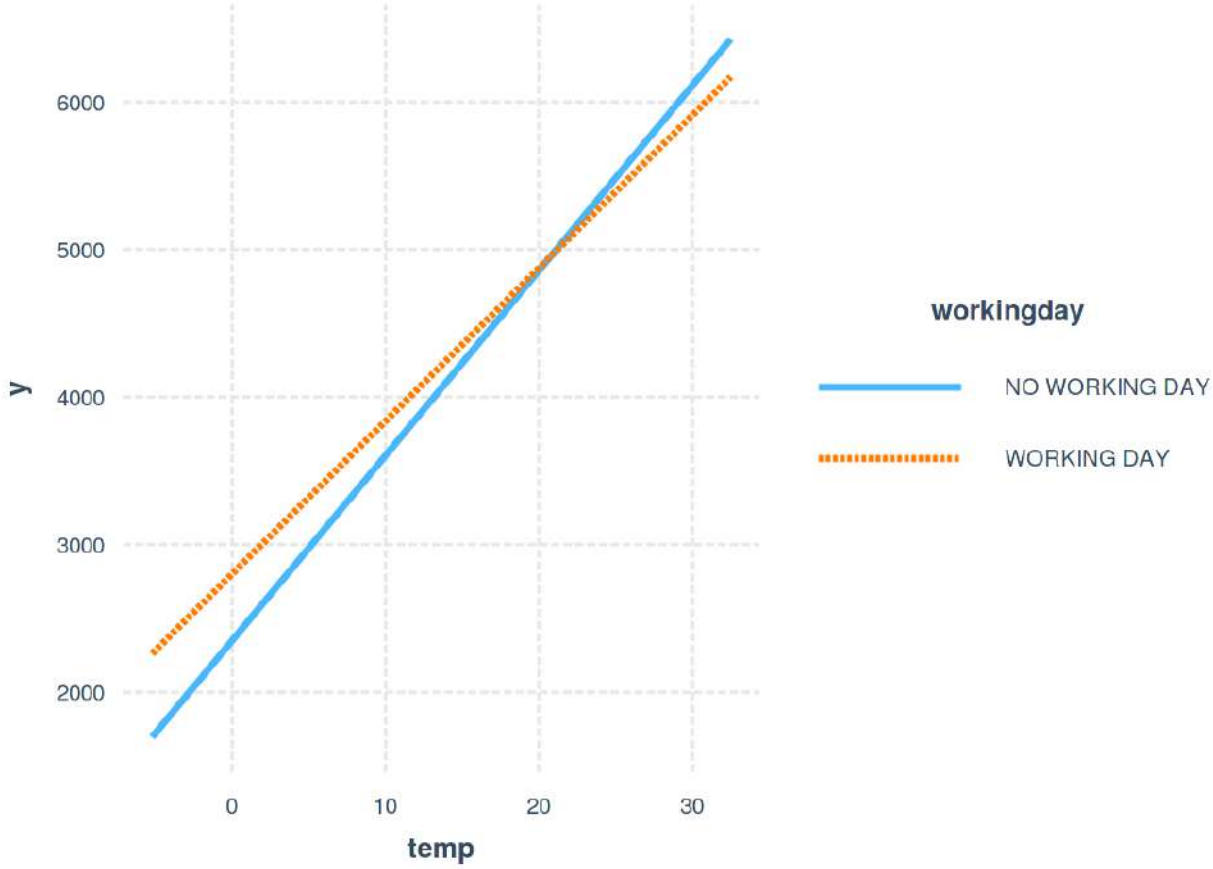
Example

Let us return to the bike rental prediction task which we have already modeled in the linear model chapter. This time, we additionally consider an interaction between the temperature and the working day feature. This results in the following estimated weights and confidence intervals.

	Weight	Std. Error	2.5%	97.5%
(Intercept)	2185.8	250.2	1694.6	2677.1
seasonSUMMER	893.8	121.8	654.7	1132.9
seasonFALL	137.1	161.0	-179.0	453.2
seasonWINTER	426.5	110.3	209.9	643.2
holidayHOLIDAY	-674.4	202.5	-1071.9	-276.9
workingdayWORKING DAY	451.9	141.7	173.7	730.1
weathersitMISTY	-382.1	87.2	-553.3	-211.0
weathersitRAIN/SNOW/STORM	-1898.2	222.7	-2335.4	-1461.0
temp	125.4	8.9	108.0	142.9
hum	-17.5	3.2	-23.7	-11.3
windspeed	-42.1	6.9	-55.5	-28.6
days_since_2011	4.9	0.2	4.6	5.3
workingdayWORKING DAY:temp	-21.8	8.1	-37.7	-5.9

The additional interaction effect is negative (-21.8) and differs significantly from zero, as shown by the 95% confidence interval, which does not include zero. By the way, the data are not iid, because days that are close to each other are not independent from each other. Confidence intervals might be misleading, just take it with a grain of salt. The interaction term changes the interpretation of the weights of the involved features. Does the temperature have a negative effect given it is a working day? The answer is no, even if the table suggests it to an untrained user. We cannot interpret the "workingdayWORKING DAY:temp" interaction weight in isolation, since the interpretation would be: "While leaving all other feature values unchanged, increasing the interaction effect of temperature for working day decreases the predicted number of bikes." But the interaction effect only adds to the main effect of the temperature. Suppose it is a working day and we want to know what would happen if the temperature were 1 degree warmer today. Then we need to sum both the weights for "temp" and "workingdayWORKING DAY:temp" to determine how much the estimate increases.

It is easier to understand the interaction visually. By introducing an interaction term between a categorical and a numerical feature, we get two slopes for the temperature instead of one. The temperature slope for days on which people do not have to work ('NO WORKING DAY') can be read directly from the table (125.4). The temperature slope for days on which people have to work ('WORKING DAY') is the sum of both temperature weights (125.4 -21.8 = 103.6). The intercept of the 'NO WORKING DAY'-line at temperature = 0 is determined by the intercept term of the linear model (2185.8). The intercept of the 'WORKING DAY'-line at temperature = 0 is determined by the intercept term + the effect of working day (2185.8 + 451.9 = 2637.7).



The effect (including interaction) of temperature and working day on the predicted number of bikes for a linear model. Effectively, we get two slopes for the temperature, one for each category of the working day feature.

Nonlinear Effects - GAMs

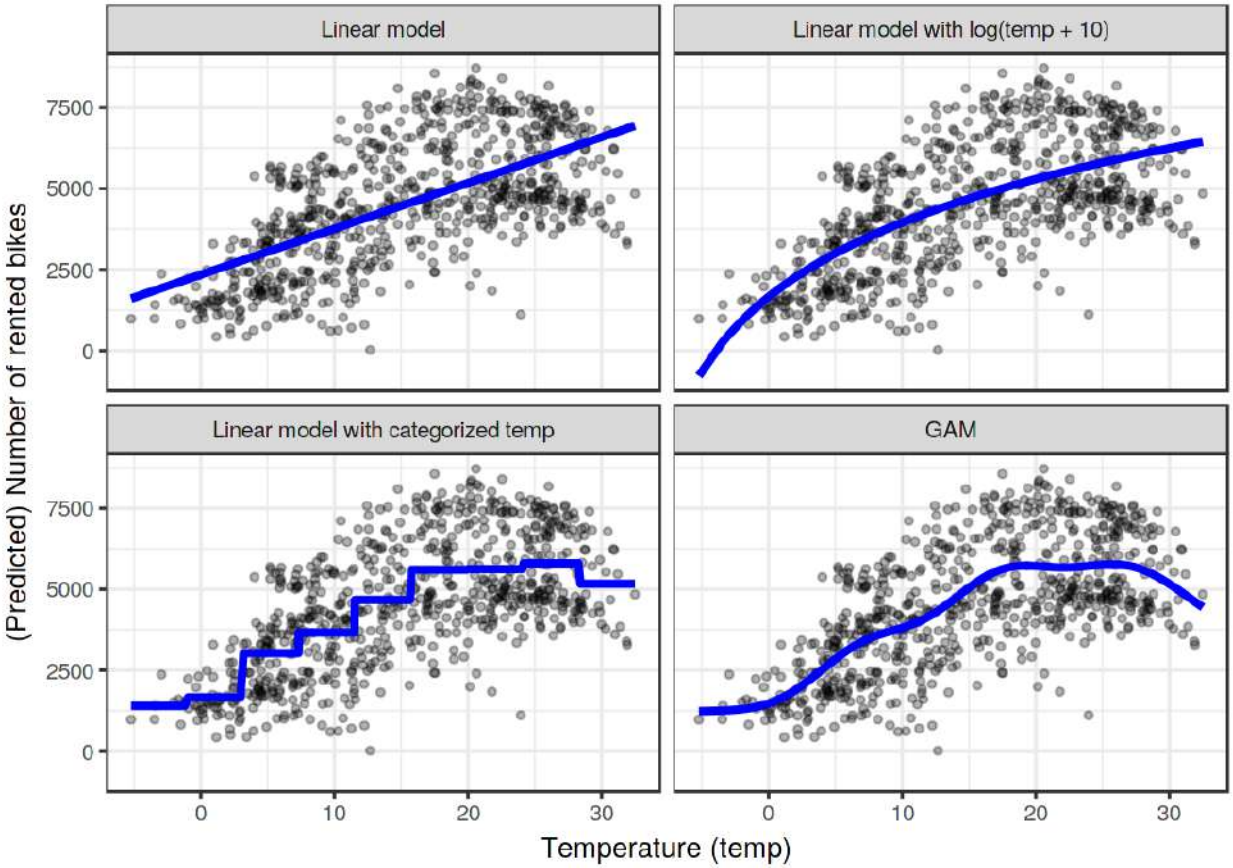
The world is not linear. Linearity in linear models means that no matter what value an instance has in a particular feature, increasing the value by one unit always has the same effect on the predicted outcome. Is it reasonable to assume that increasing the temperature by one degree at 10 degrees Celsius has the same effect on the number of rental bikes as increasing the temperature when it already has 40 degrees? Intuitively, one expects that increasing the temperature from 10 to 11 degrees Celsius has a positive effect on bicycle rentals and from 40 to 41 a negative effect, which is also the

case, as you will see, in many examples throughout the book. The temperature feature has a linear, positive effect on the number of rental bikes, but at some point it flattens out and even has a negative effect at high temperatures. The linear model does not care, it will dutifully find the best linear plane (by minimizing the Euclidean distance).

You can model nonlinear relationships using one of the following techniques:

- Simple transformation of the feature (e.g. logarithm)
- Categorization of the feature
- Generalized Additive Models (GAMs)

Before I go into the details of each method, let us start with an example that illustrates all three of them. I took the bike rental dataset and trained a linear model with only the temperature feature to predict the number of rental bikes. The following figure shows the estimated slope with: the standard linear model, a linear model with transformed temperature (logarithm), a linear model with temperature treated as categorical feature and using regression splines (GAM).



Predicting the number of rented bicycles using only the temperature feature. A linear model (top left) does not fit the data well. One solution is to transform the feature with e.g. the logarithm (top right), categorize it (bottom left), which is usually a bad decision, or use Generalized Additive Models that can automatically fit a smooth curve for temperature (bottom right).

Feature transformation

Often the logarithm of the feature is used as a transformation. Using the logarithm indicates that every 10-fold temperature increase has the same linear effect on the number of bikes, so changing from 1 degree Celsius to 10 degrees Celsius has the same effect as changing from 0.1 to 1 (sounds wrong). Other examples for feature transformations are the square root, the square function and the exponential function. Using a feature transformation means that you replace the column of this feature in the data with a function of the feature, such as the logarithm, and fit the linear model as usual. Some statistical programs also allow you to specify transformations in the call of the linear model. You can be creative when you transform the feature. The interpretation of the feature changes according to the selected transformation. If you use a log transformation, the interpretation in a linear model becomes: "If the logarithm of the feature is increased by one, the prediction is increased by the corresponding weight." When you use a GLM with a link function that is not the identity function, then the interpretation gets more complicated, because you have to incorporate both transformations into the interpretation (except when they cancel each other out, like log and exp, then the interpretation gets easier).

Feature categorization

Another possibility to achieve a nonlinear effect is to discretize the feature; turn it into a categorical feature. For example, you could cut the temperature feature into 20 intervals with the levels [-10, -5), [-5, 0), ... and so on. When you use the categorized temperature instead of the continuous temperature, the linear model would estimate a step function because each level gets its own estimate. The problem with this approach is that it needs more data, it is more likely to overfit and it is unclear how to discretize the feature meaningfully (equidistant intervals or quantiles? how many intervals?). I would only use discretization if there is a very strong case for it. For example, to make the model comparable to another study.

Generalized Additive Models (GAMs)

Why not 'simply' allow the (generalized) linear model to learn nonlinear relationships? That is the motivation behind GAMs. GAMs relax the restriction that the relationship must be a simple weighted sum, and instead assume that the outcome can be modeled by a sum of arbitrary functions of each feature. Mathematically, the relationship in a GAM looks like this:

$$g(E_Y(y|x)) = \beta_0 + f_1(x_1) + f_2(x_2) + \ldots + f_p(x_p)$$

The formula is similar to the GLM formula with the difference that the linear term $\beta_j x_j$ is replaced by a more flexible function $f_j(x_j)$. The core of a GAM is still a sum of feature effects, but you have the option to allow nonlinear relationships between some features and the output. Linear effects are also covered by the framework, because for features to be handled linearly, you can limit their $f_j(x_j)$ only to take the form of $x_j\beta_j$.

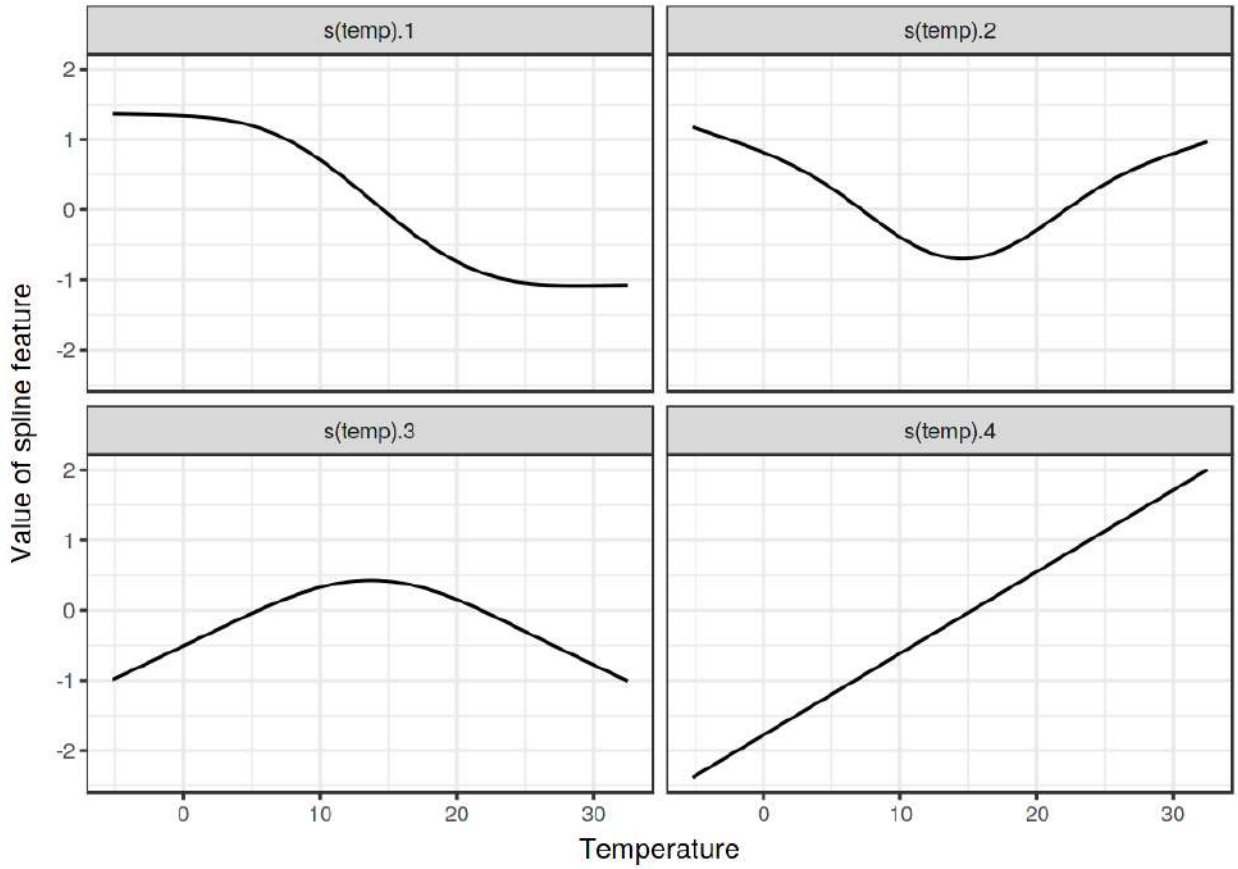
The big question is how to learn nonlinear functions. The answer is called "splines" or "spline functions". Splines are functions that can be combined in order to approximate arbitrary functions. A bit like stacking Lego bricks to build something more complex. There is a confusing number of

ways to define these spline functions. If you are interested in learning more about all the ways to define splines, I wish you good luck on your journey. I am not going to go into details here, I am just going to build an intuition. What personally helped me the most for understanding splines was to visualize the individual spline functions and to look into how the data matrix is modified. For example, to model the temperature with splines, we remove the temperature feature from the data and replace it with, say, 4 columns, each representing a spline function. Usually you would have more spline functions, I only reduced the number for illustration purposes. The value for each instance of these new spline features depends on the instances' temperature values. Together with all linear effects, the GAM then also estimates these spline weights. GAMs also introduce a penalty term for the weights to keep them close to zero. This effectively reduces the flexibility of the splines and reduces overfitting. A smoothness parameter that is commonly used to control the flexibility of the curve is then tuned via cross-validation. Ignoring the penalty term, nonlinear modeling with splines is fancy feature engineering.

In the example where we are predicting the number of bicycles with a GAM using only the temperature, the model feature matrix looks like this:

(Intercept)	s(temp).1	s(temp).2	s(temp).3	s(temp).4
1	0.93	-0.14	0.21	-0.83
1	0.83	-0.27	0.27	-0.72
1	1.32	0.71	-0.39	-1.63
1	1.32	0.70	-0.38	-1.61
1	1.29	0.58	-0.26	-1.47
1	1.32	0.68	-0.36	-1.59

Each row represents an individual instance from the data (one day). Each spline column contains the value of the spline function at the particular temperature values. The following figure shows how these spline functions look like:

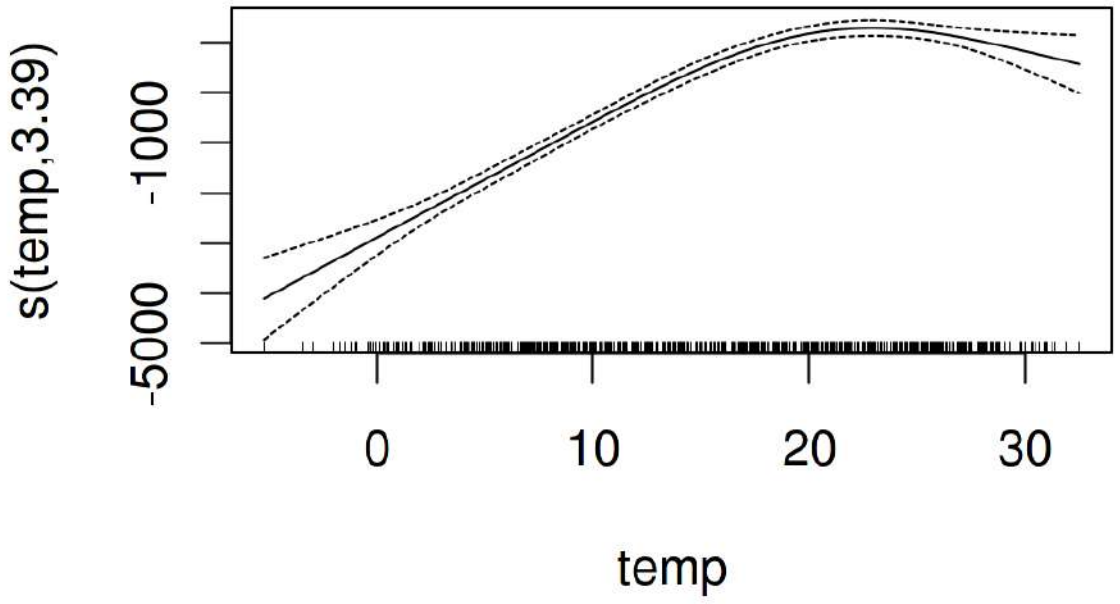


To smoothly model the temperature effect, we use 4 spline functions. Each temperature value is mapped to (here) 4 spline values. If an instance has a temperature of 30 ℃, the value for the first spline feature is -1, for the second 0.7, for the third -0.8 and for the 4th 1.7.

The GAM assigns weights to each temperature spline feature:

	weight
(Intercept)	4504.35
s(temp).1	-989.34
s(temp).2	740.08
s(temp).3	2309.84
s(temp).4	558.27

And the actual curve, which results from the sum of the spline functions weighted with the estimated weights, looks like this:



GAM feature effect of the temperature for predicting the number of rented bikes (temperature used as the only feature).

The interpretation of smooth effects requires a visual check of the fitted curve. Splines are usually centered around the mean prediction, so a point on the curve is the difference to the mean prediction. For example, at 0 degrees Celsius, the predicted number of bicycles is 3000 lower than the average prediction.

Advantages

All these extensions of the linear model are a bit of a universe in themselves. Whatever problems you face with linear models, **you will probably find an extension that fixes it**.

Most methods have been used for decades. For example, GAMs are almost 30 years old. Many researchers and practitioners from industry are very **experienced** with linear models and the methods are **accepted in many communities as status quo for modeling**.

In addition to making predictions, you can use the models to **do inference**, draw conclusions about the data – given the model assumptions are not violated. You get confidence intervals for weights, significance tests, prediction intervals and much more.

Statistical software usually has really good interfaces to fit GLMs, GAMs and more special linear models.

The opacity of many machine learning models comes from 1) a lack of sparseness, which means that many features are used, 2) features that are treated in a nonlinear fashion, which means you need more than a single weight to describe the effect, and 3) the modeling of interactions between the features. Assuming that linear models are highly interpretable but often underfit reality, the extensions described in this chapter offer a good way to achieve a **smooth transition to more flexible models**, while preserving some of the interpretability.

Disadvantages

As advantage I have said that linear models live in their own universe. The sheer **number of ways you can extend the simple linear model is overwhelming**, not just for beginners. Actually, there are multiple parallel universes, because many communities of researchers and practitioners have their own names for methods that do more or less the same thing, which can be very confusing.

Most modifications of the linear model make the model **less interpretable**. Any link function (in a GLM) that is not the identity function complicates the interpretation; interactions also complicate the interpretation; nonlinear feature effects are either less intuitive (like the log transformation) or can no longer be summarized by a single number (e.g. spline functions).

GLMs, GAMs and so on **rely on assumptions** about the data generating process. If those are violated, the interpretation of the weights is no longer valid.

The performance of tree-based ensembles like the random forest or gradient tree boosting is in many cases better than the most sophisticated linear models. This is partly my own experience and partly observations from the winning models on platforms like kaggle.com.

Software

All examples in this chapter were created using the R language. For GAMs, the `gam` package was used, but there are many others. R has an incredible number of packages to extend linear regression models. Unsurpassed by any other analytics language, R is home to every conceivable extension of the linear regression model extension. You will find implementations of e.g. GAMs in Python (such as pyGAM[40]), but theses implementation are not as mature.

Further Extensions

As promised, here is a list of problems you might encounter with linear models, along with the name of a solution for this problem that you can copy and paste into your favorite search engine.

[40] https://github.com/dswah/pyGAM

My data violates the assumption of being independent and identically distributed (iid).
For example, repeated measurements on the same patient.
Search for **mixed models** or **generalized estimating equations**.

My model has heteroscedastic errors.
For example, when predicting the value of a house, the model errors are usually higher in expensive houses, which violates the homoscedasticity of the linear model.
Search for **robust regression**.

I have outliers that strongly influence my model.
Search for **robust regression**.

I want to predict the time until an event occurs.
Time-to-event data usually comes with censored measurements, which means that for some instances there was not enough time to observe the event. For example, a company wants to predict the failure of its ice machines, but only has data for two years. Some machines are still intact after two years, but might fail later.
Search for **parametric survival models**, **cox regression**, **survival analysis**.

My outcome to predict is a category.
If the outcome has two categories use a logistic regression model, which models the probability for the categories.
If you have more categories, search for **multinomial regression**.
Logistic regression and multinomial regression are both GLMs.

I want to predict ordered categories.
For example school grades.
Search for **proportional odds model**.

My outcome is a count (like number of children in a family).
Search for **Poisson regression**.
The Poisson model is also a GLM. You might also have the problem that the count value of 0 is very frequent.
Search for **zero-inflated Poisson regression**, **hurdle model**.

I am not sure what features need to be included in the model to draw correct causal conclusions.
For example, I want to know the effect of a drug on the blood pressure. The drug has a direct effect on some blood value and this blood value affects the outcome. Should I include the blood value into the regression model?
Search for **causal inference**, **mediation analysis**.

I have missing data.
Search for **multiple imputation**.

I want to integrate prior knowledge into my models.
Search for **Bayesian inference**.

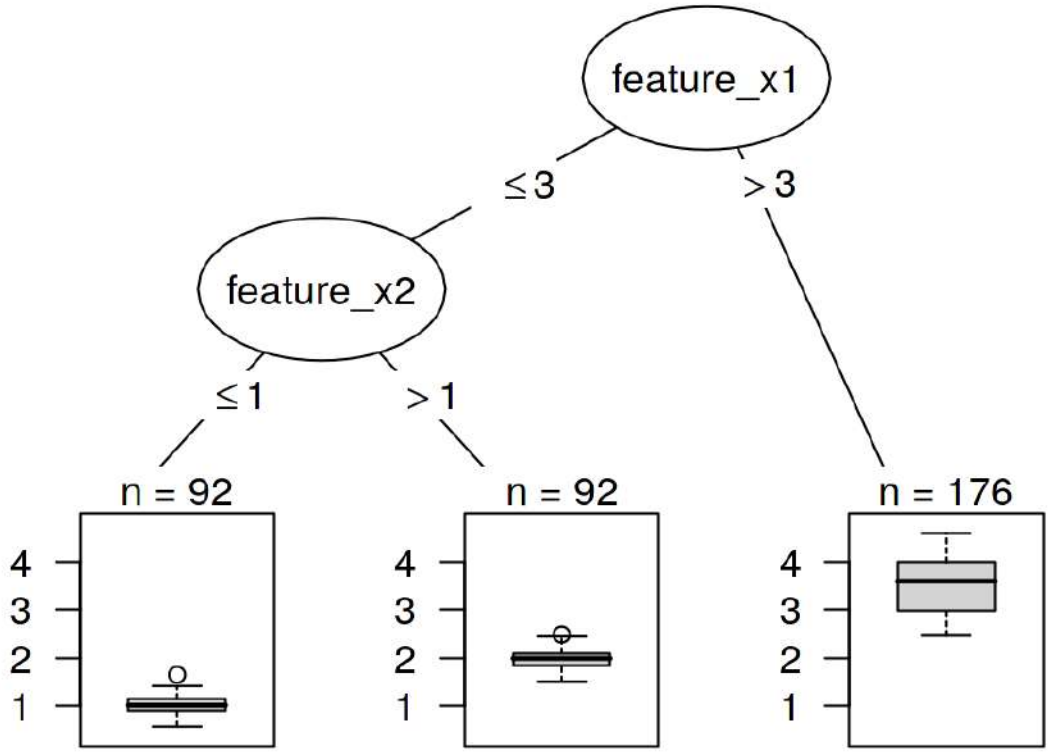
I am feeling a bit down lately.
Search for "**Amazon Alexa Gone Wild!!! Full version from beginning to end**".

Decision Tree

Linear regression and logistic regression models fail in situations where the relationship between features and outcome is nonlinear or where features interact with each other. Time to shine for the decision tree! Tree based models split the data multiple times according to certain cutoff values in the features. Through splitting, different subsets of the dataset are created, with each instance belonging to one subset. The final subsets are called terminal or leaf nodes and the intermediate subsets are called internal nodes or split nodes. To predict the outcome in each leaf node, the average outcome of the training data in this node is used. Trees can be used for classification and regression.

There are various algorithms that can grow a tree. They differ in the possible structure of the tree (e.g. number of splits per node), the criteria how to find the splits, when to stop splitting and how to estimate the simple models within the leaf nodes. The classification and regression trees (CART) algorithm is probably the most popular algorithm for tree induction. We will focus on CART, but the interpretation is similar for most other tree types. I recommend the book 'The Elements of Statistical Learning' (Friedman, Hastie and Tibshirani 2009)[41] for a more detailed introduction to CART.

[41] Friedman, Jerome, Trevor Hastie, and Robert Tibshirani. "The elements of statistical learning". www.web.stanford.edu/~hastie/ElemStatLearn/ (2009).



Decision tree with artificial data. Instances with a value greater than 3 for feature x1 end up in node 5. All other instances are assigned to node 3 or node 4, depending on whether values of feature x2 exceed 1.

The following formula describes the relationship between the outcome y and features x.

$$\hat{y} = \hat{f}(x) = \sum_{m=1}^{M} c_m I\{x \in R_m\}$$

Each instance falls into exactly one leaf node (=subset R_m). $I_{\{x \in R_m\}}$ is the identity function that returns 1 if x is in the subset R_m and 0 otherwise. If an instance falls into a leaf node R_l, the predicted outcome is $\hat{y} = c_l$, where c_l is the average of all training instances in leaf node R_l.

But where do the subsets come from? This is quite simple: CART takes a feature and determines which cut-off point minimizes the variance of y for a regression task or the Gini index of the class distribution of y for classification tasks. The variance tells us how much the y values in a node are spread around their mean value. The Gini index tells us how "impure" a node is, e.g. if all classes have the same frequency, the node is impure, if only one class is present, it is maximally pure. Variance and Gini index are minimized when the data points in the nodes have very similar values for y. As a consequence, the best cut-off point makes the two resulting subsets as different as possible with respect to the target outcome. For categorical features, the algorithm tries to create subsets

by trying different groupings of categories. After the best cutoff per feature has been determined, the algorithm selects the feature for splitting that would result in the best partition in terms of the variance or Gini index and adds this split to the tree. The algorithm continues this search-and-split recursively in both new nodes until a stop criterion is reached. Possible criteria are: A minimum number of instances that have to be in a node before the split, or the minimum number of instances that have to be in a terminal node.

Interpretation

The interpretation is simple: Starting from the root node, you go to the next nodes and the edges tell you which subsets you are looking at. Once you reach the leaf node, the node tells you the predicted outcome. All the edges are connected by 'AND'.

Template: If feature x is [smaller/bigger] than threshold c AND ... then the predicted outcome is the mean value of y of the instances in that node.

Feature importance

The overall importance of a feature in a decision tree can be computed in the following way: Go through all the splits for which the feature was used and measure how much it has reduced the variance or Gini index compared to the parent node. The sum of all importances is scaled to 100. This means that each importance can be interpreted as share of the overall model importance.

Tree decomposition

Individual predictions of a decision tree can be explained by decomposing the decision path into one component per feature. We can track a decision through the tree and explain a prediction by the contributions added at each decision node.

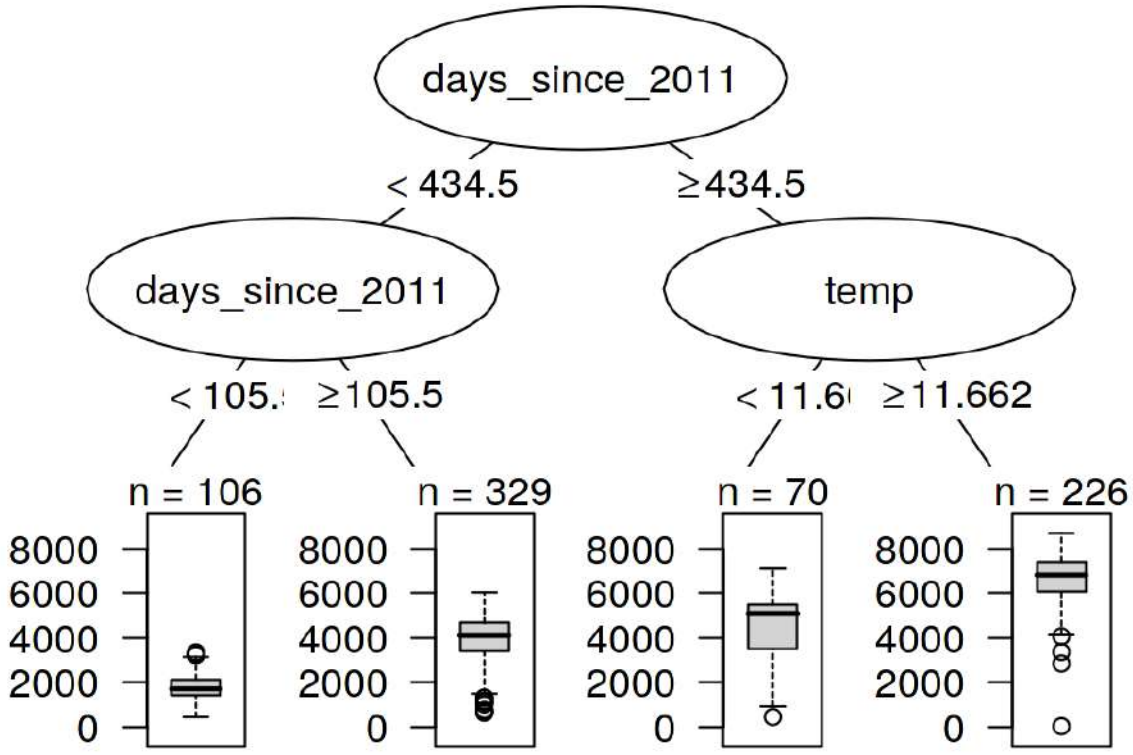
The root node in a decision tree is our starting point. If we were to use the root node to make predictions, it would predict the mean of the outcome of the training data. With the next split, we either subtract or add a term to this sum, depending on the next node in the path. To get to the final prediction, we have to follow the path of the data instance that we want to explain and keep adding to the formula.

$$\hat{f}(x) = \bar{y} + \sum_{d=1}^{D} \text{split.contrib(d,x)} = \bar{y} + \sum_{j=1}^{p} \text{feat.contrib(j,x)}$$

The prediction of an individual instance is the mean of the target outcome plus the sum of all contributions of the D splits that occur between the root node and the terminal node where the instance ends up. We are not interested in the split contributions though, but in the feature contributions. A feature might be used for more than one split or not at all. We can add the contributions for each of the p features and get an interpretation of how much each feature has contributed to a prediction.

Example

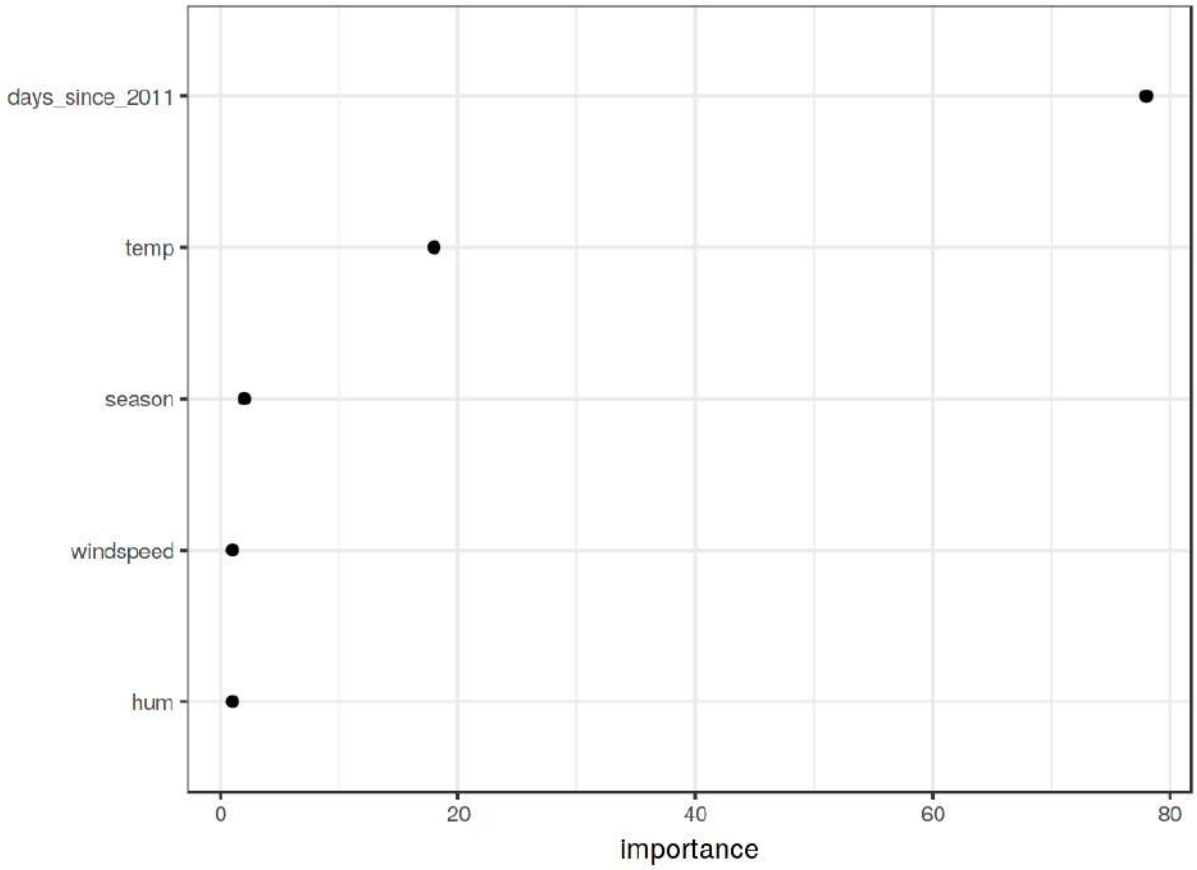
Let us have another look at the bike rental data. We want to predict the number of rented bikes on a certain day with a decision tree. The learned tree looks like this:



Regression tree fitted on the bike rental data. The maximum allowed depth for the tree was set to 2. The trend feature (days since 2011) and the temperature (temp) have been selected for the splits. The boxplots show the distribution of bicycle counts in the terminal node.

The feature importance tells us how much a feature helped to improve the purity of all nodes. Here, the variance was used, since predicting bicycle rentals is a regression task.

The visualized tree shows that both temperature and time trend were used for the splits, but does not quantify which feature was more important. The feature importance measure shows that the time trend is far more important than temperature.



Importance of the features measured by how much the node purity is improved on average.

Advantages

The tree structure is ideal for **capturing interactions** between features in the data.

The data ends up in **distinct groups** that are often easier to understand than points on a multi-dimensional hyperplane as in linear regression. The interpretation is arguably pretty simple.

The tree structure also has a **natural visualization**, with its nodes and edges.

Trees **create good explanations** as defined in the chapter on "Human-Friendly Explanations". The tree structure automatically invites to think about predicted values for individual instances as counterfactuals: "If a feature had been greater / smaller than the split point, the prediction would have been y1 instead of y2. The tree explanations are contrastive, since you can always compare the prediction of an instance with relevant "what if"-scenarios (as defined by the tree) that are simply the other leaf nodes of the tree. If the tree is short, like one to three splits deep, the resulting explanations are selective. A tree with a depth of three requires a maximum of three features and split points to create the explanation for the prediction of an individual instance. The truthfulness of the prediction depends on the predictive performance of the tree. The explanations for short trees

are very simple and general, because for each split the instance falls into either one or the other leaf, and binary decisions are easy to understand.

There is no need to transform features. In linear models, it is sometimes necessary to take the logarithm of a feature. A decision tree works equally well with any monotonic transformation of a feature.

Disadvantages

Trees fail to deal with linear relationships. Any linear relationship between an input feature and the outcome has to be approximated by splits, creating a step function. This is not efficient.

This goes hand in hand with **lack of smoothness**. Slight changes in the input feature can have a big impact on the predicted outcome, which is usually not desirable. Imagine a tree that predicts the value of a house and the tree uses the size of the house as one of the split feature. The split occurs at 100.5 square meters. Imagine user of a house price estimator using your decision tree model: They measure their house, come to the conclusion that the house has 99 square meters, enter it into the price calculator and get a prediction of 200 000 Euro. The users notice that they have forgotten to measure a small storage room with 2 square meters. The storage room has a sloping wall, so they are not sure whether they can count all of the area or only half of it. So they decide to try both 100.0 and 101.0 square meters. The results: The price calculator outputs 200 000 Euro and 205 000 Euro, which is rather unintuitive, because there has been no change from 99 square meters to 100.

Trees are also quite **unstable**. A few changes in the training dataset can create a completely different tree. This is because each split depends on the parent split. And if a different feature is selected as the first split feature, the entire tree structure changes. It does not create confidence in the model if the structure changes so easily.

Decision trees are very interpretable – as long as they are short. **The number of terminal nodes increases quickly with depth**. The more terminal nodes and the deeper the tree, the more difficult it becomes to understand the decision rules of a tree. A depth of 1 means 2 terminal nodes. Depth of 2 means max. 4 nodes. Depth of 3 means max. 8 nodes. The maximum number of terminal nodes in a tree is 2 to the power of the depth.

Software

For the examples in this chapter, I used the `rpart` R package that implements CART (classification and regression trees). CART is implemented in many programming languages, including Python[42]. Arguably, CART is a pretty old and somewhat outdated algorithm and there are some interesting new algorithms for fitting trees. You can find an overview of some R packages for decision trees in the Machine Learning and Statistical Learning CRAN Task View[43] under the keyword "Recursive Partitioning".

[42] https://scikit-learn.org/stable/modules/tree.html
[43] https://cran.r-project.org/web/views/MachineLearning.html

Decision Rules

A decision rule is a simple IF-THEN statement consisting of a condition (also called antecedent) and a prediction. For example: IF it rains today AND if it is April (condition), THEN it will rain tomorrow (prediction). A single decision rule or a combination of several rules can be used to make predictions.

Decision rules follow a general structure: IF the conditions are met THEN make a certain prediction. Decision rules are probably the most interpretable prediction models. Their IF-THEN structure semantically resembles natural language and the way we think, provided that the condition is built from intelligible features, the length of the condition is short (small number of `feature=value` pairs combined with an AND) and there are not too many rules. In programming, it is very natural to write IF-THEN rules. New in machine learning is that the decision rules are learned through an algorithm.

Imagine using an algorithm to learn decision rules for predicting the value of a house (`low`, `medium` or `high`). One decision rule learned by this model could be: If a house is bigger than 100 square meters and has a garden, then its value is high. More formally: IF `size>100 AND garden=1` THEN `value=high`.

Let us break down the decision rule:

- `size>100` is the first condition in the IF-part.
- `garden=1` is the second condition in the IF-part.
- The two conditions are connected with an 'AND' to create a new condition. Both must be true for the rule to apply.
- The predicted outcome (THEN-part) is `value=high`.

A decision rule uses at least one `feature=value` statement in the condition, with no upper limit on how many more can be added with an 'AND'. An exception is the default rule that has no explicit IF-part and that applies when no other rule applies, but more about this later.

The usefulness of a decision rule is usually summarized in two numbers: Support and accuracy.

Support or coverage of a rule: The percentage of instances to which the condition of a rule applies is called the support. Take for example the rule `size=big AND location=good THEN value=high` for predicting house values. Suppose 100 of 1000 houses are big and in a good location, then the support of the rule is 10%. The prediction (THEN-part) is not important for the calculation of support.

Accuracy or confidence of a rule: The accuracy of a rule is a measure of how accurate the rule is in predicting the correct class for the instances to which the condition of the rule applies. For example: Let us say of the 100 houses, where the rule `size=big AND location=good THEN value=high` applies, 85 have `value=high`, 14 have `value=medium` and 1 has `value=low`, then the accuracy of the rule is 85%.

Usually there is a trade-off between accuracy and support: By adding more features to the condition, we can achieve higher accuracy, but lose support.

To create a good classifier for predicting the value of a house you might need to learn not only one rule, but maybe 10 or 20. Then things can get more complicated and you can run into one of the following problems:

- Rules can overlap: What if I want to predict the value of a house and two or more rules apply and they give me contradictory predictions?
- No rule applies: What if I want to predict the value of a house and none of the rules apply?

There are two main strategies for combining multiple rules: Decision lists (ordered) and decision sets (unordered). Both strategies imply different solutions to the problem of overlapping rules.

A **decision list** introduces an order to the decision rules. If the condition of the first rule is true for an instance, we use the prediction of the first rule. If not, we go to the next rule and check if it applies and so on. Decision lists solve the problem of overlapping rules by only returning the prediction of the first rule in the list that applies.

A **decision set** resembles a democracy of the rules, except that some rules might have a higher voting power. In a set, the rules are either mutually exclusive, or there is a strategy for resolving conflicts, such as majority voting, which may be weighted by the individual rule accuracies or other quality measures. Interpretability suffers potentially when several rules apply.

Both decision lists and sets can suffer from the problem that no rule applies to an instance. This can be resolved by introducing a default rule. The default rule is the rule that applies when no other rule applies. The prediction of the default rule is often the most frequent class of the data points which are not covered by other rules. If a set or list of rules covers the entire feature space, we call it exhaustive. By adding a default rule, a set or list automatically becomes exhaustive.

There are many ways to learn rules from data and this book is far from covering them all. This chapter shows you three of them. The algorithms are chosen to cover a wide range of general ideas for learning rules, so all three of them represent very different approaches.

1. **OneR** learns rules from a single feature. OneR is characterized by its simplicity, interpretability and its use as a benchmark.
2. **Sequential covering** is a general procedure that iteratively learns rules and removes the data points that are covered by the new rule. This procedure is used by many rule learning algorithms.
3. **Bayesian Rule Lists** combine pre-mined frequent patterns into a decision list using Bayesian statistics. Using pre-mined patterns is a common approach used by many rule learning algorithms.

Let's start with the simplest approach: Using the single best feature to learn rules.

Learn Rules from a Single Feature (OneR)

The OneR algorithm suggested by Holte (1993)[44] is one of the simplest rule induction algorithms. From all the features, OneR selects the one that carries the most information about the outcome of interest and creates decision rules from this feature.

Despite the name OneR, which stands for "One Rule", the algorithm generates more than one rule: It is actually one rule per unique feature value of the selected best feature. A better name would be OneFeatureRules.

The algorithm is simple and fast:

1. Discretize the continuous features by choosing appropriate intervals.
2. For each feature:
    - Create a cross table between the feature values and the (categorical) outcome.
    - For each value of the feature, create a rule which predicts the most frequent class of the instances that have this particular feature value (can be read from the cross table).
    - Calculate the total error of the rules for the feature.
3. Select the feature with the smallest total error.

OneR always covers all instances of the dataset, since it uses all levels of the selected feature. Missing values can be either treated as an additional feature value or be imputed beforehand.

A OneR model is a decision tree with only one split. The split is not necessarily binary as in CART, but depends on the number of unique feature values.

Let us look at an example how the best feature is chosen by OneR. The following table shows an artificial dataset about houses with information about its value, location, size and whether pets are allowed. We are interested in learning a simple model to predict the value of a house.

location	size	pets	value
good	small	yes	high
good	big	no	high
good	big	no	high
bad	medium	no	medium
good	medium	only cats	medium
good	small	only cats	medium
bad	medium	yes	medium
bad	small	yes	low
bad	medium	yes	low
bad	small	no	low

OneR creates the cross tables between each feature and the outcome:

[44]Holte, Robert C. "Very simple classification rules perform well on most commonly used datasets." Machine learning 11.1 (1993): 63-90.

	value=low	value=medium	value=high
location=bad	3	2	0
location=good	0	2	3

	value=low	value=medium	value=high
size=big	0	0	2
size=medium	1	3	0
size=small	2	1	1

	value=low	value=medium	value=high
pets=no	1	1	2
pets=only cats	0	2	0
pets=yes	2	1	1

For each feature, we go through the table row by row: Each feature value is the IF-part of a rule; the most common class for instances with this feature value is the prediction, the THEN-part of the rule. For example, the size feature with the levels `small`, `medium` and `big` results in three rules. For each feature we calculate the total error rate of the generated rules, which is the sum of the errors. The location feature has the possible values `bad` and `good`. The most frequent value for houses in bad locations is `low` and when we use `low` as a prediction, we make two mistakes, because two houses have a `medium` value. The predicted value of houses in good locations is `high` and again we make two mistakes, because two houses have a `medium` value. The error we make by using the location feature is 4/10, for the size feature it is 3/10 and for the pet feature it is 4/10 . The size feature produces the rules with the lowest error and will be used for the final OneR model:

IF `size=small` THEN `value=small`
IF `size=medium` THEN `value=medium`
IF `size=big` THEN `value=high`

OneR prefers features with many possible levels, because those features can overfit the target more easily. Imagine a dataset that contains only noise and no signal, which means that all features take on random values and have no predictive value for the target. Some features have more levels than others. The features with more levels can now more easily overfit. A feature that has a separate level for each instance from the data would perfectly predict the entire training dataset. A solution would be to split the data into training and validation sets, learn the rules on the training data and evaluate the total error for choosing the feature on the validation set.

Ties are another issue, i.e. when two features result in the same total error. OneR solves ties by either taking the first feature with the lowest error or the one with the lowest p-value of a chi-squared test.

Example

Let us try OneR with real data. We use the cervical cancer classification task to test the OneR algorithm. All continuous input features were discretized into their 5 quantiles. The following rules are created:

Age	prediction
(12.9,27.2]	Healthy
(27.2,41.4]	Healthy
(41.4,55.6]	Healthy
(55.6,69.8]	Healthy
(69.8,84.1]	Healthy

The age feature was chosen by OneR as the best predictive feature. Since cancer is rare, for each rule the majority class and therefore the predicted label is always Healthy, which is rather unhelpful. It does not make sense to use the label prediction in this unbalanced case. The cross table between the 'Age' intervals and Cancer/Healthy together with the percentage of women with cancer is more informative:

	# Cancer	# Healthy	P(Cancer)
Age=(12.9,27.2]	26	477	0.05
Age=(27.2,41.4]	25	290	0.08
Age=(41.4,55.6]	4	31	0.11
Age=(55.6,69.8]	0	1	0.00
Age=(69.8,84.1]	0	4	0.00

But before you start interpreting anything: Since the prediction for every feature and every value is Healthy, the total error rate is the same for all features. The ties in the total error are, by default, resolved by using the first feature from the ones with the lowest error rates (here, all features have 55/858), which happens to be the Age feature.

OneR does not support regression tasks. But we can turn a regression task into a classification task by cutting the continuous outcome into intervals. We use this trick to predict the number of rented bikes with OneR by cutting the number of bikes into its four quartiles (0-25%, 25-50%, 50-75% and 75-100%). The following table shows the selected feature after fitting the OneR model:

mnth	prediction
JAN	[22,3152]
FEB	[22,3152]
MAR	[22,3152]
APR	(3152,4548]
MAY	(5956,8714]
JUN	(4548,5956]
JUL	(5956,8714]
AUG	(5956,8714]
SEP	(5956,8714]
OKT	(5956,8714]
NOV	(3152,4548]
DEZ	[22,3152]

The selected feature is the month. The month feature has (surprise!) 12 feature levels, which is more than most other features have. So there is a danger of overfitting. On the more optimistic side: the month feature can handle the seasonal trend (e.g. less rented bikes in winter) and the predictions

seem sensible.

Now we move from the simple OneR algorithm to a more complex procedure using rules with more complex conditions consisting of several features: Sequential Covering.

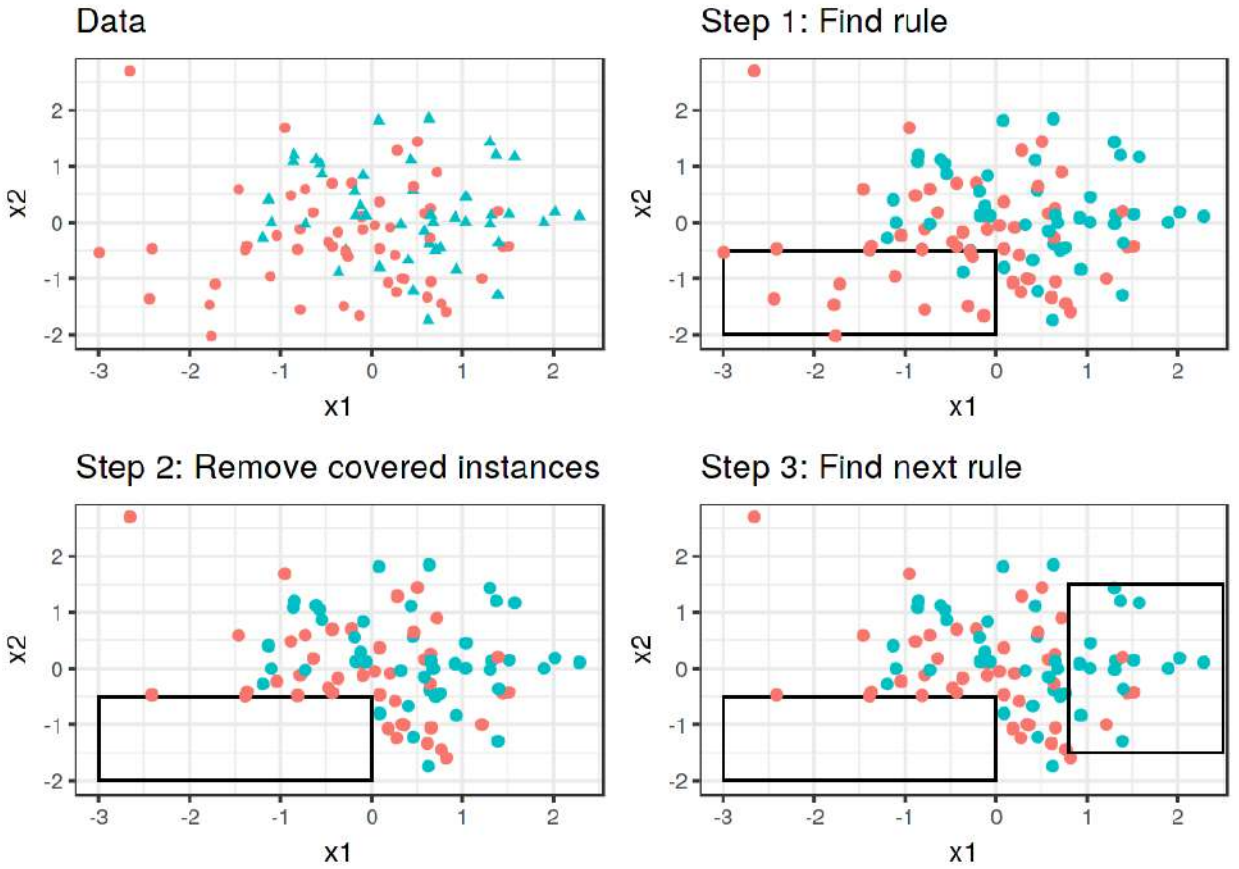
Sequential Covering

Sequential covering is a general procedure that repeatedly learns a single rule to create a decision list (or set) that covers the entire dataset rule by rule. Many rule-learning algorithms are variants of the sequential covering algorithm. This chapter introduces the main recipe and uses RIPPER, a variant of the sequential covering algorithm for the examples.

The idea is simple: First, find a good rule that applies to some of the data points. Remove all data points which are covered by the rule. A data point is covered when the conditions apply, regardless of whether the points are classified correctly or not. Repeat the rule-learning and removal of covered points with the remaining points until no more points are left or another stop condition is met. The result is a decision list. This approach of repeated rule-learning and removal of covered data points is called "separate-and-conquer".

Suppose we already have an algorithm that can create a single rule that covers part of the data. The sequential covering algorithm for two classes (one positive, one negative) works like this:

- Start with an empty list of rules (rlist).
- Learn a rule r.
- While the list of rules is below a certain quality threshold (or positive examples are not yet covered):
  - Add rule r to rlist.
  - Remove all data points covered by rule r.
  - Learn another rule on the remaining data.
- Return the decision list.



The covering algorithm works by sequentially covering the feature space with single rules and removing the data points that are already covered by those rules. For visualization purposes, the features x1 and x2 are continuous, but most rule learning algorithms require categorical features.

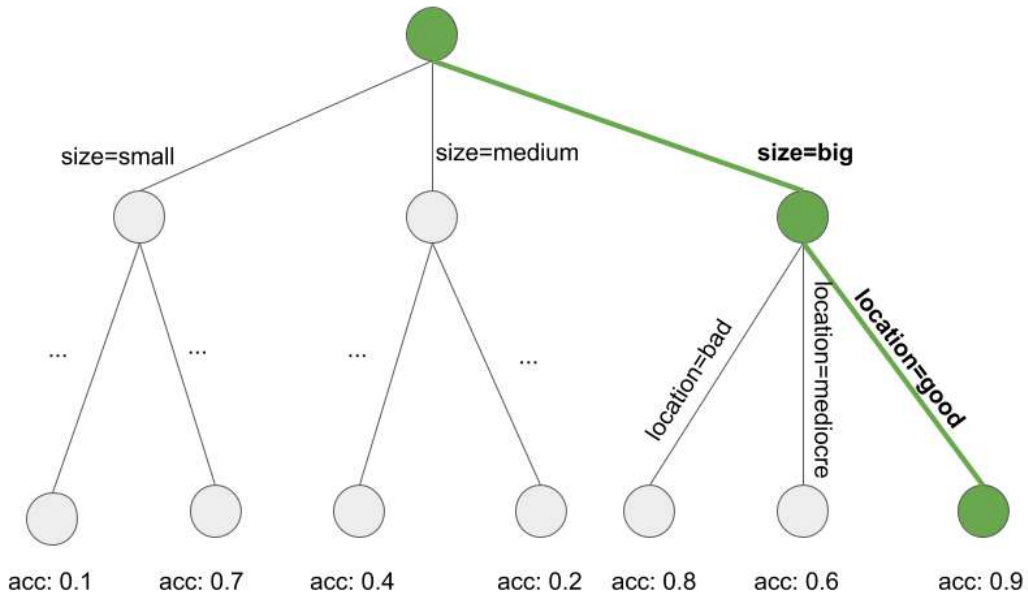
For example: We have a task and dataset for predicting the values of houses from size, location and whether pets are allowed. We learn the first rule, which turns out to be: If `size=big` and `location=good`, then `value=high`. Then we remove all big houses in good locations from the dataset. With the remaining data we learn the next rule. Maybe: If `location=good`, then `value=medium`. Note that this rule is learned on data without big houses in good locations, leaving only medium and small houses in good locations.

For multi-class settings, the approach must be modified. First, the classes are ordered by increasing prevalence. The sequential covering algorithm starts with the least common class, learns a rule for it, removes all covered instances, then moves on to the second least common class and so on. The current class is always treated as the positive class and all classes with a higher prevalence are combined in the negative class. The last class is the default rule. This is also referred to as one-versus-all strategy in classification.

How do we learn a single rule? The OneR algorithm would be useless here, since it would always cover the whole feature space. But there are many other possibilities. One possibility is to learn a single rule from a decision tree with beam search:

- Learn a decision tree (with CART or another tree learning algorithm).
- Start at the root node and recursively select the purest node (e.g. with the lowest misclassification rate).
- The majority class of the terminal node is used as the rule prediction; the path leading to that node is used as the rule condition.

The following figure illustrates the beam search in a tree:



Learning a rule by searching a path through a decision tree. A decision tree is grown to predict the target of interest. We start at the root node, greedily and iteratively follow the path which locally produces the purest subset (e.g. highest accuracy) and add all the split values to the rule condition. We end up with: If `location=good` and `size=big`, then `value=high`.

Learning a single rule is a search problem, where the search space is the space of all possible rules. The goal of the search is to find the best rule according to some criteria. There are many different search strategies: hill-climbing, beam search, exhaustive search, best-first search, ordered search, stochastic search, top-down search, bottom-up search, ...

RIPPER (Repeated Incremental Pruning to Produce Error Reduction) by Cohen (1995)[45] is a variant of the Sequential Covering algorithm. RIPPER is a bit more sophisticated and uses a post-processing phase (rule pruning) to optimize the decision list (or set). RIPPER can run in ordered or unordered mode and generate either a decision list or decision set.

Examples

We will use RIPPER for the examples.

[45]Cohen, William W. "Fast effective rule induction." Machine Learning Proceedings (1995). 115-123.

The RIPPER algorithm does not find any rule in the classification task for cervical cancer.

When we use RIPPER on the regression task to predict bike counts some rules are found. Since RIPPER only works for classification, the bike counts must be turned into a categorical outcome. I achieved this by cutting the bike counts into the quartiles. For example (4548, 5956) is the interval covering predicted bike counts between 4548 and 5956. The following table shows the decision list of learned rules.

rules
(days_since_2011 >= 438) and (temp >= 17) and (temp <= 27) and (hum <= 67) ⇒ cnt=(5956,8714]
(days_since_2011 >= 443) and (temp >= 12) and (weathersit = GOOD) and (hum >= 59) ⇒ cnt=(5956,8714]
(days_since_2011 >= 441) and (windspeed <= 10) and (temp >= 13) ⇒ cnt=(5956,8714]
(temp >= 12) and (hum <= 68) and (days_since_2011 >= 551) ⇒ cnt=(5956,8714]
(days_since_2011 >= 100) and (days_since_2011 <= 434) and (hum <= 72) and (workingday = WORKING DAY) ⇒ cnt=(3152,4548]
(days_since_2011 >= 106) and (days_since_2011 <= 323) ⇒ cnt=(3152,4548]
⇒ cnt=[22,3152]

The interpretation is simple: If the conditions apply, we predict the interval on the right hand side for the number of bikes. The last rule is the default rule that applies when none of the other rules apply to an instance. To predict a new instance, start at the top of the list and check whether a rule applies. When a condition matches, then the right hand side of the rule is the prediction for this instance. The default rule ensures that there is always a prediction.

Bayesian Rule Lists

In this section, I will show you another approach to learning a decision list, which follows this rough recipe:

1. Pre-mine frequent patterns from the data that can be used as conditions for the decision rules.
2. Learn a decision list from a selection of the pre-mined rules.

A specific approach using this recipe is called Bayesian Rule Lists (Letham et. al, 2015)[46] or BRL for short. BRL uses Bayesian statistics to learn decision lists from frequent patterns which are pre-mined with the FP-tree algorithm (Borgelt 2005)[47]

But let us start slowly with the first step of BRL.

Pre-mining of frequent patterns

A frequent pattern is the frequent (co-)occurrence of feature values. As a pre-processing step for the BRL algorithm, we use the features (we do not need the target outcome in this step) and extract

[46]Letham, Benjamin, et al. "Interpretable classifiers using rules and Bayesian analysis: Building a better stroke prediction model." The Annals of Applied Statistics 9.3 (2015): 1350-1371.

[47]Borgelt, C. "An implementation of the FP-growth algorithm." Proceedings of the 1st International Workshop on Open Source Data Mining Frequent Pattern Mining Implementations - OSDM '05, 1–5. http://doi.org/10.1145/1133905.1133907 (2005).

frequently occurring patterns from them. A pattern can be a single feature value such as `size=medium` or a combination of feature values such as `size=medium AND location=bad`.

The frequency of a pattern is measured with its support in the dataset:

$$Support(x_j = A) = \frac{1}{n}\sum_{i=1}^{n} I(x_j^{(i)} = A)$$

where A is the feature value, n the number of data points in the dataset and I the indicator function that returns 1 if the feature x_j of the instance i has level A otherwise 0. In a dataset of house values, if 20% of houses have no balcony and 80% have one or more, then the support for the pattern `balcony=0` is 20%. Support can also be measured for combinations of feature values, for example for `balcony=0 AND pets=allowed`.

There are many algorithms to find such frequent patterns, for example Apriori or FP-Growth. Which you use does not matter much, only the speed at which the patterns are found is different, but the resulting patterns are always the same.

I will give you a rough idea of how the Apriori algorithm works to find frequent patterns. Actually the Apriori algorithm consists of two parts, where the first part finds frequent patterns and the second part builds association rules from them. For the BRL algorithm, we are only interested in the frequent patterns that are generated in the first part of Apriori.

In the first step, the Apriori algorithm starts with all feature values that have a support greater than the minimum support defined by the user. If the user says that the minimum support should be 10% and only 5% of the houses have `size=big`, we would remove that feature value and keep only `size=medium` and `size=small` as patterns. This does not mean that the houses are removed from the data, it just means that `size=big` is not returned as frequent pattern. Based on frequent patterns with a single feature value, the Apriori algorithm iteratively tries to find combinations of feature values of increasingly higher order. Patterns are constructed by combining `feature=value` statements with a logical AND, e.g. `size=medium AND location=bad`. Generated patterns with a support below the minimum support are removed. In the end we have all the frequent patterns. Any subset of a frequent pattern is frequent again, which is called the Apriori property. It makes sense intuitively: By removing a condition from a pattern, the reduced pattern can only cover more or the same number of data points, but not less. For example, if 20% of the houses are `size=medium and location=good`, then the support of houses that are only `size=medium` is 20% or greater. The Apriori property is used to reduce the number of patterns to be inspected. Only in the case of frequent patterns we have to check patterns of higher order.

Now we are done with pre-mining conditions for the Bayesian Rule List algorithm. But before we move on to the second step of BRL, I would like to hint at another way for rule-learning based on pre-mined patterns. Other approaches suggest including the outcome of interest into the frequent pattern mining process and also executing the second part of the Apriori algorithm that builds IF-THEN rules. Since the algorithm is unsupervised, the THEN-part also contains feature values we are not interested in. But we can filter by rules that have only the outcome of interest in the THEN-part.

These rules already form a decision set, but it would also be possible to arrange, prune, delete or recombine the rules.

In the BRL approach however, we work with the frequent patterns and learn the THEN-part and how to arrange the patterns into a decision list using Bayesian statistics.

Learning Bayesian Rule Lists

The goal of the BRL algorithm is to learn an accurate decision list using a selection of the pre-mined conditions, while prioritizing lists with few rules and short conditions. BRL addresses this goal by defining a distribution of decision lists with prior distributions for the length of conditions (preferably shorter rules) and the number of rules (preferably a shorter list).

The posteriori probability distribution of lists makes it possible to say how likely a decision list is, given assumptions of shortness and how well the list fits the data. Our goal is to find the list that maximizes this posterior probability. Since it is not possible to find the exact best list directly from the distributions of lists, BRL suggests the following recipe:
1) Generate an initial decision list, which is randomly drawn from the priori distribution.
2) Iteratively modify the list by adding, switching or removing rules, ensuring that the resulting lists follow the posterior distribution of lists.
3) Select the decision list from the sampled lists with the highest probability according to the posteriori distribution.

Let us go over the algorithm more closely: The algorithm starts with pre-mining feature value patterns with the FP-Growth algorithm. BRL makes a number of assumptions about the distribution of the target and the distribution of the parameters that define the distribution of the target. (That's Bayesian statistic.) If you are unfamiliar with Bayesian statistics, do not get too caught up in the following explanations. It is important to know that the Bayesian approach is a way to combine existing knowledge or requirements (so-called priori distributions) while also fitting to the data. In the case of decision lists, the Bayesian approach makes sense, since the prior assumptions nudges the decision lists to be short with short rules.

The goal is to sample decision lists d from the posteriori distribution:

$$\underbrace{p(d|x,y,A,\alpha,\lambda,\eta)}_{posteriori} \propto \underbrace{p(y|x,d,\alpha)}_{likelihood} \cdot \underbrace{p(d|A,\lambda,\eta)}_{priori}$$

where d is a decision list, x are the features, y is the target, A the set of pre-mined conditions, λ the prior expected length of the decision lists, η the prior expected number of conditions in a rule, α the prior pseudo-count for the positive and negative classes which is best fixed at (1,1).

$$p(d|x,y,A,\alpha,\lambda,\eta)$$

quantifies how probable a decision list is, given the observed data and the priori assumptions. This is proportional to the likelihood of the outcome y given the decision list and the data times the probability of the list given prior assumptions and the pre-mined conditions.

$$p(y|x, d, \alpha)$$

is the likelihood of the observed y, given the decision list and the data. BRL assumes that y is generated by a Dirichlet-Multinomial distribution. The better the decision list d explains the data, the higher the likelihood.

$$p(d|A, \lambda, \eta)$$

is the prior distribution of the decision lists. It multiplicatively combines a truncated Poisson distribution (parameter λ) for the number of rules in the list and a truncated Poisson distribution (parameter η) for the number of feature values in the conditions of the rules.

A decision list has a high posterior probability if it explains the outcome y well and is also likely according to the prior assumptions.

Estimations in Bayesian statistics are always a bit tricky, because we usually cannot directly calculate the correct answer, but we have to draw candidates, evaluate them and update our posteriori estimates using the Markov chain Monte Carlo method. For decision lists, this is even more tricky, because we have to draw from the distribution of decision lists. The BRL authors propose to first draw an initial decision list and then iteratively modify it to generate samples of decision lists from the posterior distribution of the lists (a Markov chain of decision lists). The results are potentially dependent on the initial decision list, so it is advisable to repeat this procedure to ensure a great variety of lists. The default in the software implementation is 10 times. The following recipe tells us how to draw an initial decision list:

- Pre-mine patterns with FP-Growth.
- Sample the list length parameter m from a truncated Poisson distribution.
- For the default rule: Sample the Dirichlet-Multinomial distribution parameter θ_0 of the target value (i.e. the rule that applies when nothing else applies).
- For decision list rule j=1,...,m, do:
  - Sample the rule length parameter l (number of conditions) for rule j.
  - Sample a condition of length l_j from the pre-mined conditions.
  - Sample the Dirichlet-Multinomial distribution parameter for the THEN-part (i.e. for the distribution of the target outcome given the rule)
- For each observation in the dataset:
  - Find the rule from the decision list that applies first (top to bottom).
  - Draw the predicted outcome from the probability distribution (Binomial) suggested by the rule that applies.

The next step is to generate many new lists starting from this initial sample to obtain many samples from the posterior distribution of decision lists.

The new decision lists are sampled by starting from the initial list and then randomly either moving a rule to a different position in the list or adding a rule to the current decision list from the pre-mined conditions or removing a rule from the decision list. Which of the rules is switched, added or deleted is chosen at random. At each step, the algorithm evaluates the posteriori probability of the decision list (mixture of accuracy and shortness). The Metropolis Hastings algorithm ensures that we sample decision lists that have a high posterior probability. This procedure provides us with many samples from the distribution of decision lists. The BRL algorithm selects the decision list of the samples with the highest posterior probability.

Examples

That is it with the theory, now let's see the BRL method in action. The examples use a faster variant of BRL called Scalable Bayesian Rule Lists (SBRL) by Yang et. al (2017) [48]. We use the SBRL algorithm to predict the risk for cervical cancer. I first had to discretize all input features for the SBRL algorithm to work. For this purpose I binned the continuous features based on the frequency of the values by quantiles.

We get the following rules:

rules
If {STDs=1} (rule[259]) then positive probability = 0.16049383
else if {Hormonal.Contraceptives..years.=[0,10)} (rule[82]) then positive probability = 0.04685408
else (default rule) then positive probability = 0.27777778

Note that we get sensible rules, since the prediction on the THEN-part is not the class outcome, but the predicted probability for cancer.

The conditions were selected from patterns that were pre-mined with the FP-Growth algorithm. The following table displays the pool of conditions the SBRL algorithm could choose from for building a decision list. The maximum number of feature values in a condition I allowed as a user was two. Here is a sample of ten patterns:

pre-mined conditions
First.sexual.intercourse=[17.3,24.7),STDs=1
Hormonal.Contraceptives=0,STDs=0
Num.of.pregnancies=[0,3.67),STDs..Number.of.diagnosis=[0,1)
Smokes=1
First.sexual.intercourse=[10,17.3)
Smokes=1,STDs..Number.of.diagnosis=[0,1)
STDs..number.=[1.33,2.67)
Num.of.pregnancies=[3.67,7.33)
Num.of.pregnancies=[3.67,7.33),IUD..years.=[0,6.33)
Age=[13,36.7),STDs..Number.of.diagnosis=[1,2)

Next, we apply the SBRL algorithm to the bike rental prediction task. This only works if the

[48] Yang, Hongyu, Cynthia Rudin, and Margo Seltzer. "Scalable Bayesian rule lists." Proceedings of the 34th International Conference on Machine Learning-Volume 70. JMLR. org, 2017.

regression problem of predicting bike counts is converted into a binary classification task. I have arbitrarily created a classification task by creating a label that is 1 if the number of bikes exceeds 4000 bikes on a day, else 0.

The following list was learned by SBRL:

rules
If {yr=2011,temp=[-5.22,7.35)} (rule[718]) then positive probability = 0.01041667
else if {yr=2012,temp=[7.35,19.9)} (rule[823]) then positive probability = 0.88125000
else if {yr=2012,temp=[19.9,32.5]} (rule[816]) then positive probability = 0.99253731
else if {season=SPRING} (rule[351]) then positive probability = 0.06410256
else if {yr=2011,temp=[7.35,19.9)} (rule[730]) then positive probability = 0.44444444
else (default rule) then positive probability = 0.79746835

Let us predict the probability that the number of bikes will exceed 4000 for a day in 2012 with a temperature of 17 degrees Celsius. The first rule does not apply, since it only applies for days in 2011. The second rule applies, because the day is in 2012 and 17 degrees lies in the interval `[7.35,19.9)`. Our prediction for the probability is that more than 4000 bikes are rented is 88%.

Advantages

This section discusses the benefits of IF-THEN rules in general.

IF-THEN rules are **easy to interpret**. They are probably the most interpretable of the interpretable models. This statement only applies if the number of rules is small, the conditions of the rules are short (maximum 3 I would say) and if the rules are organized in a decision list or a non-overlapping decision set.

Decision rules can be **as expressive as decision trees**, **while being more compact**. Decision trees often also suffer from replicated sub-trees, that is, when the splits in a left and a right child node have the same structure.

The **prediction with IF-THEN rules is fast**, since only a few binary statements need to be checked to determine which rules apply.

Decision rules are **robust** against monotonous transformations of the input features, because only the threshold in the conditions changes. They are also robust against outliers, since it only matters if a condition applies or not.

IF-THEN rules usually generate sparse models, which means that not many features are included. They **select only the relevant features** for the model. For example, a linear model assigns a weight to every input feature by default. Features that are irrelevant can simply be ignored by IF-THEN rules.

Simple rules like from OneR **can be used as baseline** for more complex algorithms.

Disadvantages

This section deals with the disadvantages of IF-THEN rules in general.

The research and literature for IF-THEN rules focuses on classification and almost **completely neglects regression**. While you can always divide a continuous target into intervals and turn it into a classification problem, you always lose information. In general, approaches are more attractive if they can be used for both regression and classification.

Often the **features also have to be categorical**. That means numeric features must be categorized if you want to use them. There are many ways to cut a continuous feature into intervals, but this is not trivial and comes with many questions without clear answers. How many intervals should the feature be divided into? What is the splitting criteria: Fixed interval lengths, quantiles or something else? Categorizing continuous features is a non-trivial issue that is often neglected and people just use the next best method (like I did in the examples).

Many of the older rule-learning algorithms are prone to overfitting. The algorithms presented here all have at least some safeguards to prevent overfitting: OneR is limited because it can only use one feature (only problematic if the feature has too many levels or if there are many features, which equates to the multiple testing problem), RIPPER does pruning and Bayesian Rule Lists impose a prior distribution on the decision lists.

Decision rules are **bad in describing linear relationships** between features and output. That is a problem they share with the decision trees. Decision trees and rules can only produce step-like prediction functions, where changes in the prediction are always discrete steps and never smooth curves. This is related to the issue that the inputs have to be categorical. In decision trees, they are implicitly categorized by splitting them.

Software and Alternatives

OneR is implemented in the R package OneR[49], which was used for the examples in this book. OneR is also implemented in the Weka machine learning library[50] and as such available in Java, R and Python. RIPPER is also implemented in Weka. For the examples, I used the R implementation of JRIP in the RWeka package[51]. SBRL is available as R package[52] (which I used for the examples), in Python[53] or as C implementation[54].

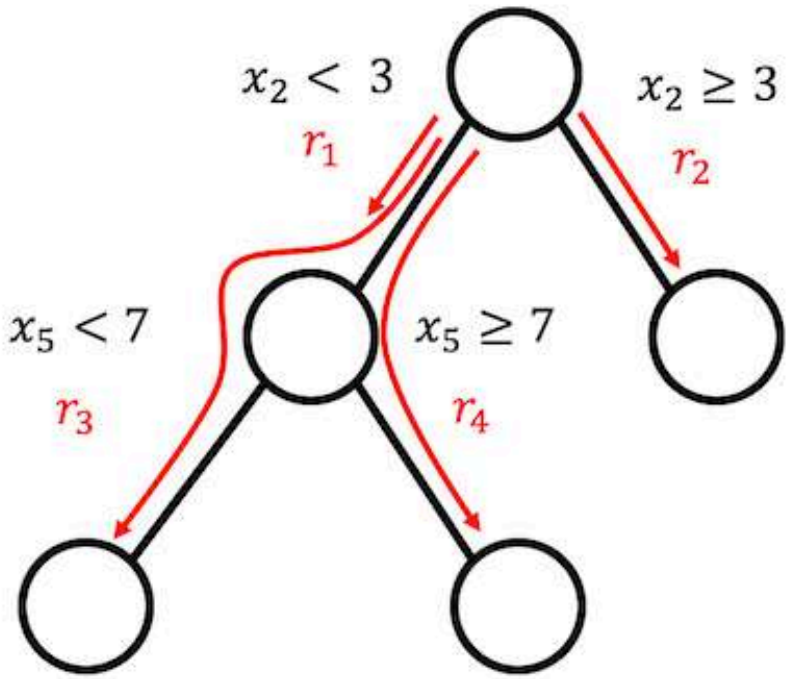
I will not even try to list all alternatives for learning decision rule sets and lists, but will point to some summarizing work. I recommend the book "Foundations of Rule Learning" by Fuernkranz et. al (2012)[55]. It is an extensive work on learning rules, for those who want to delve deeper into the topic. It provides a holistic framework for thinking about learning rules and presents many rule learning algorithms. I also recommend to checkout the Weka rule learners[56], which implement RIPPER, M5Rules, OneR, PART and many more. IF-THEN rules can be used in linear models as described in this book in the chapter about the RuleFit algorithm.

[49] https://cran.r-project.org/web/packages/OneR/
[50] (https://www.eecs.yorku.ca/tdb/_doc.php/userg/sw/weka/doc/weka/classifiers/rules/package-summary.html)
[51] https://cran.r-project.org/web/packages/RWeka/index.html
[52] https://cran.r-project.org/web/packages/sbrl/index.html
[53] https://github.com/datascienceinc/Skater
[54] https://github.com/Hongyuy/sbrlmod
[55] Fürnkranz, Johannes, Dragan Gamberger, and Nada Lavrač. "Foundations of rule learning." Springer Science & Business Media, (2012).
[56] http://weka.sourceforge.net/doc.dev/weka/classifiers/rules/package-summary.html

RuleFit

The RuleFit algorithm by Friedman and Popescu (2008)[57] learns sparse linear models that include automatically detected interaction effects in the form of decision rules.

The linear regression model does not account for interactions between features. Would it not be convenient to have a model that is as simple and interpretable as linear models, but also integrates feature interactions? RuleFit fills this gap. RuleFit learns a sparse linear model with the original features and also a number of new features that are decision rules. These new features capture interactions between the original features. RuleFit automatically generates these features from decision trees. Each path through a tree can be transformed into a decision rule by combining the split decisions into a rule. The node predictions are discarded and only the splits are used in the decision rules:



4 rules can be generated from a tree with 3 terminal nodes.

Where do those decision trees come from? The trees are trained to predict the outcome of interest. This ensures that the splits are meaningful for the prediction task. Any algorithm that generates a lot of trees can be used for RuleFit, for example a random forest. Each tree is decomposed into decision rules that are used as additional features in a sparse linear regression model (Lasso).

The RuleFit paper uses the Boston housing data to illustrate this: The goal is to predict the median house value of a Boston neighborhood. One of the rules generated by RuleFit is: IF `number of rooms`

[57]Friedman, Jerome H, and Bogdan E Popescu. "Predictive learning via rule ensembles." The Annals of Applied Statistics. JSTOR, 916â€“54. (2008).

> 6.64 AND concentration of nitric oxide <0.67 THEN 1 ELSE 0.

RuleFit also comes with a feature importance measure that helps to identify linear terms and rules that are important for the predictions. Feature importance is calculated from the weights of the regression model. The importance measure can be aggregated for the original features (which are used in their "raw" form and possibly in many decision rules).

RuleFit also introduces partial dependence plots to show the average change in prediction by changing a feature. The partial dependence plot is a model-agnostic method that can be used with any model, and is explained in the book chapter on partial dependence plots.

Interpretation and Example

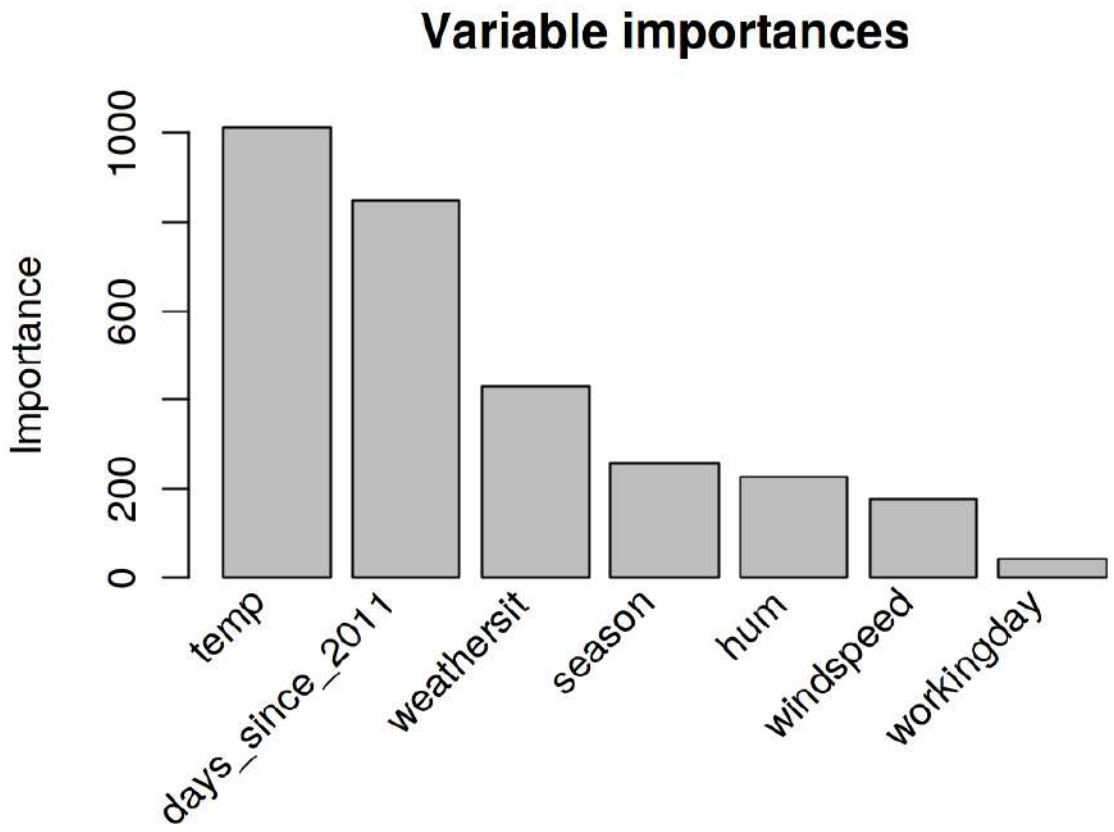
Since RuleFit estimates a linear model in the end, the interpretation is the same as for "normal" linear models. The only difference is that the model has new features derived from decision rules. Decision rules are binary features: A value of 1 means that all conditions of the rule are met, otherwise the value is 0. For linear terms in RuleFit, the interpretation is the same as in linear regression models: If the feature increases by one unit, the predicted outcome changes by the corresponding feature weight.

In this example, we use RuleFit to predict the number of rented bicycles on a given day. The table shows five of the rules that were generated by RuleFit, along with their Lasso weights and importances. The calculation is explained later in the chapter.

Description	Weight	Importance
days_since_2011 > 111 & weathersit in ("GOOD", "MISTY")	664	253
37.25 <= hum <= 90	-17	227
days_since_2011 > 428 & temp > 5	460	225
temp > 13 & days_since_2011 > 554	550	194
temp > 8 & weathersit in ("GOOD", "MISTY")	409	188

The most important rule was: "days_since_2011 > 111 & weathersit in ("GOOD", "MISTY")" and the corresponding weight is 664. The interpretation is: If days_since_2011 > 111 & weathersit in ("GOOD", "MISTY"), then the predicted number of bikes increases by 664, when all other feature values remain fixed. In total, 278 such rules were created from the original 8 features. Quite a lot! But thanks to Lasso, only 39 of the 278 have a weight different from 0.

Computing the global feature importances reveals that temperature and time trend are the most important features:



Feature importance measures for a RuleFit model predicting bike counts. The most important features for the predictions were temperature and time trend.

The feature importance measurement includes the importance of the raw feature term and all the decision rules in which the feature appears.

Interpretation template

The interpretation is analogous to linear models: The predicted outcome changes by β_j if feature x_j changes by one unit, provided all other features remain unchanged. The weight interpretation of a decision rule is a special case: If all conditions of a decision rule r_k apply, the predicted outcome changes by α_k (the learned weight of rule r_k in the linear model).

For classification (using logistic regression instead of linear regression): If all conditions of the decision rule r_k apply, the odds for event vs. no-event changes by a factor of α_k.

Theory

Let us dive deeper into the technical details of the RuleFit algorithm. RuleFit consists of two components: The first component creates "rules" from decision trees and the second component fits a linear model with the original features and the new rules as input (hence the name "RuleFit").

Step 1: Rule generation

What does a rule look like? The rules generated by the algorithm have a simple form. For example: IF `x2 < 3` AND `x5 < 7` THEN 1 ELSE 0. The rules are constructed by decomposing decision trees: Any path to a node in a tree can be converted to a decision rule. The trees used for the rules are fitted to predict the target outcome. Therefore the splits and resulting rules are optimized to predict the outcome you are interested in. You simply chain the binary decisions that lead to a certain node with "AND", and voilÃ , you have a rule. It is desirable to generate a lot of diverse and meaningful rules. Gradient boosting is used to fit an ensemble of decision trees by regressing or classifying y with your original features X. Each resulting tree is converted into multiple rules. Not only boosted trees, but any tree ensemble algorithm can be used to generate the trees for RuleFit. A tree ensemble can be described with this general formula:

$$f(x) = a_0 + \sum_{m=1}^{M} a_m f_m(X)$$

M is the number of trees and $f_m(x)$ is the prediction function of the m-th tree. The α's are the weights. Bagged ensembles, random forest, AdaBoost and MART produce tree ensembles and can be used for RuleFit.

We create the rules from all trees of the ensemble. Each rule r_m takes the form of:

$$r_m(x) = \prod_{j \in \mathrm{T}_m} I(x_j \in s_{jm})$$

where T_m is the set of features used in the m-th tree, I is the indicator function that is 1 when feature x_j is in the specified subset of values s for the j-th feature (as specified by the tree splits) and 0 otherwise. For numerical features, s_{jm} is an interval in the value range of the feature. The interval looks like one of the two cases:

$$x_{s_{jm},\text{lower}} < x_j$$

$$x_j < x_{s_{jm},upper}$$

Further splits in that feature possibly lead to more complicated intervals. For categorical features the subset s contains some specific categories of the feature.

A made up example for the bike rental dataset:

$$r_{17}(x) = I(x_{\text{temp}} < 15) \cdot I(x_{\text{weather}} \in \{\text{good}, \text{cloudy}\}) \cdot I(10 \leq x_{\text{windspeed}} < 20)$$

This rule returns 1 if all three conditions are met, otherwise 0. RuleFit extracts all possible rules from a tree, not only from the leaf nodes. So another rule that would be created is:

$$r_{18}(x) = I(x_{\text{temp}} < 15) \cdot I(x_{\text{weather}} \in \{\text{good}, \text{cloudy}\}$$

Altogether, the number of rules created from an ensemble of M trees with t_m terminal nodes each is:

$$K = \sum_{m=1}^{M} 2(t_m - 1)$$

A trick introduced by the RuleFit authors is to learn trees with random depth so that many diverse rules with different lengths are generated. Note that we discard the predicted value in each node and only keep the conditions that lead us to a node and then we create a rule from it. The weighting of the decision rules is done in step 2 of RuleFit.

Another way to see step 1: RuleFit generates a new set of features from your original features. These features are binary and can represent quite complex interactions of your original features. The rules are chosen to maximize the prediction task. The rules are automatically generated from the covariates matrix X. You can simply see the rules as new features based on your original features.

Step 2: Sparse linear model

You get MANY rules in step 1. Since the first step can be seen as only a feature transformation, you are still not done with fitting a model. Also, you want to reduce the number of rules. In addition to the rules, all your "raw" features from your original dataset will also be used in the sparse linear model. Every rule and every original feature becomes a feature in the linear model and gets a weight estimate. The original raw features are added because trees fail at representing simple linear relationships between y and x. Before we train a sparse linear model, we winsorize the original features so that they are more robust against outliers:

$$l_j^*(x_j) = min(\delta_j^+, max(\delta_j^-, x_j))$$

where δ_j^- and δ_j^+ are the δ quantiles of the data distribution of feature x_j. A choice of 0.05 for δ means that any value of feature x_j that is in the 5% lowest or 5% highest values will be set to the quantiles at 5% or 95% respectively. As a rule of thumb, you can choose δ = 0.025. In addition, the linear terms have to be normalized so that they have the same prior importance as a typical decision rule:

$$l_j(x_j) = 0.4 \cdot l_j^*(x_j)/std(l_j^*(x_j))$$

The 0.4 is the average standard deviation of rules with a uniform support distribution of $s_k \sim U(0, 1)$.

We combine both types of features to generate a new feature matrix and train a sparse linear model with Lasso, with the following structure:

$$\hat{f}(x) = \hat{\beta}_0 + \sum_{k=1}^{K} \hat{\alpha}_k r_k(x) + \sum_{j=1}^{p} \hat{\beta}_j l_j(x_j)$$

where $\hat{\alpha}$ is the estimated weight vector for the rule features and $\hat{\beta}$ the weight vector for the original features. Since RuleFit uses Lasso, the loss function gets the additional constraint that forces some of the weights to get a zero estimate:

$$(\{\hat{\alpha}\}_1^K, \{\hat{\beta}\}_0^p) = argmin_{\{\hat{\alpha}\}_1^K, \{\hat{\beta}\}_0^p} \sum_{i=1}^{n} L(y^{(i)}, f(x^{(i)})) + \lambda \cdot \left(\sum_{k=1}^{K} |\alpha_k| + \sum_{j=1}^{p} |b_j| \right)$$

The result is a linear model that has linear effects for all of the original features and for the rules. The interpretation is the same as for linear models, the only difference is that some features are now binary rules.

Step 3 (optional): Feature importance

For the linear terms of the original features, the feature importance is measured with the standardized predictor:

$$I_j = |\hat{\beta}_j| \cdot std(l_j(x_j))$$

where β_j is the weight from the Lasso model and $std(l_j(x_j))$ is the standard deviation of the linear term over the data.

For the decision rule terms, the importance is calculated with the following formula:

$$I_k = |\hat{\alpha}_k| \cdot \sqrt{s_k(1 - s_k)}$$

where $\hat{\alpha}_k$ is the associated Lasso weight of the decision rule and s_k is the support of the feature in the data, which is the percentage of data points to which the decision rule applies (where $r_k(x) = 0$):

$$s_k = \frac{1}{n} \sum_{i=1}^{n} r_k(x^{(i)})$$

A feature occurs as a linear term and possibly also within many decision rules. How do we measure the total importance of a feature? The importance $J_j(x)$ of a feature can be measured for each individual prediction:

$$J_j(x) = I_l(x) + \sum_{x_j \in r_k} I_k(x)/m_k$$

where I_l is the importance of the linear term and I_k the importance of the decision rules in which x_j

appears, and m_k is the number of features constituting the rule r_k. Adding the feature importance from all instances gives us the global feature importance:

$$J_j(X) = \sum_{i=1}^{n} J_j(x^{(i)})$$

It is possible to select a subset of instances and calculate the feature importance for this group.

Advantages

RuleFit automatically adds **feature interactions** to linear models. Therefore, it solves the problem of linear models that you have to add interaction terms manually and it helps a bit with the issue of modeling nonlinear relationships.

RuleFit can handle both classification and regression tasks.

The rules created are easy to interpret, because they are binary decision rules. Either the rule applies to an instance or not. Good interpretability is only guaranteed if the number of conditions within a rule is not too large. A rule with 1 to 3 conditions seems reasonable to me. This means a maximum depth of 3 for the trees in the tree ensemble.

Even if there are many rules in the model, they do not apply to every instance. For an individual instance only a handful of rules apply (= have a non-zero weights). This improves local interpretability.

RuleFit proposes a bunch of useful diagnostic tools. These tools are model-agnostic, so you can find them in the model-agnostic section of the book: feature importance, partial dependence plots and feature interactions.

Disadvantages

Sometimes RuleFit creates many rules that get a non-zero weight in the Lasso model. The interpretability degrades with increasing number of features in the model. A promising solution is to force feature effects to be monotonic, meaning that an increase of a feature has to lead to an increase of the prediction.

An anecdotal drawback: The papers claim a good performance of RuleFit – often close to the predictive performance of random forests! – but in the few cases where I tried it personally, the performance was disappointing. Just try it out for your problem and see how it performs.

The end product of the RuleFit procedure is a linear model with additional fancy features (the decision rules). But since it is a linear model, the weight interpretation is still unintuitive. It comes with the same "footnote" as a usual linear regression model: "... given all features are fixed." It gets a bit more tricky when you have overlapping rules. For example, one decision rule (feature) for the bicycle prediction could be: "temp > 10" and another rule could be "temp > 15 & weather='GOOD'". If the weather is good and the temperature is above 15 degrees, the temperature is automatically greater

then 10. In the cases where the second rule applies, the first rule applies as well. The interpretation of the estimated weight for the second rule is: "Assuming all other features remain fixed, the predicted number of bikes increases by β_2 when the weather is good and temperature above 15 degrees.". But, now it becomes really clear that the 'all other feature fixed' is problematic, because if rule 2 applies, also rule 1 applies and the interpretation is nonsensical.

Software and Alternative

The RuleFit algorithm is implemented in R by Fokkema and Christoffersen (2017)[58] and you can find a Python version on Github[59].

A very similar framework is scope-rules[60], a Python module that also extracts rules from ensembles. It differs in the way it learns the final rules: First, similar and duplicate rules are removed. Then scope-rules chooses rules based on recall and precision instead of relying on Lasso.

[58]Fokkema, Marjolein, and Benjamin Christoffersen. "Pre: Prediction rule ensembles". https://CRAN.R-project.org/package=pre (2017).
[59]https://github.com/christophM/rulefit
[60]https://github.com/scikit-learn-contrib/skope-rules

Other Interpretable Models

The list of interpretable models is constantly growing and of unknown size. It includes simple models such as linear models, decision trees and naive Bayes, but also more complex ones that combine or modify non-interpretable machine learning models to make them more interpretable. Especially publications on the latter type of models are currently being produced at high frequency and it is hard to keep up with developments. The book teases only the Naive Bayes classifier and k-nearest neighbors in this chapter.

Naive Bayes Classifier

The Naive Bayes classifier uses the Bayes' theorem of conditional probabilities. For each feature, it calculates the probability for a class depending on the value of the feature. The Naive Bayes classifier calculates the class probabilities for each feature independently, which is equivalent to a strong (= naive) assumption of independence of the features. Naive Bayes is a conditional probability model and models the probability of a class C_k as follows:

$$P(C_k|x) = \frac{1}{Z}P(C_k)\prod_{i=1}^{n}P(x_i|C_k)$$

The term Z is a scaling parameter that ensures that the sum of probabilities for all classes is 1 (otherwise they would not be probabilities). The conditional probability of a class is the class probability times the probability of each feature given the class, normalized by Z. This formula can be derived by using the Bayes' theorem.

Naive Bayes is an interpretable model because of the independence assumption. It can be interpreted on the modular level. It is very clear for each feature how much it contributes towards a certain class prediction, since we can interpret the conditional probability.

K-Nearest Neighbors

The k-nearest neighbor method can be used for regression and classification and uses the nearest neighbors of a data point for prediction. For classification, the k-nearest neighbor method assigns the most common class of the nearest neighbors of an instance. For regression, it takes the average of the outcome of the neighbors. The tricky parts are finding the right k and deciding how to measure the distance between instances, which ultimately defines the neighborhood.

The k-nearest neighbor model differs from the other interpretable models presented in this book because it is an instance-based learning algorithm. How can k-nearest neighbors be interpreted? First of all, there are no parameters to learn, so there is no interpretability on a modular level. Furthermore, there is a lack of global model interpretability because the model is inherently local and there are no global weights or structures explicitly learned. Maybe it is interpretable at the

local level? To explain a prediction, you can always retrieve the k neighbors that were used for the prediction. Whether the model is interpretable depends solely on the question whether you can 'interpret' a single instance in the dataset. If an instance consists of hundreds or thousands of features, then it is not interpretable, I would argue. But if you have few features or a way to reduce your instance to the most important features, presenting the k-nearest neighbors can give you good explanations.

Model-Agnostic Methods

Separating the explanations from the machine learning model (= model-agnostic interpretation methods) has some advantages (Ribeiro, Singh, and Guestrin 2016[61]). The great advantage of model-agnostic interpretation methods over model-specific ones is their flexibility. Machine learning developers are free to use any machine learning model they like when the interpretation methods can be applied to any model. Anything that builds on an interpretation of a machine learning model, such as a graphic or user interface, also becomes independent of the underlying machine learning model. Typically, not just one, but many types of machine learning models are evaluated to solve a task, and when comparing models in terms of interpretability, it is easier to work with model-agnostic explanations, because the same method can be used for any type of model.

An alternative to model-agnostic interpretation methods is to use only interpretable models, which often has the big disadvantage that predictive performance is lost compared to other machine learning models and you limit yourself to one type of model. The other alternative is to use model-specific interpretation methods. The disadvantage of this is that it also binds you to one model type and it will be difficult to switch to something else.

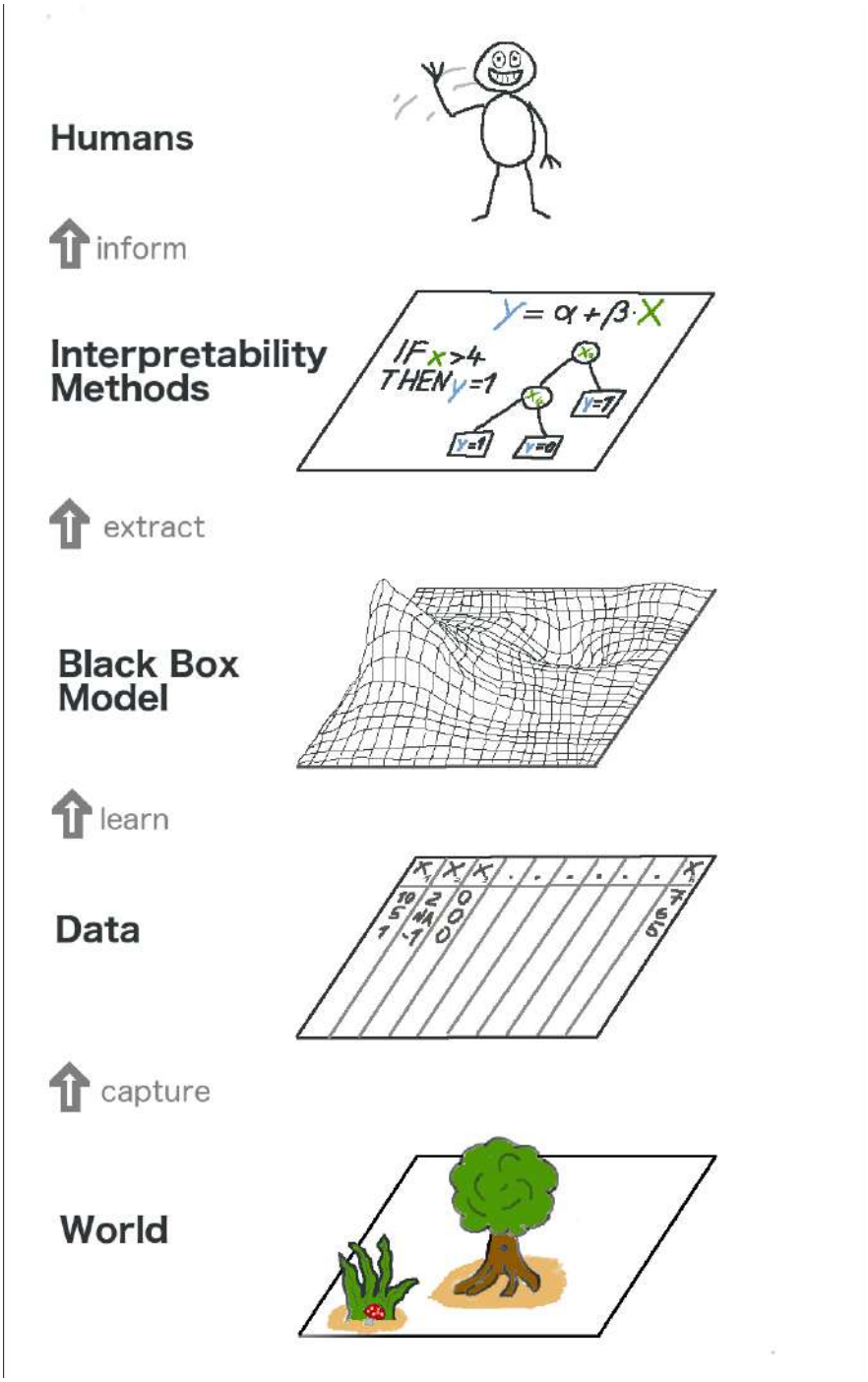
Desirable aspects of a model-agnostic explanation system are (Ribeiro, Singh, and Guestrin 2016):

- **Model flexibility**: The interpretation method can work with any machine learning model, such as random forests and deep neural networks.
- **Explanation flexibility**: You are not limited to a certain form of explanation. In some cases it might be useful to have a linear formula, in other cases a graphic with feature importances.
- **Representation flexibility**: The explanation system should be able to use a different feature representation as the model being explained. For a text classifier that uses abstract word embedding vectors, it might be preferable to use the presence of individual words for the explanation.

The bigger picture

Let us take a high level look at model-agnostic interpretability. We capture the world by collecting data, and abstract it further by learning to predict the data (for the task) with a machine learning model. Interpretability is just another layer on top that helps humans understand.

[61] Ribeiro, Marco Tulio, Sameer Singh, and Carlos Guestrin. "Model-agnostic interpretability of machine learning." ICML Workshop on Human Interpretability in Machine Learning. (2016).



The big picture of explainable machine learning. The real world goes through many layers before it reaches the human in the form of explanations.

The lowest layer is the **World**. This could literally be nature itself, like the biology of the human body and how it reacts to medication, but also more abstract things like the real estate market. The World layer contains everything that can be observed and is of interest. Ultimately, we want to learn something about the World and interact with it.

The second layer is the **Data** layer. We have to digitize the World in order to make it processable for computers and also to store information. The Data layer contains anything from images, texts, tabular data and so on.

By fitting machine learning models based on the Data layer, we get the **Black Box Model** layer. Machine learning algorithms learn with data from the real world to make predictions or find structures.

Above the Black Box Model layer is the **Interpretability Methods** layer, which helps us deal with the opacity of machine learning models. What were the most important features for a particular diagnosis? Why was a financial transaction classified as fraud?

The last layer is occupied by a **Human**. Look! This one waves to you because you are reading this book and helping to provide better explanations for black box models! Humans are ultimately the consumers of the explanations.

This multi-layered abstraction also helps to understand the differences in approaches between statisticians and machine learning practitioners. Statisticians deal with the Data layer, such as planning clinical trials or designing surveys. They skip the Black Box Model layer and go right to the Interpretability Methods layer. Machine learning specialists also deal with the Data layer, such as collecting labeled samples of skin cancer images or crawling Wikipedia. Then they train a black box machine learning model. The Interpretability Methods layer is skipped and humans directly deal with the black box model predictions. It's great that interpretable machine learning fuses the work of statisticians and machine learning specialists.

Of course this graphic does not capture everything: Data could come from simulations. Black box models also output predictions that might not even reach humans, but only supply other machines, and so on. But overall it is a useful abstraction to understand how interpretability becomes this new layer on top of machine learning models.

Partial Dependence Plot (PDP)

The partial dependence plot (short PDP or PD plot) shows the marginal effect one or two features have on the predicted outcome of a machine learning model (J. H. Friedman 2001[62]). A partial dependence plot can show whether the relationship between the target and a feature is linear, monotonous or more complex. For example, when applied to a linear regression model, partial dependence plots always show a linear relationship.

The partial dependence function for regression is defined as:

$$\hat{f}_{x_S}(x_S) = E_{x_C}\left[\hat{f}(x_S, x_C)\right] = \int \hat{f}(x_S, x_C) d\mathbb{P}(x_C)$$

The x_S are the features for which the partial dependence function should be plotted and x_C are the other features used in the machine learning model $\hat{f}$. Usually, there are only one or two features in the set S. The feature(s) in S are those for which we want to know the effect on the prediction. The feature vectors x_S and x_C combined make up the total feature space x. Partial dependence works by marginalizing the machine learning model output over the distribution of the features in set C, so that the function shows the relationship between the features in set S we are interested in and the predicted outcome. By marginalizing over the other features, we get a function that depends only on features in S, interactions with other features included.

The partial function $\hat{f}_{x_S}$ is estimated by calculating averages in the training data, also known as Monte Carlo method:

$\hat{f}_{x_S}(x_S) = \frac{1}{n}\sum_{i=1}^{n}\hat{f}(x_S, x_C^{(i)})$ The partial function tells us for given value(s) of features S what the average marginal effect on the prediction is. In this formula, $x_C^{(i)}$ are actual feature values from the dataset for the features in which we are not interested, and n is the number of instances in the dataset. An assumption of the PDP is that the features in C are not correlated with the features in S. If this assumption is violated, the averages calculated for the partial dependence plot will include data points that are very unlikely or even impossible (see disadvantages).

For classification where the machine learning model outputs probabilities, the partial dependence plot displays the probability for a certain class given different values for feature(s) in S. An easy way to deal with multiple classes is to draw one line or plot per class.

The partial dependence plot is a global method: The method considers all instances and gives a statement about the global relationship of a feature with the predicted outcome.

Categorical features

So far, we have only considered numerical features. For categorical features, the partial dependence is very easy to calculate. For each of the categories, we get a PDP estimate by forcing all data instances to have the same category. For example, if we look at the bike rental dataset and are interested in the partial dependence plot for the season, we get 4 numbers, one for each season. To compute the

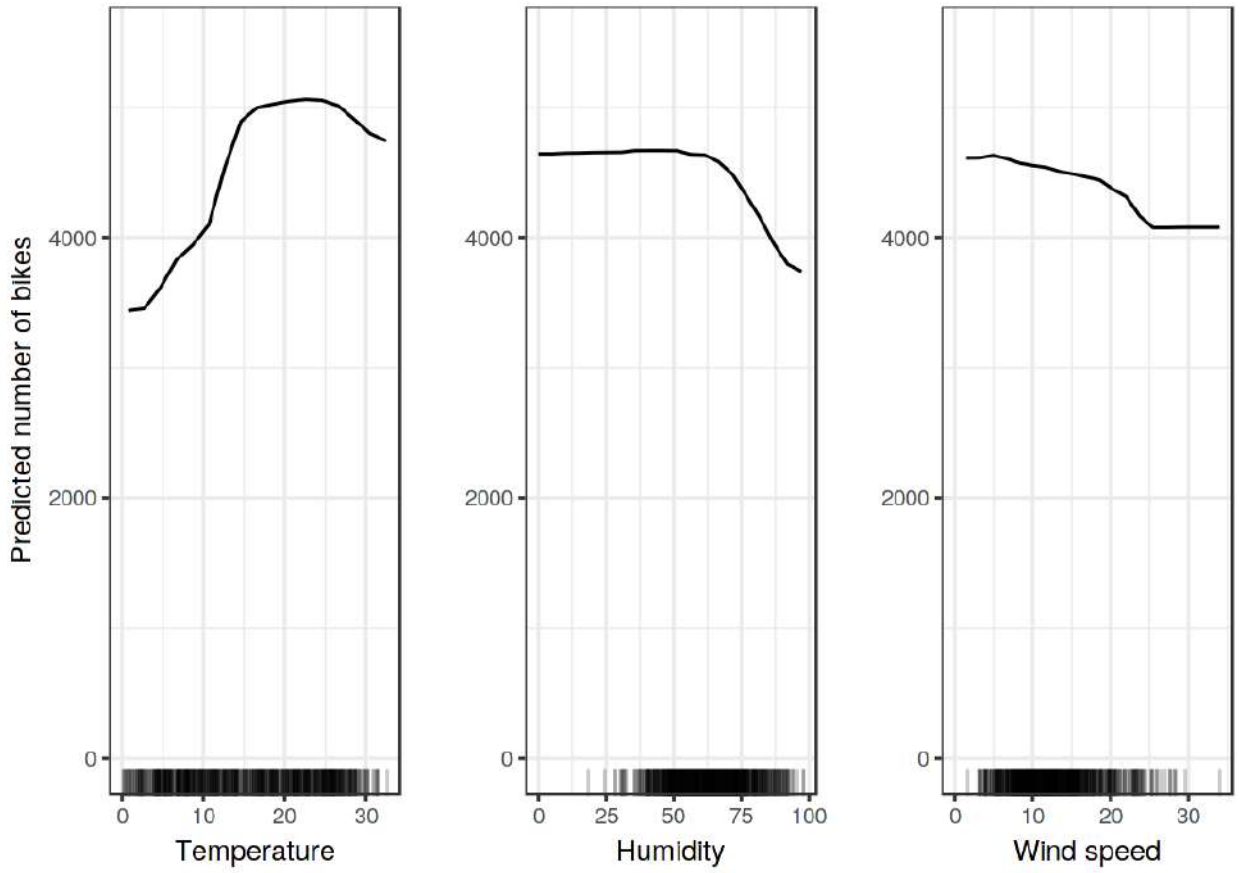
[62]Friedman, Jerome H. "Greedy function approximation: A gradient boosting machine." Annals of statistics (2001): 1189-1232.

value for "summer", we replace the season of all data instances with "summer" and average the predictions.

Examples

In practice, the set of features S usually only contains one feature or a maximum of two, because one feature produces 2D plots and two features produce 3D plots. Everything beyond that is quite tricky. Even 3D on a 2D paper or monitor is already challenging.

Let us return to the regression example, in which we predict the number of bikes that will be rented on a given day. First we fit a machine learning model, then we analyze the partial dependencies. In this case, we have fitted a random forest to predict the number of bicycles and use the partial dependence plot to visualize the relationships the model has learned. The influence of the weather features on the predicted bike counts is visualized in the following figure.

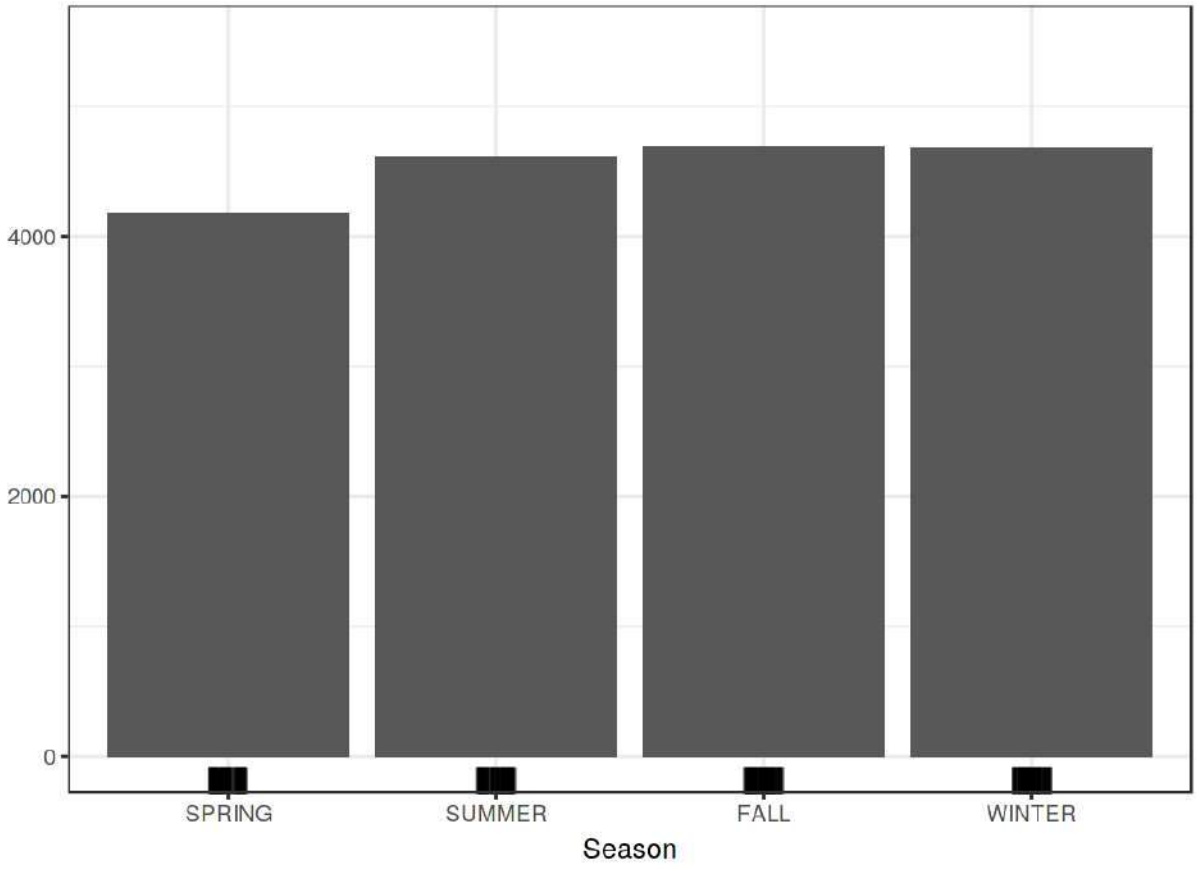


PDPs for the bicycle count prediction model and temperature, humidity and wind speed. The largest differences can be seen in the temperature. The hotter, the more bikes are rented. This trend goes up to 20 degrees Celsius, then flattens and drops slightly at 30. Marks on the x-axis indicate the data distribution.

For warm but not too hot weather, the model predicts on average a high number of rented bicycles. Potential bikers are increasingly inhibited in renting a bike when humidity exceeds 60%. In addition,

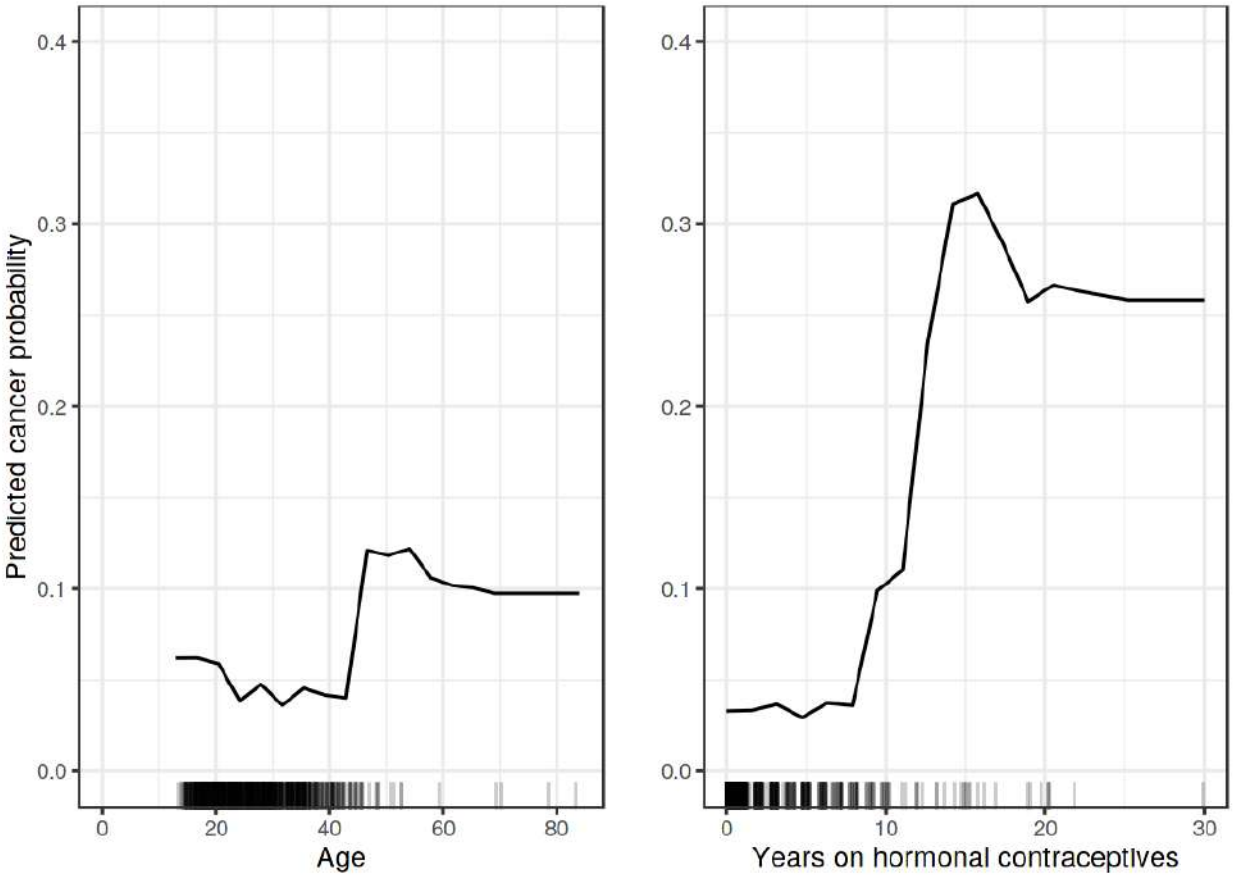
the more wind the fewer people like to cycle, which makes sense. Interestingly, the predicted number of bike rentals does not fall when wind speed increases from 25 to 35 km/h, but there is not much training data, so the machine learning model could probably not learn a meaningful prediction for this range. At least intuitively, I would expect the number of bicycles to decrease with increasing wind speed, especially when the wind speed is very high.

To illustrate a partial dependence plot with a categorical feature, we examine the effect of the season feature on the predicted bike rentals.



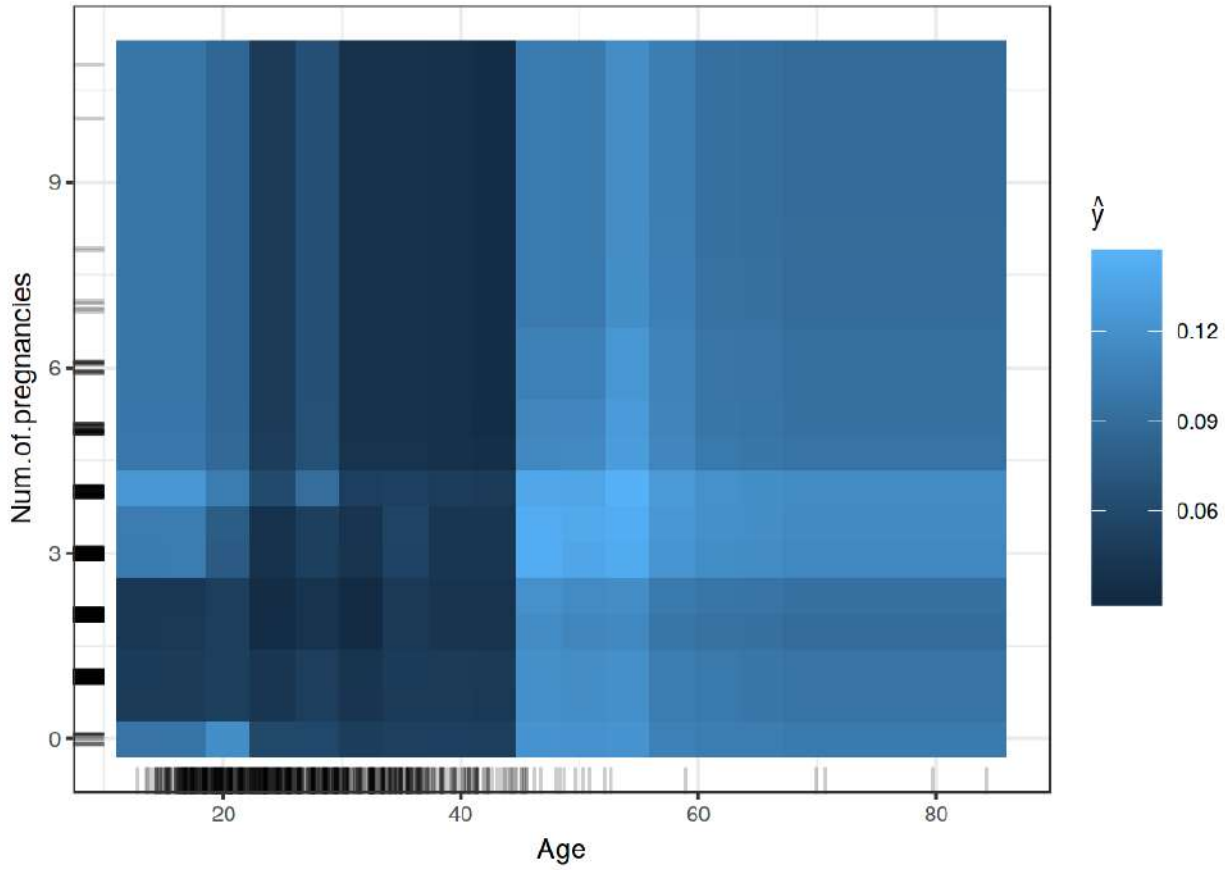
PDPs for the bike count prediction model and the season. Unexpectedly all seasons show the same effect, only for spring the model predicts less bicycle rentals.

We also compute the partial dependence for cervical cancer classification. This time we fit a random forest to predict whether a woman might get cervical cancer based on risk factors. We compute and visualize the partial dependence of the cancer probability on different features for the random forest:



PDPs of cancer probability based on age and years with hormonal contraceptives. For age, the PDP shows that the probability is low until 40 and increases after. The more years on hormonal contraceptives the higher the predicted cancer risk, especially after 10 years. For both features not many data points with large values were available, so the PD estimates are less reliable in those regions.

We can also visualize the partial dependence of two features at once:



PDP of cancer probability and the interaction of age and number of pregnancies. The plot shows the increase in cancer probability at 45. For ages below 25, women who had 1 or 2 pregnancies have a lower predicted cancer risk, compared with women who had 0 or more than 2 pregnancies. But be careful when drawing conclusions: This might just be a correlation and not causal!

Advantages

The computation of partial dependence plots is **intuitive**: The partial dependence function at a particular feature value represents the average prediction if we force all data points to assume that feature value. In my experience, lay people usually understand the idea of PDPs quickly.

If the feature for which you computed the PDP is not correlated with the other features, then the PDPs perfectly represent how the feature influences the prediction on average. In the uncorrelated case, the **interpretation is clear**: The partial dependence plot shows how the average prediction in your dataset changes when the j-th feature is changed. It is more complicated when features are correlated, see also disadvantages.

Partial dependence plots are **easy to implement**.

The calculation for the partial dependence plots has a **causal interpretation**. We intervene on a feature and measure the changes in the predictions. In doing so, we analyze the causal relationship

between the feature and the prediction.[63] The relationship is causal for the model – because we explicitly model the outcome as a function of the features – but not necessarily for the real world!

Disadvantages

The realistic **maximum number of features** in a partial dependence function is two. This is not the fault of PDPs, but of the 2-dimensional representation (paper or screen) and also of our inability to imagine more than 3 dimensions.

Some PD plots do not show the **feature distribution**. Omitting the distribution can be misleading, because you might overinterpret regions with almost no data. This problem is easily solved by showing a rug (indicators for data points on the x-axis) or a histogram.

The **assumption of independence** is the biggest issue with PD plots. It is assumed that the feature(s) for which the partial dependence is computed are not correlated with other features. For example, suppose you want to predict how fast a person walks, given the person's weight and height. For the partial dependence of one of the features, e.g. height, we assume that the other features (weight) are not correlated with height, which is obviously a false assumption. For the computation of the PDP at a certain height (e.g. 200 cm), we average over the marginal distribution of weight, which might include a weight below 50 kg, which is unrealistic for a 2 meter person. In other words: When the features are correlated, we create new data points in areas of the feature distribution where the actual probability is very low (for example it is unlikely that someone is 2 meters tall but weighs less than 50 kg). One solution to this problem is Accumulated Local Effect plots or short ALE plots that work with the conditional instead of the marginal distribution.

Heterogeneous effects might be hidden because PD plots only show the average marginal effects. Suppose that for a feature half your data points have a positive association with the prediction – the larger the feature value the larger the prediction – and the other half has a negative association – the smaller the feature value the larger the prediction. The PD curve could be a horizontal line, since the effects of both halves of the dataset could cancel each other out. You then conclude that the feature has no effect on the prediction. By plotting the individual conditional expectation curves instead of the aggregated line, we can uncover heterogeneous effects.

Software and Alternatives

There are a number of R packages that implement PDPs. I used the `iml` package for the examples, but there is also `pdp` or `DALEX`. In Python you can use `Skater`.

Alternatives to PDPs presented in this book are ALE plots and ICE curves.

[63] Zhao, Qingyuan, and Trevor Hastie. "Causal interpretations of black-box models." Journal of Business & Economic Statistics, to appear. (2017).

Individual Conditional Expectation (ICE)

Individual Conditional Expectation (ICE) plots display one line per instance that shows how the instance's prediction changes when a feature changes.

The partial dependence plot for the average effect of a feature is a global method because it does not focus on specific instances, but on an overall average. The equivalent to a PDP for individual data instances is called individual conditional expectation (ICE) plot (Goldstein et al. 2017[64]). An ICE plot visualizes the dependence of the prediction on a feature for *each* instance separately, resulting in one line per instance, compared to one line overall in partial dependence plots. A PDP is the average of the lines of an ICE plot. The values for a line (and one instance) can be computed by keeping all other features the same, creating variants of this instance by replacing the feature's value with values from a grid and making predictions with the black box model for these newly created instances. The result is a set of points for an instance with the feature value from the grid and the respective predictions.

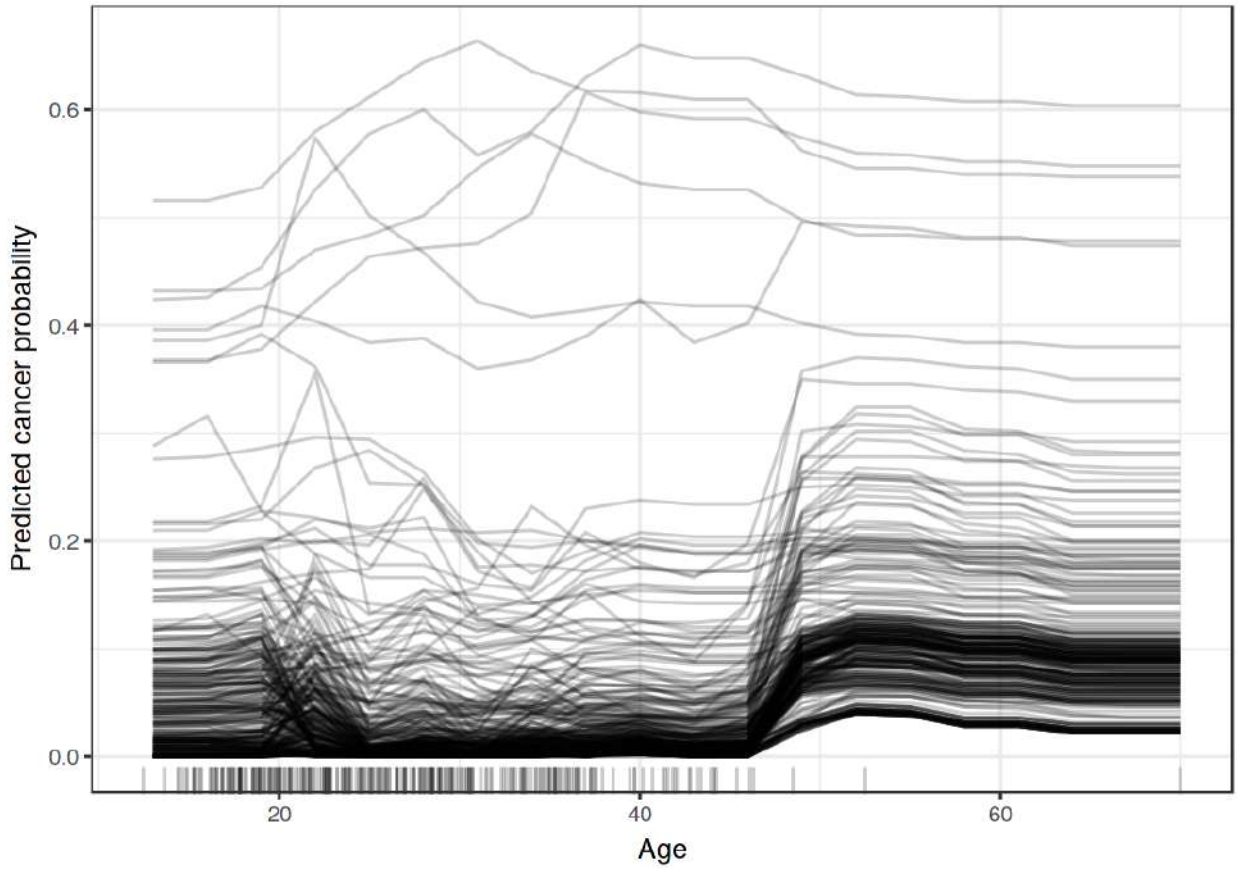
What is the point of looking at individual expectations instead of partial dependencies? Partial dependence plots can obscure a heterogeneous relationship created by interactions. PDPs can show you what the average relationship between a feature and the prediction looks like. This only works well if the interactions between the features for which the PDP is calculated and the other features are weak. In case of interactions, the ICE plot will provide much more insight.

A more formal definition: In ICE plots, for each instance in $\{(x_S^{(i)}, x_C^{(i)})\}_{i=1}^N$ the curve $\hat{f}_S^{(i)}$ is plotted against $x_S^{(i)}$, while $x_C^{(i)}$ remains fixed.

Examples

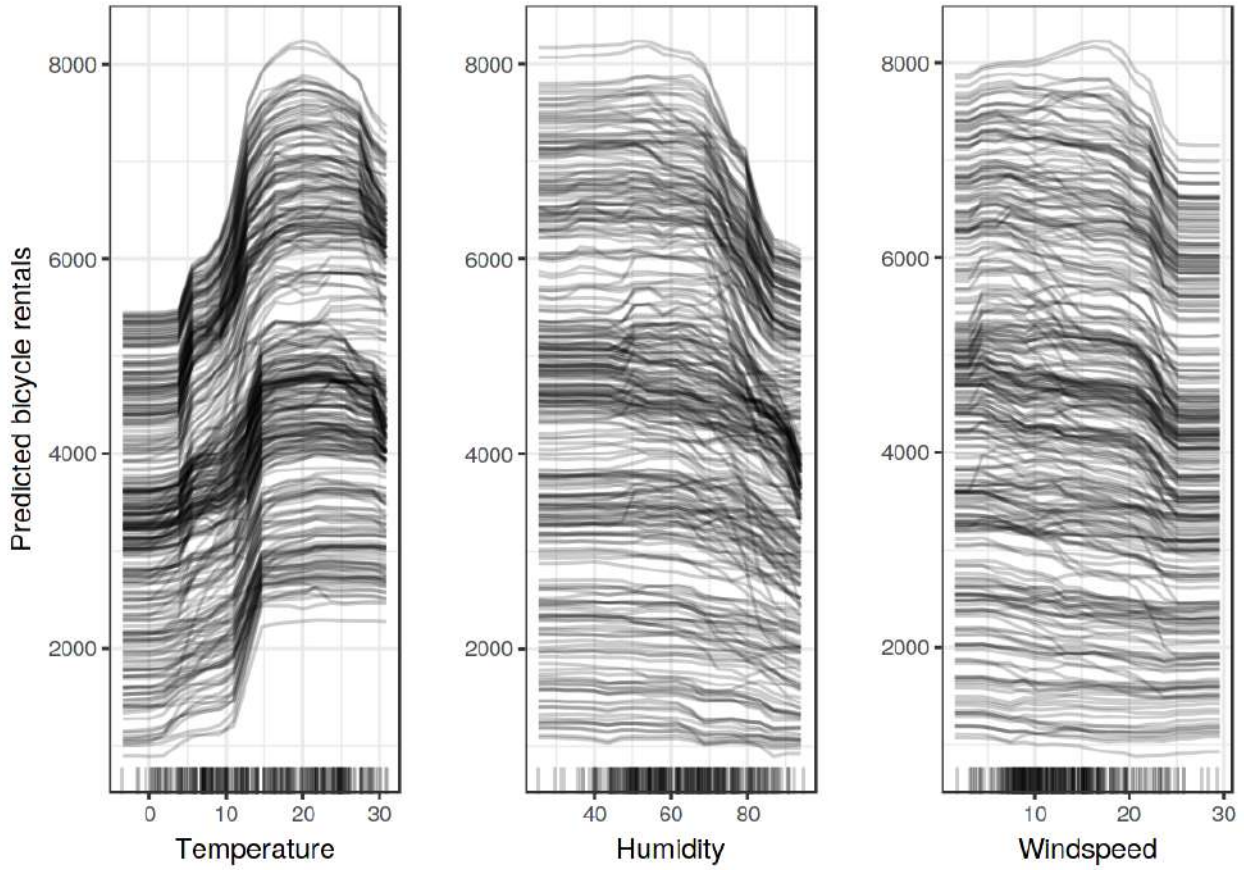
Let's go back to the cervical cancer dataset and see how the prediction of each instance is associated with the feature "Age". We will analyze a random forest that predicts the probability of cancer for a woman given risk factors. In the partial dependence plot we have seen that the cancer probability increases around the age of 50, but is this true for every woman in the dataset? The ICE plot reveals that for most women the age effect follows the average pattern of an increase at age 50, but there are some exceptions: For the few women that have a high predicted probability at a young age, the predicted cancer probability does not change much with age.

[64] Goldstein, Alex, et al. "Package â€˜ICEboxâ€™." (2017).



ICE plot of cervical cancer probability by age. Each line represents one woman. For most women there is an increase in predicted cancer probability with increasing age. For some women with a predicted cancer probability above 0.4, the prediction does not change much at higher age.

The next figure shows ICE plots for the bike rental prediction. The underlying prediction model is a random forest.



ICE plots of predicted bicycle rentals by weather conditions. The same effects can be observed as in the partial dependence plots.

All curves seem to follow the same course, so there are no obvious interactions. That means that the PDP is already a good summary of the relationships between the displayed features and the predicted number of bicycles

Centered ICE Plot

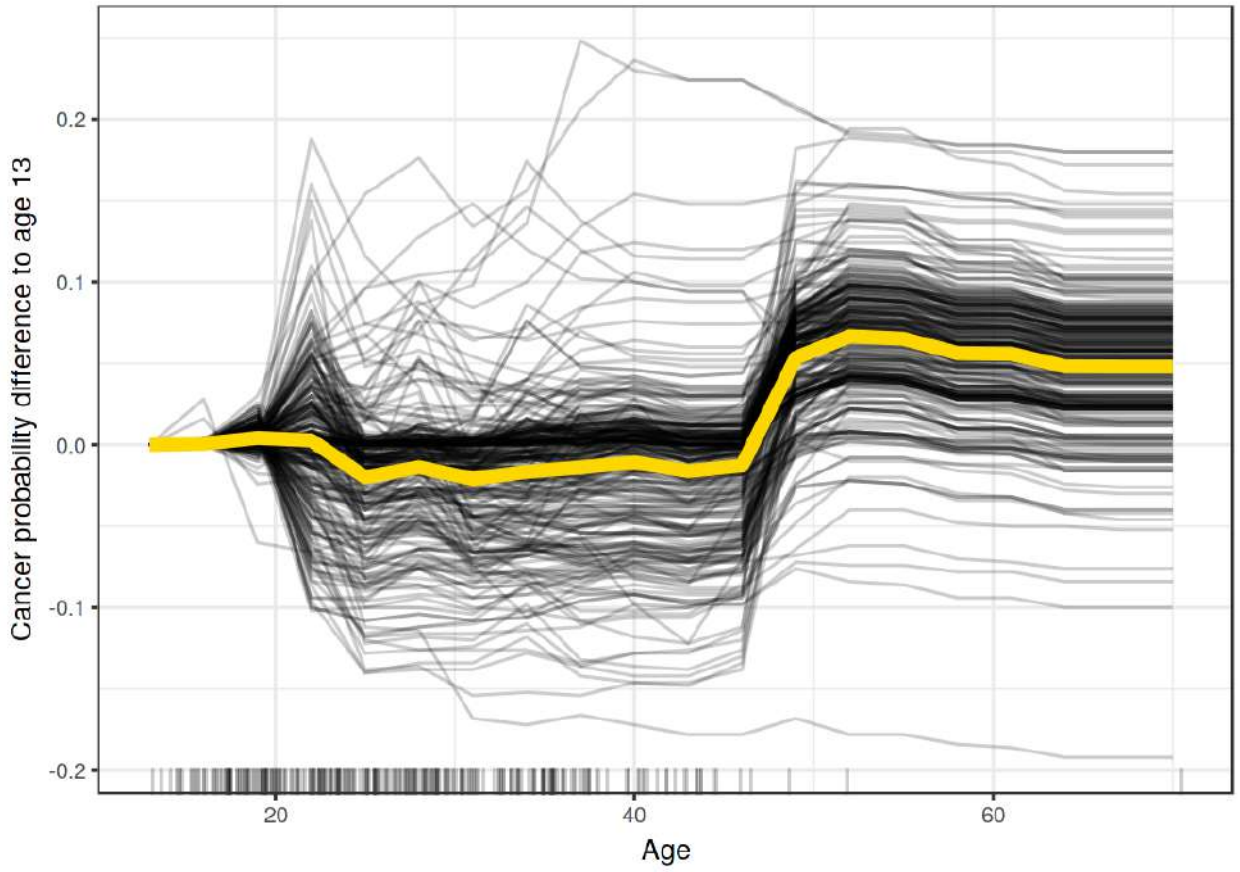
There is a problem with ICE plots: Sometimes it can be hard to tell whether the ICE curves differ between individuals because they start at different predictions. A simple solution is to center the curves at a certain point in the feature and display only the difference in the prediction to this point. The resulting plot is called centered ICE plot (c-ICE). Anchoring the curves at the lower end of the feature is a good choice. The new curves are defined as:

$$\hat{f}_{cent}^{(i)} = \hat{f}^{(i)} - \mathbf{1}\hat{f}(x^a, x_C^{(i)})$$

where $\mathbf{1}$ is a vector of 1's with the appropriate number of dimensions (usually one or two), $\hat{f}$ is the fitted model and x^a is the anchor point.

Example

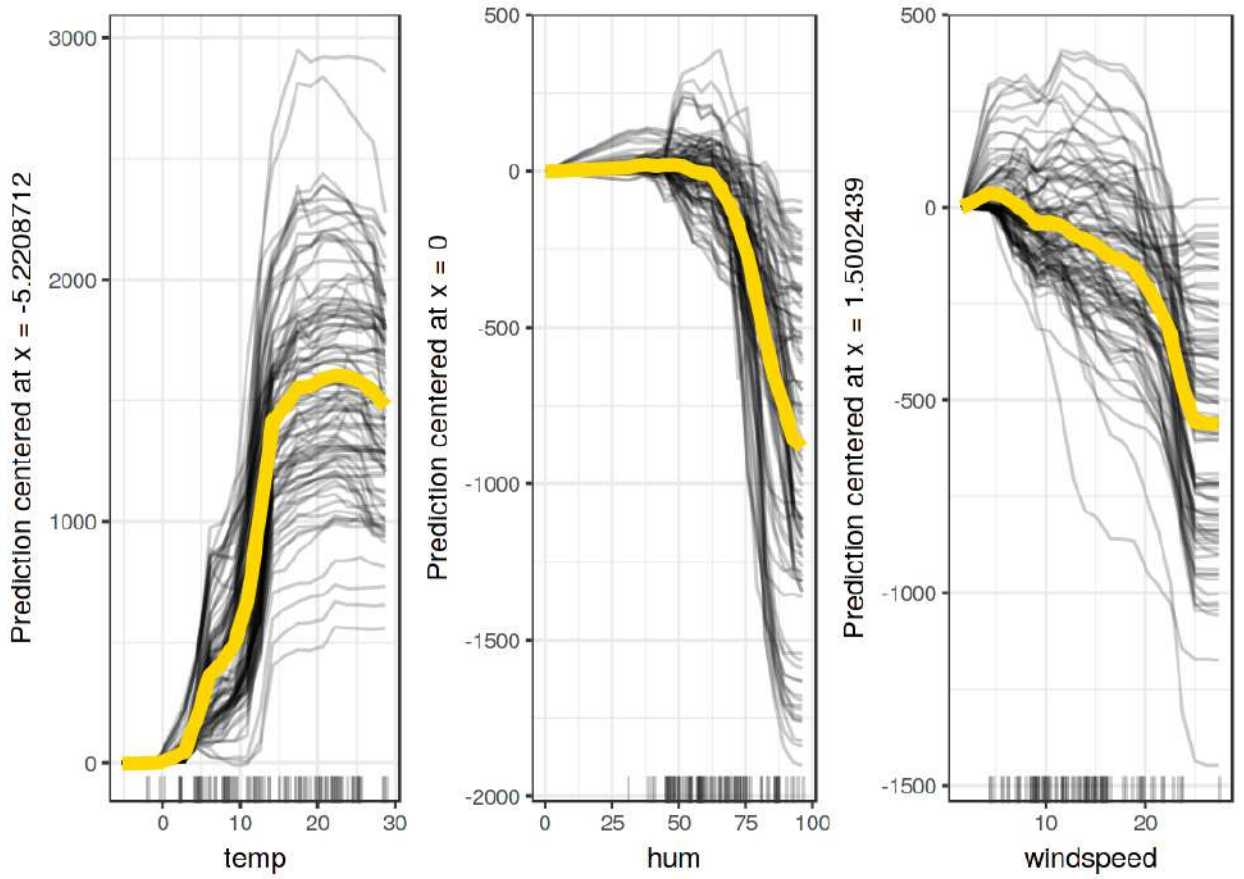
For example, take the cervical cancer ICE plot for age and center the lines on the youngest observed age:



Centered ICE plot for predicted cancer probability by age. Lines are fixed to 0 at age 13. Compared to age 13, the predictions for most women remain unchanged until the age of 45 where the predicted probability increases.

The centered ICE plots make it easier to compare the curves of individual instances. This can be useful if we do not want to see the absolute change of a predicted value, but the difference in the prediction compared to a fixed point of the feature range.

Let's have a look at centered ICE plots for the bicycle rental prediction:



Centered ICE plots of predicted number of bikes by weather condition. The lines show the difference in prediction compared to the prediction with the respective feature value at its observed minimum.

Derivative ICE Plot

Another way to make it visually easier to spot heterogeneity is to look at the individual derivatives of the prediction function with respect to a feature. The resulting plot is called the derivative ICE plot (d-ICE). The derivatives of a function (or curve) tell you whether changes occur and in which direction they occur. With the derivative ICE plot, it is easy to spot ranges of feature values where the black box predictions change for (at least some) instances. If there is no interaction between the analyzed feature x_S and the other features x_C, then the prediction function can be expressed as:

$$\hat{f}(x) = \hat{f}(x_S, x_C) = g(x_S) + h(x_C), \quad \text{with} \quad \frac{\delta \hat{f}(x)}{\delta x_S}$$

Without interactions, the individual partial derivatives should be the same for all instances. If they differ, it is due to interactions and it becomes visible in the d-ICE plot. In addition to displaying the individual curves for the derivative of the prediction function with respect to the feature in S, showing the standard deviation of the derivative helps to highlight regions in feature in S with

heterogeneity in the estimated derivatives. The derivative ICE plot takes a long time to compute and is rather impractical.

Advantages

Individual conditional expectation curves are **even more intuitive to understand** than partial dependence plots. One line represents the predictions for one instance if we vary the feature of interest.

Unlike partial dependence plots, ICE curves can **uncover heterogeneous relationships**.

Disadvantages

ICE curves **can only display one feature** meaningfully, because two features would require the drawing of several overlaying surfaces and you would not see anything in the plot.

ICE curves suffer from the same problem as PDPs: If the feature of interest is correlated with the other features, then **some points in the lines might be invalid data points** according to the joint feature distribution.

If many ICE curves are drawn, the **plot can become overcrowded** and you will not see anything. The solution: Either add some transparency to the lines or draw only a sample of the lines.

In ICE plots it might not be easy to **see the average**. This has a simple solution: Combine individual conditional expectation curves with the partial dependence plot.

Software and Alternatives

ICE plots are implemented in the R packages `iml` (used for these examples), `ICEbox`, and `pdp`. Another R package that does something very similar to ICE is `condvis`.

Accumulated Local Effects (ALE) Plot

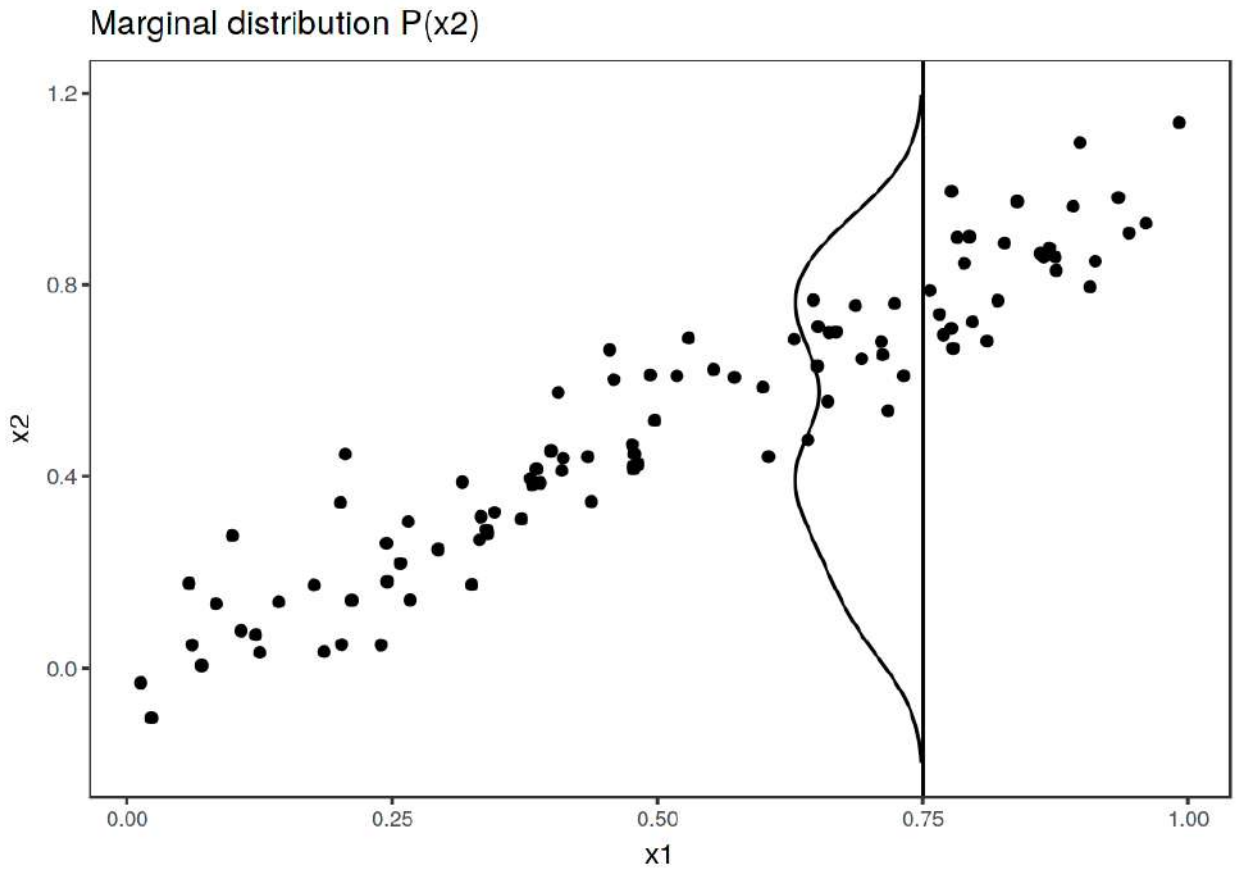
Accumulated local effects[65] describe how features influence the prediction of a machine learning model on average. ALE plots are a faster and unbiased alternative to partial dependence plots (PDPs).

I recommend reading the chapter on partial dependence plots first, as they are easier to understand and both methods share the same goal: Both describe how a feature affects the prediction on average. In the following section, I want to convince you that partial dependence plots have a serious problem when the features are correlated.

Motivation and Intuition

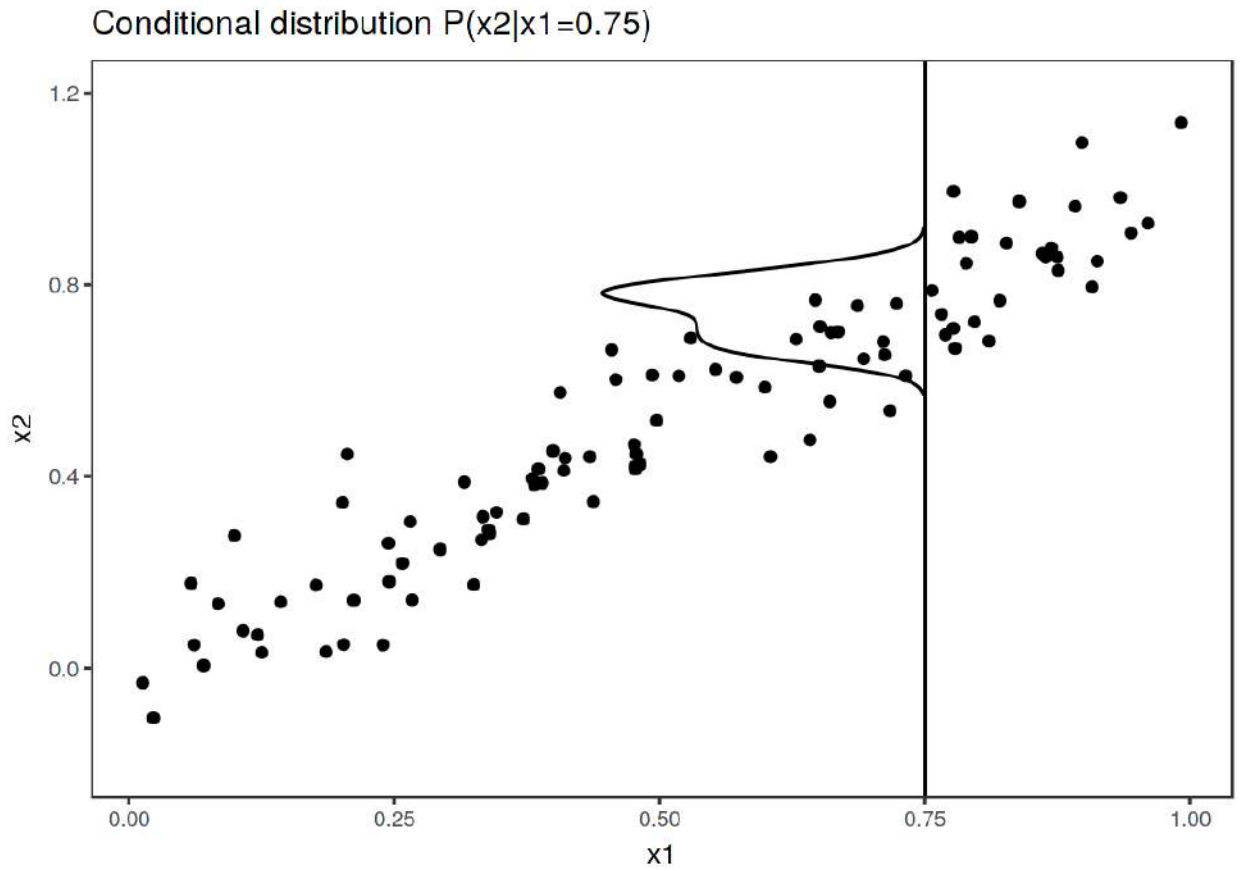
If features of a machine learning model are correlated, the partial dependence plot cannot be trusted. The computation of a partial dependence plot for a feature that is strongly correlated with other features involves averaging predictions of artificial data instances that are unlikely in reality. This can greatly bias the estimated feature effect. Imagine calculating partial dependence plots for a machine learning model that predicts the value of a house depending on the number of rooms and the size of the living area. We are interested in the effect of the living area on the predicted value. As a reminder, the recipe for partial dependence plots is: 1) Select feature. 2) Define grid. 3) Per grid value: a) Replace feature with grid value and b) average predictions. 4) Draw curve. For the calculation of the first grid value of the PDP – say 30 m² – we replace the living area for **all** instances by 30 m², even for houses with 10 rooms. Sounds to me like a very unusual house. The partial dependence plot includes these unrealistic houses in the feature effect estimation and pretends that everything is fine. The following figure illustrates two correlated features and how it comes that the partial dependence plot method averages predictions of unlikely instances.

[65] Apley, Daniel W. "Visualizing the effects of predictor variables in black box supervised learning models." arXiv preprint arXiv:1612.08468 (2016).



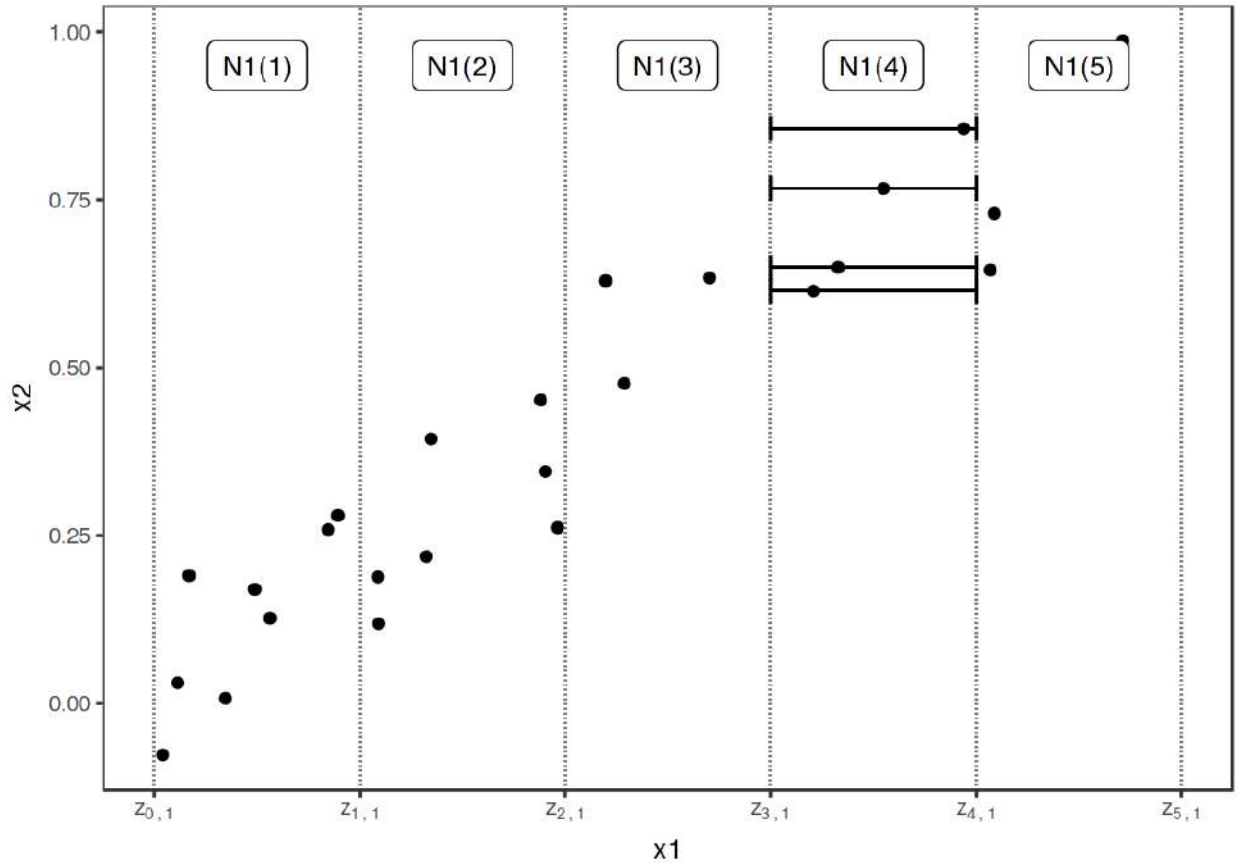
Strongly correlated features x1 and x2. To calculate the feature effect of x1 at 0.75, the PDP replaces x1 of all instances with 0.75, falsely assuming that the distribution of x2 at x1 = 0.75 is the same as the marginal distribution of x2 (vertical line). This results in unlikely combinations of x1 and x2 (e.g. x2=0.2 at x1=0.75), which the PDP uses for the calculation of the average effect.

What can we do to get a feature effect estimate that respects the correlation of the features? We could average over the conditional distribution of the feature, meaning at a grid value of x1, we average the predictions of instances with a similar x1 value. The solution for calculating feature effects using the conditional distribution is called Marginal Plots, or M-Plots (confusing name, since they are based on the conditional, not the marginal distribution). Wait, did I not promise you to talk about ALE plots? M-Plots are not the solution we are looking for. Why do M-Plots not solve our problem? If we average the predictions of all houses of about 30 m^2, we estimate the **combined** effect of living area and of number of rooms, because of their correlation. Suppose that the living area has no effect on the predicted value of a house, only the number of rooms has. The M-Plot would still show that the size of the living area increases the predicted value, since the number of rooms increases with the living area. The following plot shows for two correlated features how M-Plots work.



Strongly correlated features x1 and x2. M-Plots average over the conditional distribution. Here the conditional distribution of x2 at x1 = 0.75. Averaging the local predictions leads to mixing the effects of both features.

M-Plots avoid averaging predictions of unlikely data instances, but they mix the effect of a feature with the effects of all correlated features. ALE plots solve this problem by calculating – also based on the conditional distribution of the features – **differences in predictions instead of averages**. For the effect of living area at 30 m^2, the ALE method uses all houses with about 30 m^2, gets the model predictions pretending these houses were 31 m^2 minus the prediction pretending they were 29 m^2. This gives us the pure effect of the living area and is not mixing the effect with the effects of correlated features. The use of differences blocks the effect of other features. The following graphic provides intuition how ALE plots are calculated.



Calculation of ALE for feature x1, which is correlated with x2. First, we divide the feature into intervals (vertical lines). For the data instances (points) in an interval, we calculate the difference in the prediction when we replace the feature with the upper and lower limit of the interval (horizontal lines). These differences are later accumulated and centered, resulting in the ALE curve.

To summarize how each type of plot (PDP, M, ALE) calculates the effect of a feature at a certain grid value v:

Partial Dependence Plots: "Let me show you what the model predicts on average when each data instance has the value v for that feature. I ignore whether the value v makes sense for all data instances."

M-Plots: "Let me show you what the model predicts on average for data instances that have values close to v for that feature. The effect could be due to that feature, but also due to correlated features."

ALE plots: "Let me show you how the model predictions change in a small "window" of the feature around v for data instances in that window."

Theory

How do PD, M and ALE plots differ mathematically? Common to all three methods is that they reduce the complex prediction function f to a function that depends on only one (or two) features. All three methods reduce the function by averaging the effects of the other features, but they differ

in whether averages of predictions or of **differences in predictions** are calculated and whether averaging is done over the marginal or conditional distribution.

Partial dependence plots average the predictions over the marginal distribution.

$$\begin{aligned}\hat{f}_{x_S,PDP}(x_S) &= E_{X_C}\left[\hat{f}(x_S, X_C)\right] \\ &= \int_{x_C} \hat{f}(x_S, x_C)\mathbb{P}(x_C)dx_C\end{aligned}$$

This is the value of the prediction function f, at feature value(s) x_S, averaged over all features in x_C. Averaging means calculating the marginal expectation E over the features in set C, which is the integral over the predictions weighted by the probability distribution. Sounds fancy, but to calculate the expected value over the marginal distribution, we simply take all our data instances, force them to have a certain grid value for the features in set S, and average the predictions for this manipulated dataset. This procedure ensures that we average over the marginal distribution of the features.

M-plots average the predictions over the conditional distribution.

$$\begin{aligned}\hat{f}_{x_S,M}(x_S) &= E_{X_C|X_S}\left[\hat{f}(X_S, X_C)|X_S = x_s\right] \\ &= \int_{x_C} \hat{f}(x_S, x_C)\mathbb{P}(x_C|x_S)dx_C\end{aligned}$$

The only thing that changes compared to PDPs is that we average the predictions conditional on each grid value of the feature of interest, instead of assuming the marginal distribution at each grid value. In practice, this means that we have to define a neighborhood, for example for the calculation of the effect of 30 m² on the predicted house value, we could average the predictions of all houses between 28 and 32 m².

ALE plots average the changes in the predictions and accumulate them over the grid (more on the calculation later).

$$\begin{aligned}\hat{f}_{x_S,ALE}(x_S) &= \int_{z_{0,1}}^{x_S} E_{X_C|X_S}\left[\hat{f}^S(X_s, X_c)|X_S = z_S\right] dz_S - \text{constant} \\ &= \int_{z_{0,1}}^{x_S}\int_{x_C} \hat{f}^S(z_s, x_c)\mathbb{P}(x_C|z_S)dx_C dz_S - \text{constant}\end{aligned}$$

The formula reveals three differences to M-Plots. First, we average the changes of predictions, not the predictions itself. The change is defined as the gradient (but later, for the actual computation, replaced by the differences in the predictions over an interval).

$$\hat{f}^S(x_s, x_c) = \frac{\delta\hat{f}(x_S, x_C)}{\delta x_S}$$

The second difference is the additional integral over z. We accumulate the local gradients over the range of features in set S, which gives us the effect of the feature on the prediction. For the actual computation, the z's are replaced by a grid of intervals over which we compute the changes in the prediction. Instead of directly averaging the predictions, the ALE method calculates the prediction differences conditional on features S and integrates the derivative over features S to estimate the effect. Well, that sounds stupid. Derivation and integration usually cancel each other out, like first subtracting, then adding the same number. Why does it make sense here? The derivative (or interval difference) isolates the effect of the feature of interest and blocks the effect of correlated features.

The third difference of ALE plots to M-plots is that we subtract a constant from the results. This step centers the ALE plot so that the average effect over the data is zero.

One problem remains: Not all models come with a gradient, for example random forests have no gradient. But as you will see, the actual computation works without gradients and uses intervals. Let us delve a little deeper into the estimation of ALE plots.

Estimation

First I will describe how ALE plots are estimated for a single numerical feature, later for two numerical features and for a single categorical feature. To estimate local effects, we divide the feature into many intervals and compute the differences in the predictions. This procedure approximates the gradients and also works for models without gradients.

First we estimate the uncentered effect:

$$\hat{\tilde{f}}_{j,ALE}(x) = \sum_{k=1}^{k_j(x)} \frac{1}{n_j(k)} \sum_{i:x_j^{(i)} \in N_j(k)} \left[f(z_{k,j}, x_{\setminus j}^{(i)}) - f(z_{k-1,j}, x_{\setminus j}^{(i)}) \right]$$

Let us break this formula down, starting from the right side. The name **Accumulated Local Effects** nicely reflects all the individual components of this formula. At its core, the ALE method calculates the differences in predictions, whereby we replace the feature of interest with grid values z. The difference in prediction is the **Effect** the feature has for an individual instance in a certain interval. The sum on the right adds up the effects of all instances within an interval which appears in the formula as neighborhood $N_j(k)$. We divide this sum by the number of instances in this interval to obtain the average difference of the predictions for this interval. This average in the interval is covered by the term **Local** in the name ALE. The left sum symbol means that we accumulate the average effects across all intervals. The (uncentered) ALE of a feature value that lies, for example, in the third interval is the sum of the effects of the first, second and third intervals. The word **Accumulated** in ALE reflects this.

This effect is centered so that the mean effect is zero.

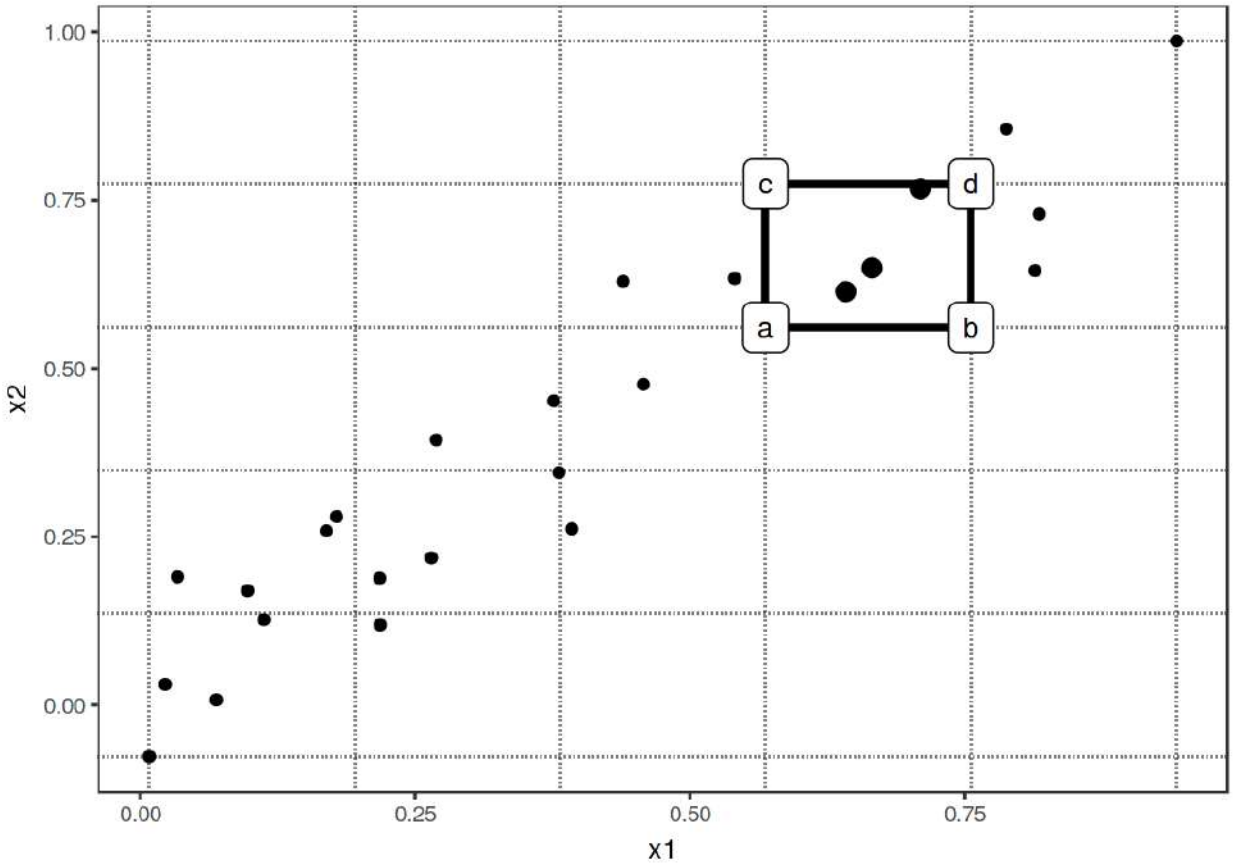
$$\hat{f}_{j,ALE}(x) = \hat{\tilde{f}}_{j,ALE}(x) - \frac{1}{n} \sum_{i=1}^{n} \hat{\tilde{f}}_{j,ALE}(x_j^{(i)})$$

The value of the ALE can be interpreted as the main effect of the feature at a certain value compared to the average prediction of the data. For example, an ALE estimate of -2 at $x_j = 3$ means that when the j-th feature has value 3, then the prediction is lower by 2 compared to the average prediction.

The quantiles of the distribution of the feature are used as the grid that defines the intervals. Using the quantiles ensures that there is the same number of data instances in each of the intervals. Quantiles have the disadvantage that the intervals can have very different lengths. This can lead to some weird ALE plots if the feature of interest is very skewed, for example many low values and only a few very high values.

ALE plots for the interaction of two features

ALE plots can also show the interaction effect of two features. The calculation principles are the same as for a single feature, but we work with rectangular cells instead of intervals, because we have to accumulate the effects in two dimensions. In addition to adjusting for the overall mean effect, we also adjust for the main effects of both features. This means that ALE for two features estimate the second-order effect, which does not include the main effects of the features. In other words, ALE for two features only shows the additional interaction effect of the two features. I spare you the formulas for 2D ALE plots because they are long and unpleasant to read. If you are interested in the calculation, I refer you to the paper, formulas (13) – (16). I will rely on visualizations to develop intuition about the second-order ALE calculation.



Calculation of 2D-ALE. We place a grid over the two features. In each grid cell we calculate the 2nd-order differences for all instance within. We first replace values of x1 and x2 with the values from the cell corners. If a, b, c and d represent the "corner"-predictions of a manipulated instance (as labeled in the graphic), then the 2nd-order difference is (d - c) - (b - a). The mean 2nd-order difference in each cell is accumulated over the grid and centered.

In the previous figure, many cells are empty due to the correlation. In the ALE plot this can be visualized with a grayed out or darkened box. Alternatively, you can replace the missing ALE estimate of an empty cell with the ALE estimate of the nearest non-empty cell.

Since the ALE estimates for two features only show the second-order effect of the features, the interpretation requires special attention. The second-order effect is the additional interaction effect of the features after we have accounted for the main effects of the features. Suppose two features do not interact, but each has a linear effect on the predicted outcome. In the 1D ALE plot for each feature, we would see a straight line as the estimated ALE curve. But when we plot the 2D ALE estimates, they should be close to zero, because the second-order effect is only the additional effect of the interaction. ALE plots and PD plots differ in this regard: PDPs always show the total effect, ALE plots show the first- or second-order effect. These are design decisions that do not depend on the underlying math. You can subtract the lower-order effects in a partial dependence plot to get the pure main or second-order effects or, you can get an estimate of the total ALE plots by refraining from subtracting the lower-order effects.

The accumulated local effects could also be calculated for arbitrarily higher orders (interactions of

three or more features), but as argued in the PDP chapter, only up to two features makes sense, because higher interactions cannot be visualized or even interpreted meaningfully.

ALE for categorical features

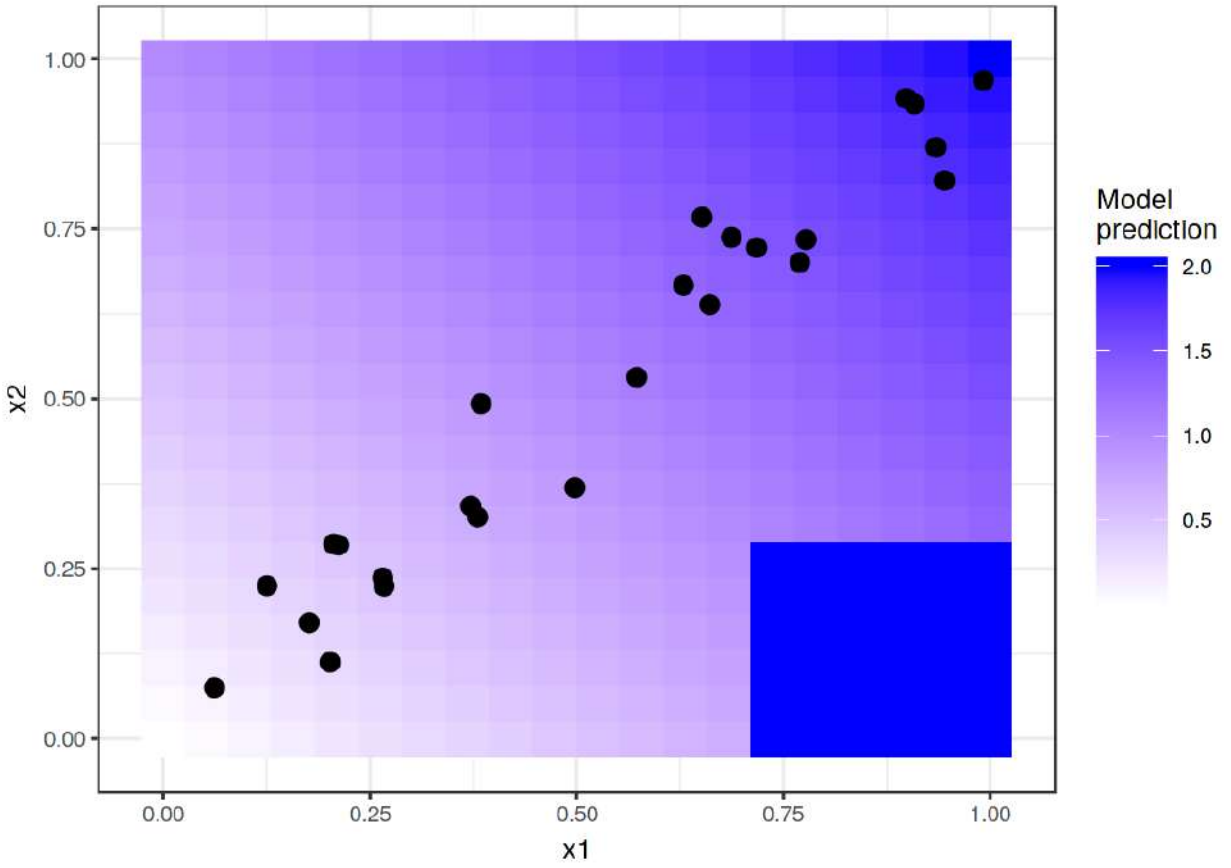
The accumulated local effects method needs – by definition – the feature values to have an order, because the method accumulates effects in a certain direction. Categorical features do not have any natural order. To compute an ALE plot for a categorical feature we have to somehow create or find an order. The order of the categories influences the calculation and interpretation of the accumulated local effects.

One solution is to order the categories according to their similarity based on the other features. The distance between two categories is the sum over the distances of each feature. The feature-wise distance compares either the cumulative distribution in both categories, also called Kolmogorov-Smirnov distance (for numerical features) or the relative frequency tables (for categorical features). Once we have the distances between all categories, we use multi-dimensional scaling to reduce the distance matrix to a one-dimensional distance measure. This gives us a similarity-based order of the categories.

To make this a little bit clearer, here is one example: Let us assume we have the two categorical features "season" and "weather" and a numerical feature "temperature". For the first categorical feature (season) we want to calculate the ALEs. The feature has the categories "spring", "summer", "fall", "winter". We start to calculate the distance between categories "spring" and "summer". The distance is the sum of distances over the features temperature and weather. For the temperature, we take all instances with season "spring", calculate the empirical cumulative distribution function and do the same for instances with season "summer" and measure their distance with the Kolmogorov-Smirnov statistic. For the weather feature we calculate for all "spring" instances the probabilities for each weather type, do the same for the "summer" instances and sum up the absolute distances in the probability distribution. If "spring" and "summer" have very different temperatures and weather, the total category-distance is large. We repeat the procedure with the other seasonal pairs and reduce the resulting distance matrix to a single dimension by multi-dimensional scaling.

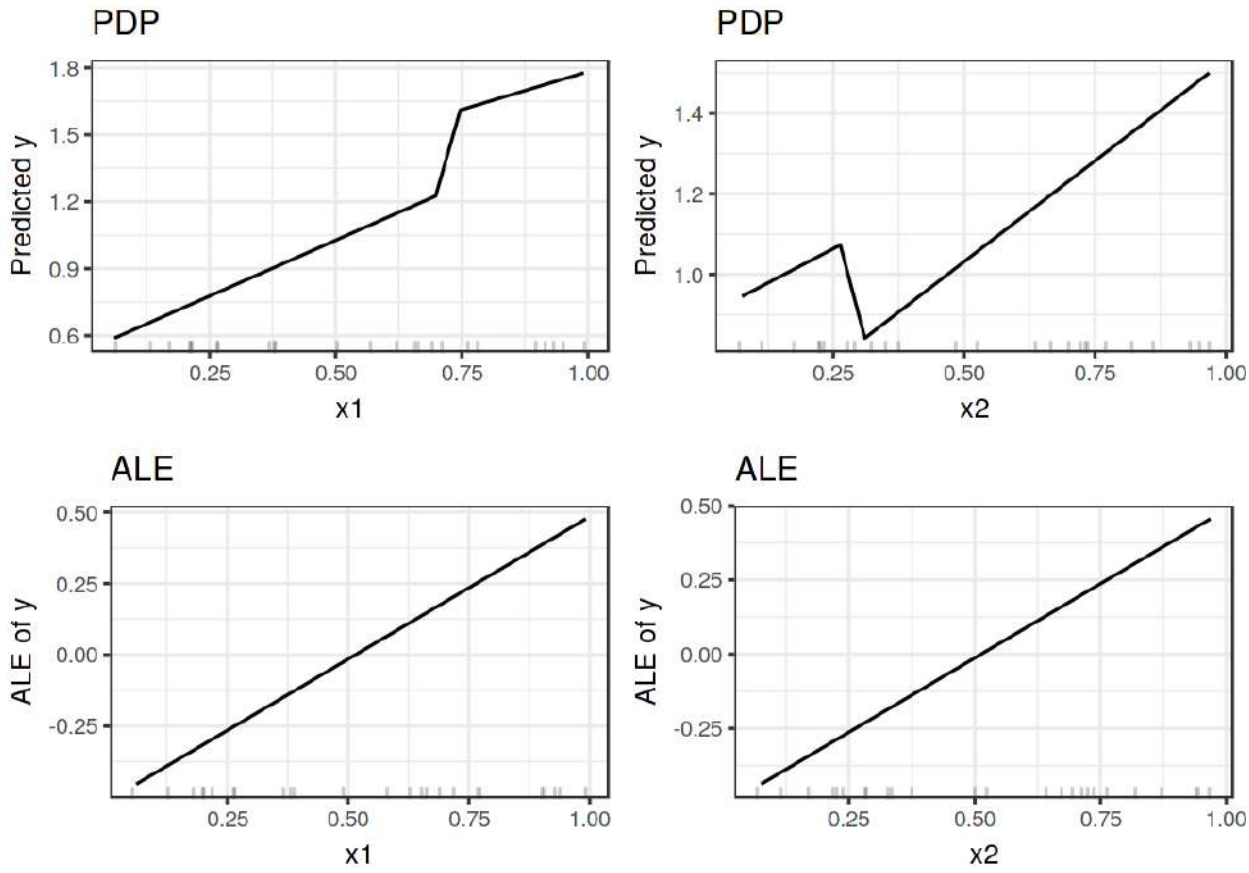
Examples

Let us see ALE plots in action. I have constructed a scenario in which partial dependence plots fail. The scenario consists of a prediction model and two strongly correlated features. The prediction model is mostly a linear regression model, but does something weird at a combination of the two features for which we have never observed instances.



Two features and the predicted outcome. The model predicts the sum of the two features (shaded background), with the exception that if x1 is greater than 0.7 and x2 less than 0.3, the model always predicts 2. This area is far from the distribution of data (point cloud) and does not affect the performance of the model and also should not affect its interpretation.

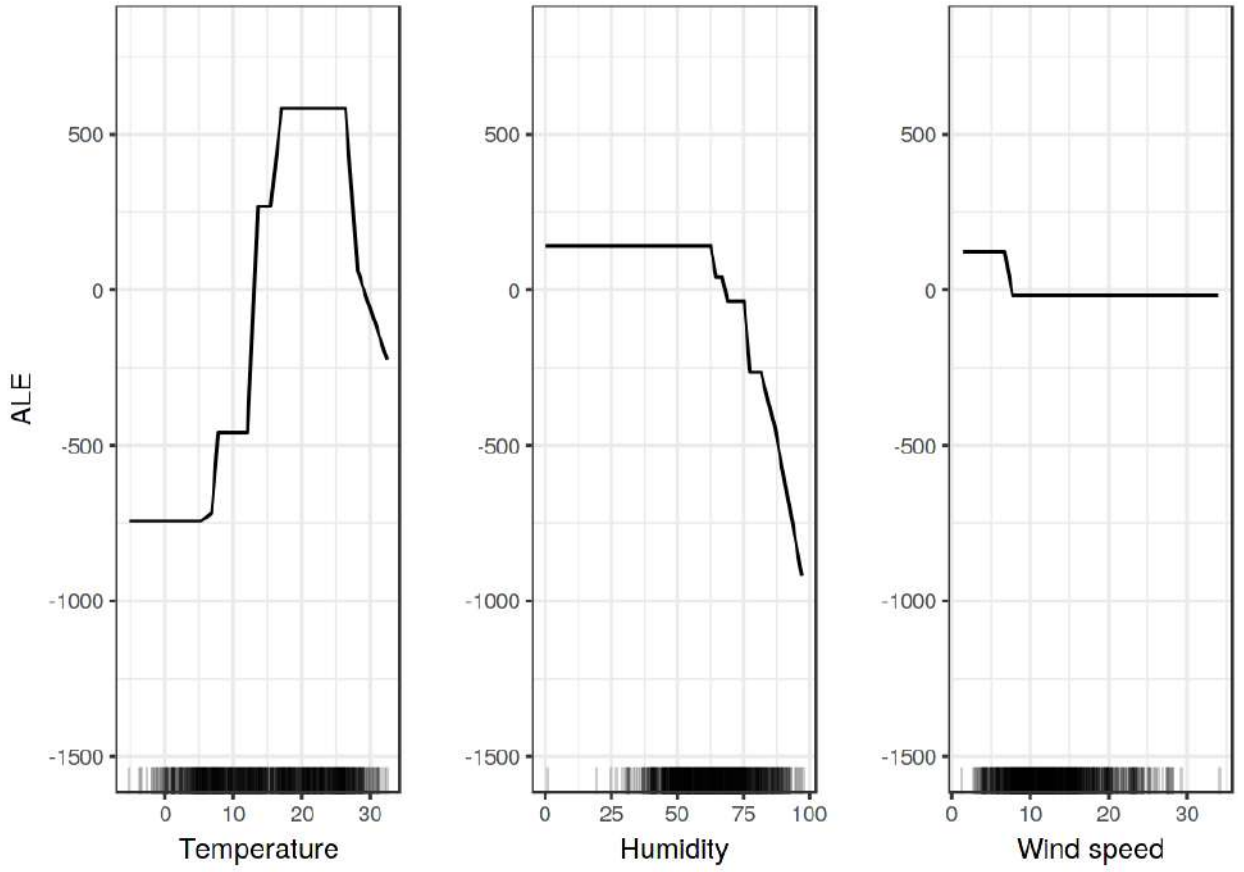
Is this a realistic, relevant scenario at all? When you train a model, the learning algorithm minimizes the loss for the existing training data instances. Weird stuff can happen outside the distribution of training data, because the model is not penalized for doing weird stuff in these areas. Leaving the data distribution is called extrapolation, which can also be used to fool machine learning models, described in the chapter on adversarial examples. See in our little example how the partial dependence plots behave compared to ALE plots.



Comparison of the feature effects computed with PDP (upper row) and ALE (lower row). The PDP estimates are influenced by the odd behavior of the model outside the data distribution (steep jumps in the plots). The ALE plots correctly identify that the machine learning model has a linear relationship between features and prediction, ignoring areas without data.

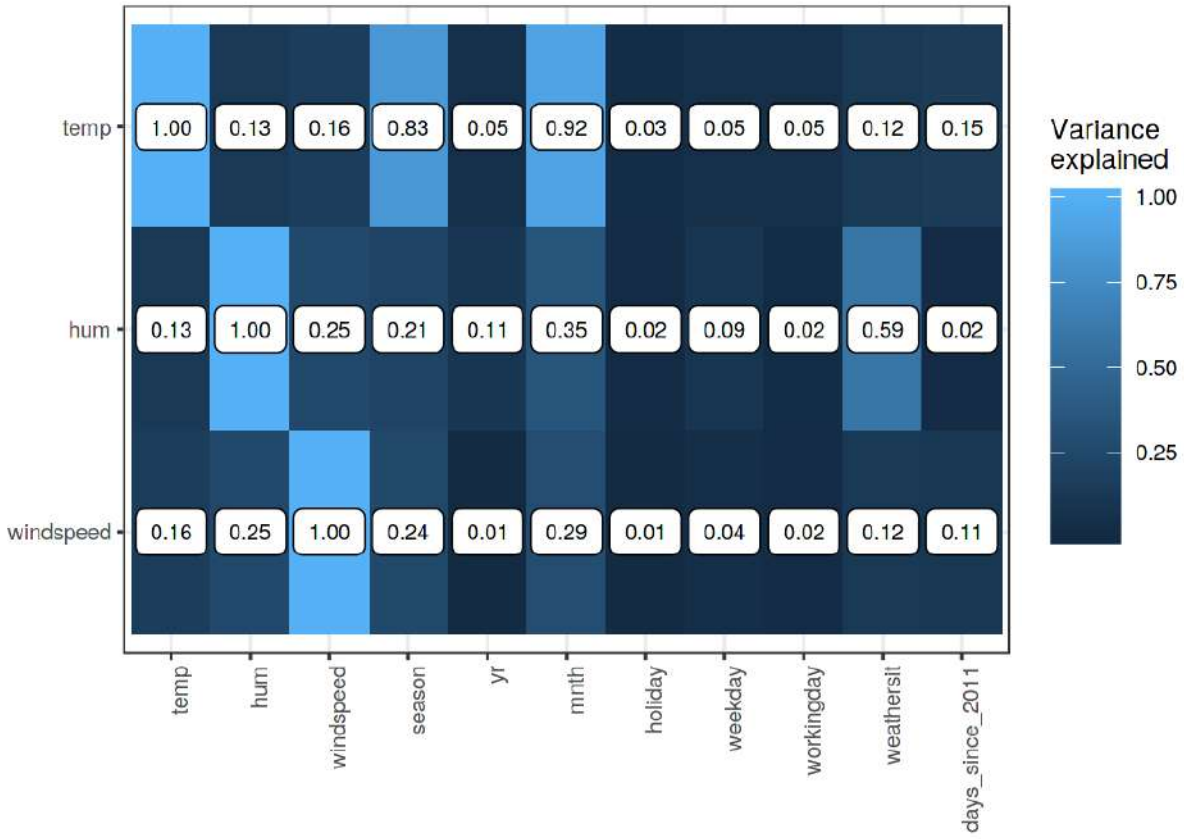
But is it not interesting to see that our model behaves oddly at x1 > 0.7 and x2 < 0.3? Well, yes and no. Since these are data instances that might be physically impossible or at least extremely unlikely, it is usually irrelevant to look into these instances. But if you suspect that your test distribution might be slightly different and some instances are actually in that range, then it would be interesting to include this area in the calculation of feature effects. But it has to be a conscious decision to include areas where we have not observed data yet and it should not be a side-effect of the method of choice like PDP. If you suspect that the model will later be used with differently distributed data, I recommend to use ALE plots and simulate the distribution of data you are expecting.

Turning to a real dataset, let us predict the number of rented bikes based on weather and day and check if the ALE plots really work as well as promised. We train a regression tree to predict the number of rented bicycles on a given day and use ALE plots to analyze how temperature, relative humidity and wind speed influence the predictions. Let us look at what the ALE plots say:



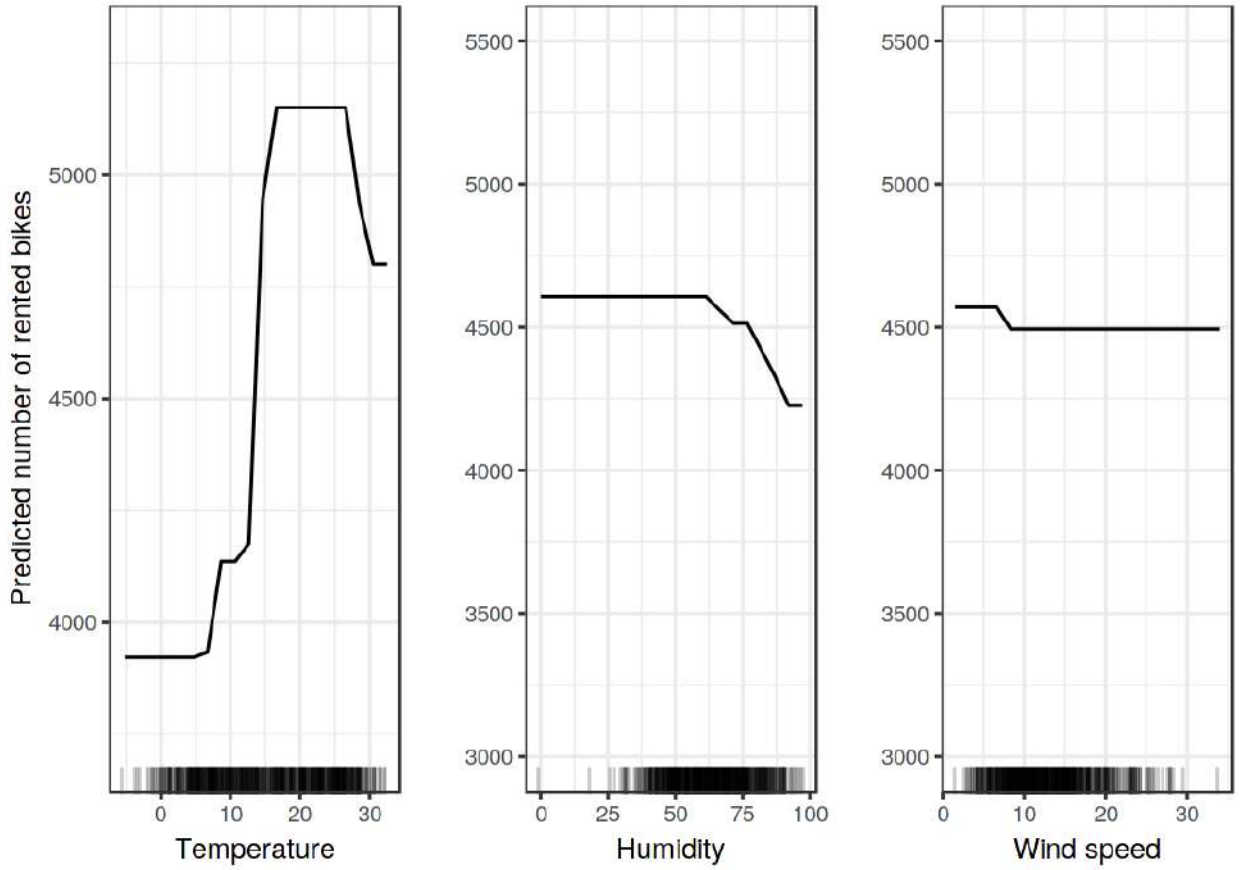
ALE plots for the bike prediction model by temperature, humidity and wind speed. The temperature has a strong effect on the prediction. The average prediction rises with increasing temperature, but falls again above 25 degrees Celsius. Humidity has a negative effect: When above 60%, the higher the relative humidity, the lower the prediction. The wind speed does not affect the predictions much.

Let us look at the correlation between temperature, humidity and wind speed and all other features. Since the data also contains categorical features, we cannot only use the Pearson correlation coefficient, which only works if both features are numerical. Instead, I train a linear model to predict, for example, temperature based on one of the other features as input. Then I measure how much variance the other feature in the linear model explains and take the square root. If the other feature was numerical, then the result is equal to the absolute value of the standard Pearson correlation coefficient. But this model-based approach of "variance-explained" (also called ANOVA, which stands for ANalysis Of VAriance) works even if the other feature is categorical. The "variance-explained" measure lies always between 0 (no association) and 1 (temperature can be perfectly predicted from the other feature). We calculate the explained variance of temperature, humidity and wind speed with all the other features. The higher the explained variance (correlation), the more (potential) problems with PD plots. The following figure visualizes how strongly the weather features are correlated with other features.



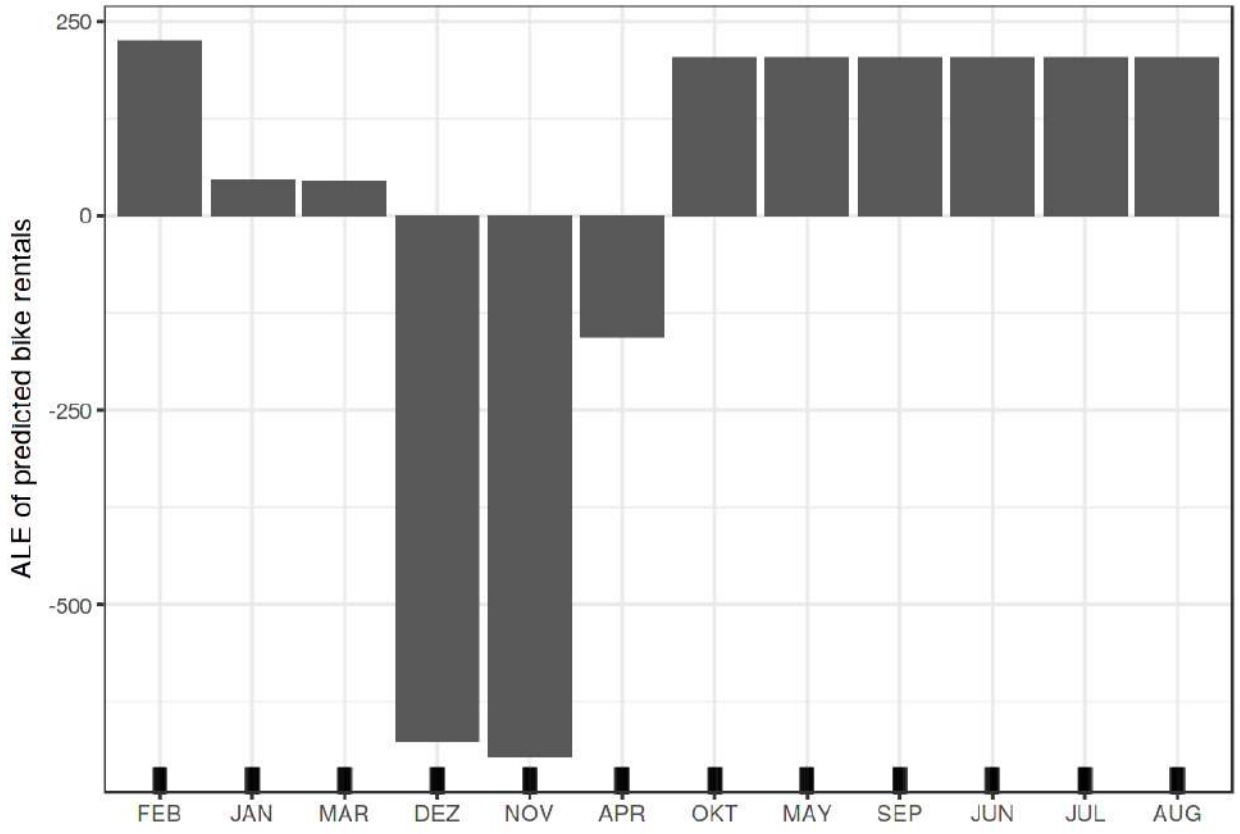
The strength of the correlation between temperature, humidity and wind speed with all features, measured as the amount of variance explained, when we train a linear model with e.g. temperature to predict and season as feature. For temperature we observe – not surprisingly – a high correlation with season and month. Humidity correlates with weather situation.

This correlation analysis reveals that we may encounter problems with partial dependence plots, especially for the temperature feature. Well, see for yourself:



PDPs for temperature, humidity and wind speed. Compared to the ALE plots, the PDPs show a smaller decrease in predicted number of bikes for high temperature or high humidity. The PDP uses all data instances to calculate the effect of high temperatures, even if they are, for example, instances with the season "winter". The ALE plots are more reliable.

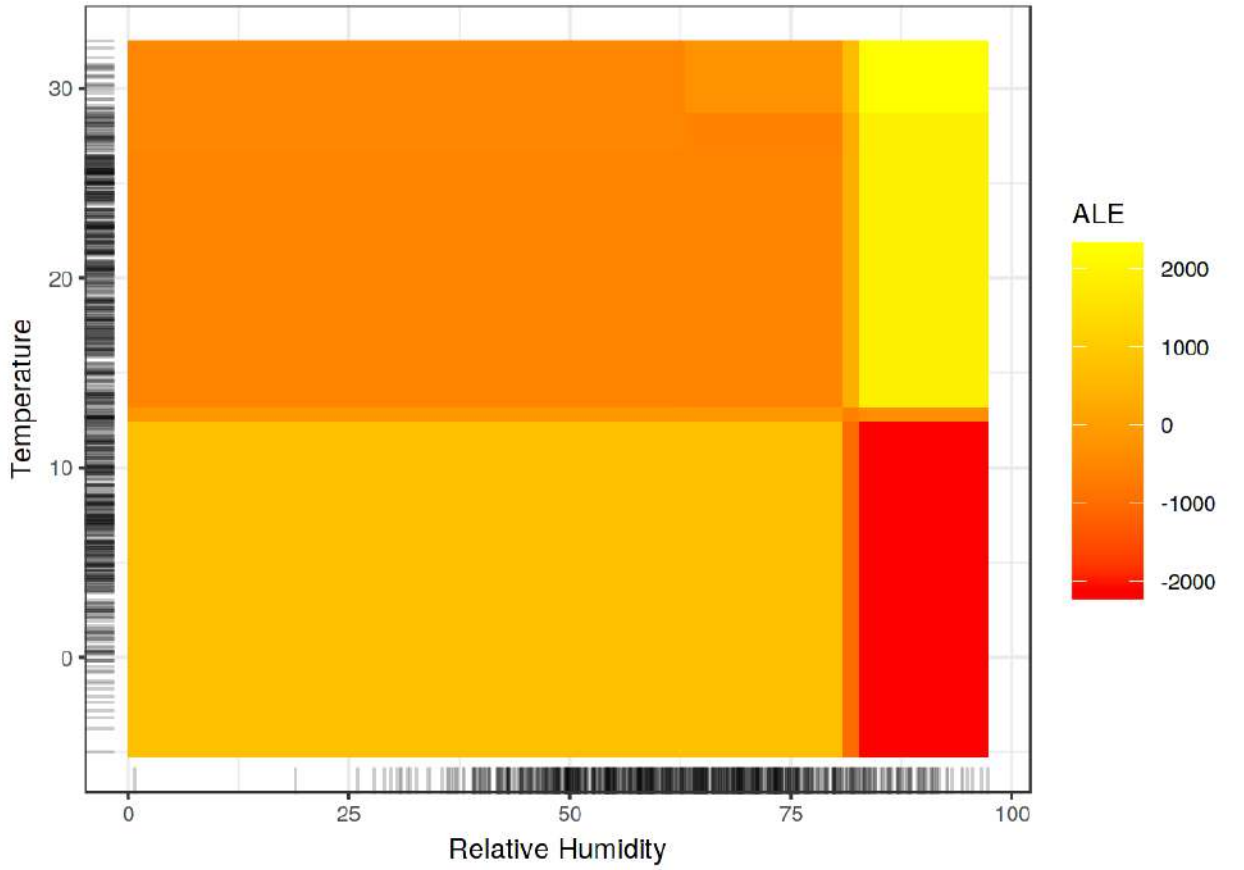
Next, let us see ALE plots in action for a categorical feature. The month is a categorical feature for which we want to analyze the effect on the predicted number of bikes. Arguably, the months already have a certain order (January to December), but let us try to see what happens if we first reorder the categories by similarity and then compute the effects. The months are ordered by the similarity of days of each month based on the other features, such as temperature or whether it is a holiday.



ALE plot for the categorical feature month. The months are ordered by their similarity to each other, based on the distributions of the other features by month. We observe that January, March and April, but especially December and November, have a lower effect on the predicted number of rented bikes compared to the other months.

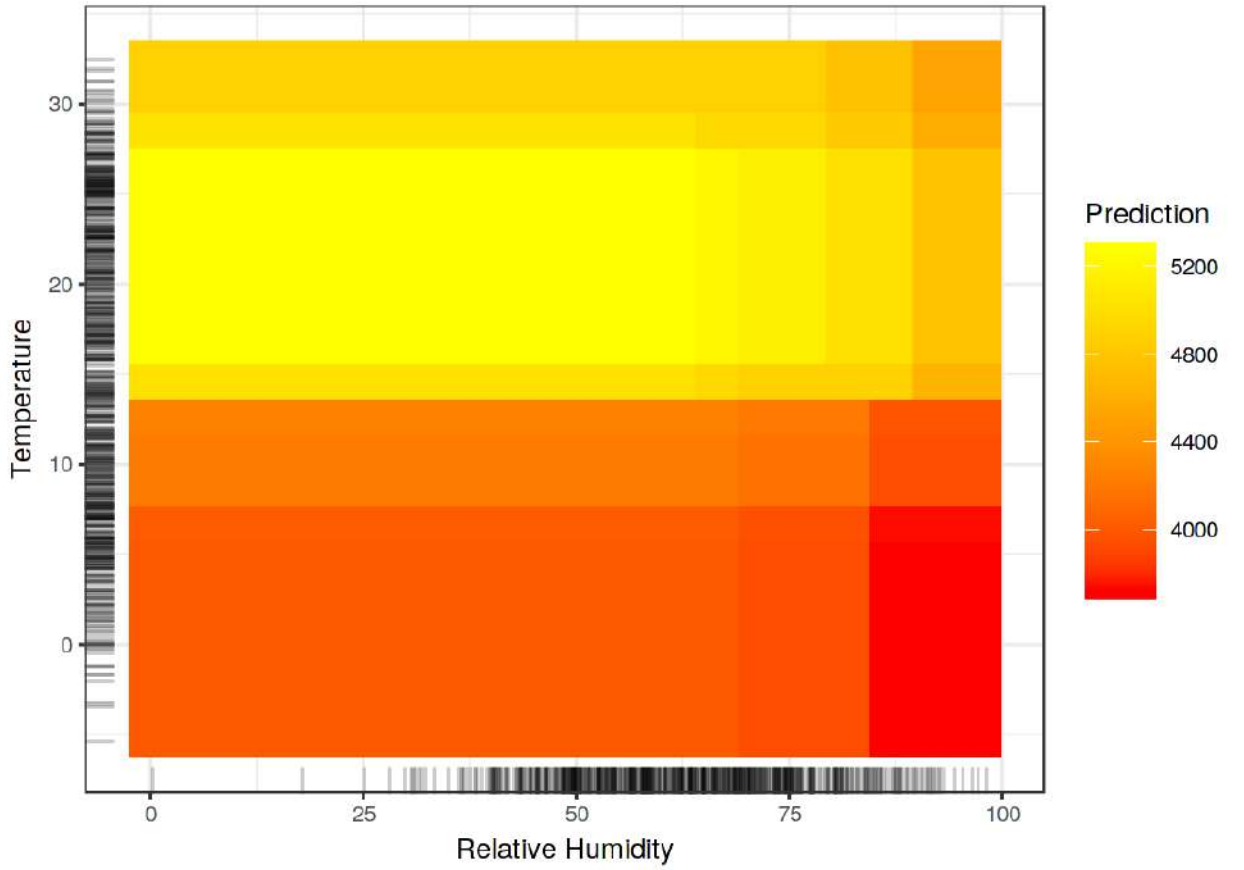
Since many of the features are related to weather, the order of the months strongly reflects how similar the weather is between the months. All colder months are on the left side (February to April) and the warmer months on the right side (October to August). Keep in mind that non-weather features have also been included in the similarity calculation, for example relative frequency of holidays has the same weight as the temperature for calculating the similarity between the months.

Next, we consider the second-order effect of humidity and temperature on the predicted number of bikes. Remember that the second-order effect is the additional interaction effect of the two features and does not include the main effects. This means that, for example, you will not see the main effect that high humidity leads to a lower number of predicted bikes on average in the second-order ALE plot.



ALE plot for the 2nd-order effect of humidity and temperature on the predicted number of rented bikes. Yellow indicates an above average and red a below average prediction when the main effects are already taken into account. The plot reveals an interaction between temperature and humidity: Hot and humid weather increases the prediction. In cold and humid weather an additional negative effect on the number of predicted bikes is shown.

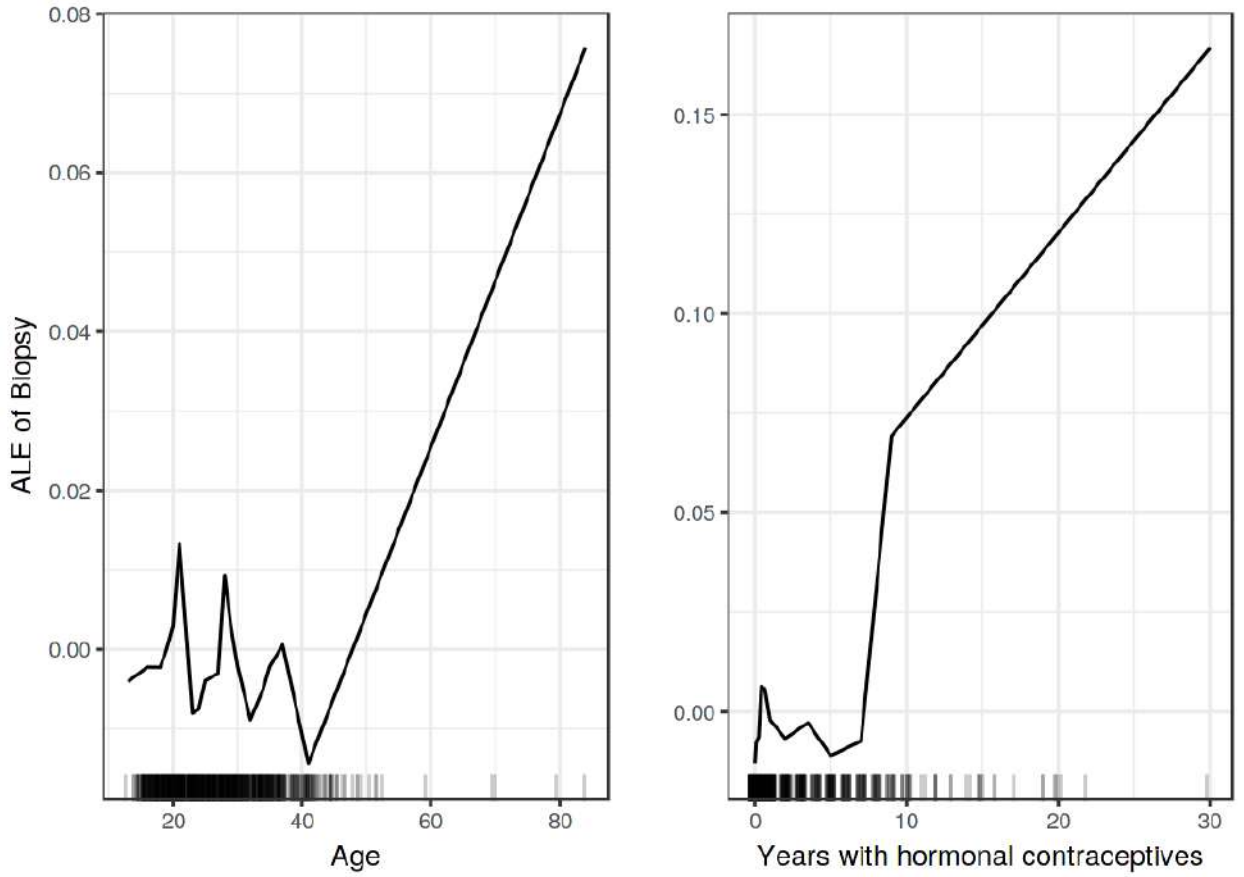
Keep in mind that both main effects of humidity and temperature say that the predicted number of bikes decreases in very hot and humid weather. In hot and humid weather, the combined effect of temperature and humidity is therefore not the sum of the main effects, but larger than the sum. To emphasize the difference between the pure second-order effect (the 2D ALE plot you just saw) and the total effect, let us look at the partial dependence plot. The PDP shows the total effect, which combines the mean prediction, the two main effects and the second-order effect (the interaction).



PDP of the total effect of temperature and humidity on the predicted number of bikes. The plot combines the main effect of each of the features and their interaction effect, as opposed to the 2D-ALE plot which only shows the interaction.

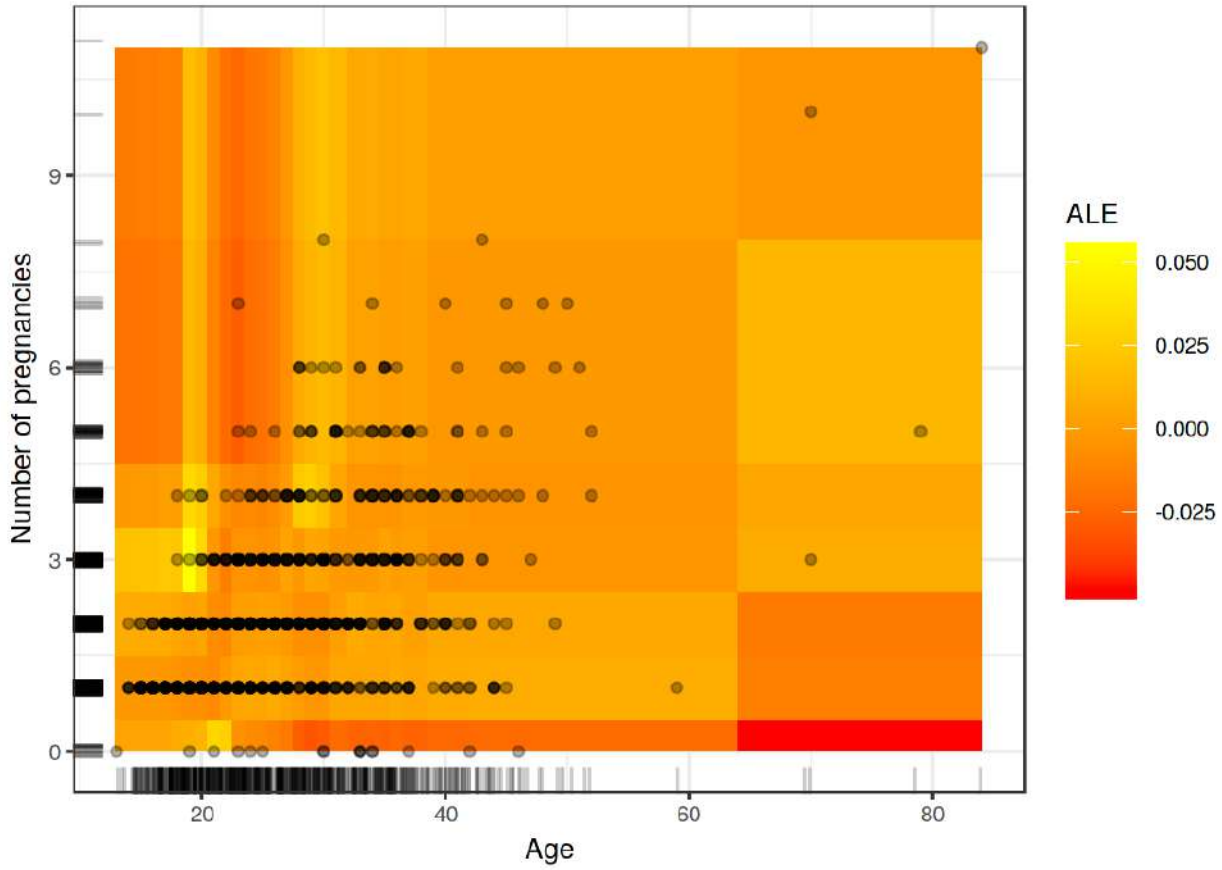
If you are only interested in the interaction, you should look at the second-order effects, because the total effect mixes the main effects into the plot. But if you want to know the combined effect of the features, you should look at the total effect (which the PDP shows). For example, if you want to know the expected number of bikes at 30 degrees Celsius and 80 percent humidity, you can read it directly from the 2D PDP. If you want to read the same from the ALE plots, you need to look at three plots: The ALE plot for temperature, for humidity and for temperature + humidity and you also need to know the overall mean prediction. In a scenario where two features have no interaction, the total effect plot of the two features could be misleading because it probably shows a complex landscape, suggesting some interaction, but it is simply the product of the two main effects. The second-order effect would immediately show that there is no interaction.

Enough bicycles for now, let's turn to a classification task. We train a random forest to predict the probability of cervical cancer based on risk factors. We visualize the accumulated local effects for two of the features:



ALE plots for the effect of age and years with hormonal contraceptives on the predicted probability of cervical cancer. For the age feature, the ALE plot shows that the predicted cancer probability is low on average up to age 40 and increases after that. The number of years with hormonal contraceptives is associated with a higher predicted cancer risk after 8 years.

Next, we look at the interaction between number of pregnancies and age.



ALE plot of the 2nd-order effect of number of pregnancies and age. The interpretation of the plot is a bit inconclusive, showing what seems like overfitting. For example, the plot shows an odd model behavior at age of 18-20 and more than 3 pregnancies (up to 5 percentage point increase in cancer probability). There are not many women in the data with this constellation of age and number of pregnancies (actual data are displayed as points), so the model is not severely penalized during the training for making mistakes for those women.

Advantages

ALE plots are unbiased, which means they still work when features are correlated. Partial dependence plots fail in this scenario because they marginalize over unlikely or even physically impossible combinations of feature values.

ALE plots are faster to compute than PDPs and scale with O(n), since the largest possible number of intervals is the number of instances with one interval per instance. The PDP requires n times the number of grid points estimations. For 20 grid points, PDPs require 20 times more predictions than the worst case ALE plot where as many intervals as instances are used.

The **interpretation of ALE plots is clear**: Conditional on a given value, the relative effect of changing the feature on the prediction can be read from the ALE plot. **ALE plots are centered at zero**. This makes their interpretation nice, because the value at each point of the ALE curve is the difference to the mean prediction. **The 2D ALE plot only shows the interaction**: If two features do not interact, the plot shows nothing.

All in all, in most situations I would **prefer ALE plots over PDPs**, because features are usually correlated to some extent.

Disadvantages

ALE plots can become a bit shaky (many small ups and downs) with a high number of intervals. In this case, reducing the number of intervals makes the estimates more stable, but also smoothes out and hides some of the true complexity of the prediction model. There is **no perfect solution for setting the number of intervals**. If the number is too small, the ALE plots might not be very accurate. If the number is too high, the curve can become shaky.

Unlike PDPs, **ALE plots are not accompanied by ICE curves**. For PDPs, ICE curves are great because they can reveal heterogeneity in the feature effect, which means that the effect of a feature looks different for subsets of the data. For ALE plots you can only check per interval whether the effect is different between the instances, but each interval has different instances so it is not the same as ICE curves.

Second-order ALE estimates have a varying stability across the feature space, which is not visualized in any way. The reason for this is that each estimation of a local effect in a cell uses a different number of data instances. As a result, all estimates have a different accuracy (but they are still the best possible estimates). The problem exists in a less severe version for main effect ALE plots. The number of instances is the same in all intervals, thanks to the use of quantiles as grid, but in some areas there will be many short intervals and the ALE curve will consist of many more estimates. But for long intervals, which can make up a big part of the entire curve, there are comparatively fewer instances. This happened in the cervical cancer prediction ALE plot for high age for example.

Second-order effect plots can be a bit annoying to interpret, as you always have to keep the main effects in mind. It is tempting to read the heat maps as the total effect of the two features, but it is only the additional effect of the interaction. The pure second-order effect is interesting for discovering and exploring interactions, but for interpreting what the effect looks like, I think it makes more sense to integrate the main effects into the plot.

The **implementation of ALE plots is much more complex** and less intuitive compared to partial dependence plots.

Even though ALE plots are not biased in case of correlated features, **interpretation remains difficult when features are strongly correlated**. Because if they have a very strong correlation, it only makes sense to analyze the effect of changing both features together and not in isolation. This disadvantage is not specific to ALE plots, but a general problem of strongly correlated features.

If the features are uncorrelated and computation time is not a problem, PDPs are slightly preferable because they are easier to understand and can be plotted along with ICE curves.

The list of disadvantages has become quite long, but do not be fooled by the number of words I use: As a rule of thumb: Use ALE instead of PDP.

Implementation and Alternatives

Did I mention that partial dependence plots and individual conditional expectation curves are an alternative? =)

To the best of my knowledge, ALE plots are currently only implemented in R, once in the ALEPlot R package[66] by the inventor himself and once in the iml package[67].

[66] https://cran.r-project.org/web/packages/ALEPlot/index.html
[67] https://cran.r-project.org/web/packages/iml/index.html

Feature Interaction

When features interact with each other in a prediction model, the prediction cannot be expressed as the sum of the feature effects, because the effect of one feature depends on the value of the other feature. Aristotle's predicate "The whole is greater than the sum of its parts" applies in the presence of interactions.

Feature Interaction?

If a machine learning model makes a prediction based on two features, we can decompose the prediction into four terms: a constant term, a term for the first feature, a term for the second feature and a term for the interaction between the two features.
The interaction between two features is the change in the prediction that occurs by varying the features after considering the individual feature effects.

For example, a model predicts the value of a house, using house size (big or small) and location (good or bad) as features, which yields four possible predictions:

Location	Size	Prediction
good	big	300,000
good	small	200,000
bad	big	250,000
bad	small	150,000

We decompose the model prediction into the following parts: A constant term (150,000), an effect for the size feature (+100,000 if big; +0 if small) and an effect for the location (+50,000 if good; +0 if bad). This decomposition fully explains the model predictions. There is no interaction effect, because the model prediction is a sum of the single feature effects for size and location. When you make a small house big, the prediction always increases by 100,000, regardless of location. Also, the difference in prediction between a good and a bad location is 50,000, regardless of size.

Let's now look at an example with interaction:

Location	Size	Prediction
good	big	400,000
good	small	200,000
bad	big	250,000
bad	small	150,000

We decompose the prediction table into the following parts: A constant term (150,000), an effect for the size feature (+100,000 if big, +0 if small) and an effect for the location (+50,000 if good, +0 if bad). For this table we need an additional term for the interaction: +100,000 if the house is big and in a good location. This is an interaction between size and location, because in this case the difference in prediction between a big and a small house depends on the location.

One way to estimate the interaction strength is to measure how much of the variation of the prediction depends on the interaction of the features. This measurement is called H-statistic, introduced by Friedman and Popescu (2008)[68].

Theory: Friedman's H-statistic

We are going to deal with two cases: First, a two-way interaction measure that tells us whether and to what extend two features in the model interact with each other; second, a total interaction measure that tells us whether and to what extend a feature interacts in the model with all the other features. In theory, arbitrary interactions between any number of features can be measured, but these two are the most interesting cases.

If two features do not interact, we can decompose the partial dependence function as follows (assuming the partial dependence functions are centered at zero):

$$PD_{jk}(x_j, x_k) = PD_j(x_j) + PD_k(x_k)$$

where $PD_{jk}(x_j, x_k)$ is the 2-way partial dependence function of both features and $PD_j(x_j)$ and $PD_k(x_k)$ the partial dependence functions of the single features.

Likewise, if a feature has no interaction with any of the other features, we can express the prediction function $\hat{f}(x)$ as a sum of partial dependence functions, where the first summand depends only on j and the second on all other features except j:

$$\hat{f}(x) = PD_j(x_j) + PD_{-j}(x_{-j})$$

where $PD_{-j}(x_{-j})$ is the partial dependence function that depends on all features except the j-th feature.

This decomposition expresses the partial dependence (or full prediction) function without interactions (between features j and k, or respectively j and all other features). In a next step, we measure the difference between the observed partial dependence function and the decomposed one without interactions. We calculate the variance of the output of the partial dependence (to measure the interaction between two features) or of the entire function (to measure the interaction between a feature and all other features). The amount of the variance explained by the interaction (difference between observed and no-interaction PD) is used as interaction strength statistic. The statistic is 0 if there is no interaction at all and 1 if all of the variance of the PD_{jk} or $\hat{f}$ is explained by the sum of the partial dependence functions. An interaction statistic of 1 between two features means that each single PD function is constant and the effect on the prediction only comes through the interaction.

Mathematically, the H-statistic proposed by Friedman and Popescu for the interaction between feature j and k is:

[68]Friedman, Jerome H, and Bogdan E Popescu. "Predictive learning via rule ensembles." The Annals of Applied Statistics. JSTOR, 916â€“54. (2008).

$$H^2_{jk} = \sum_{i=1}^{n} \left[PD_{jk}(x_j^{(i)}, x_k^{(i)}) - PD_j(x_j^{(i)}) - PD_k(x_k^{(i)}) \right]^2 / \sum_{i=1}^{n} PD^2_{jk}(x_j^{(i)}, x_k^{(i)})$$

The same applies to measuring whether a feature j interacts with any other feature:

$$H^2_j = \sum_{i=1}^{n} \left[\hat{f}(x^{(i)}) - PD_j(x_j^{(i)}) - PD_{-j}(x_{-j}^{(i)}) \right]^2 / \sum_{i=1}^{n} \hat{f}^2(x^{(i)})$$

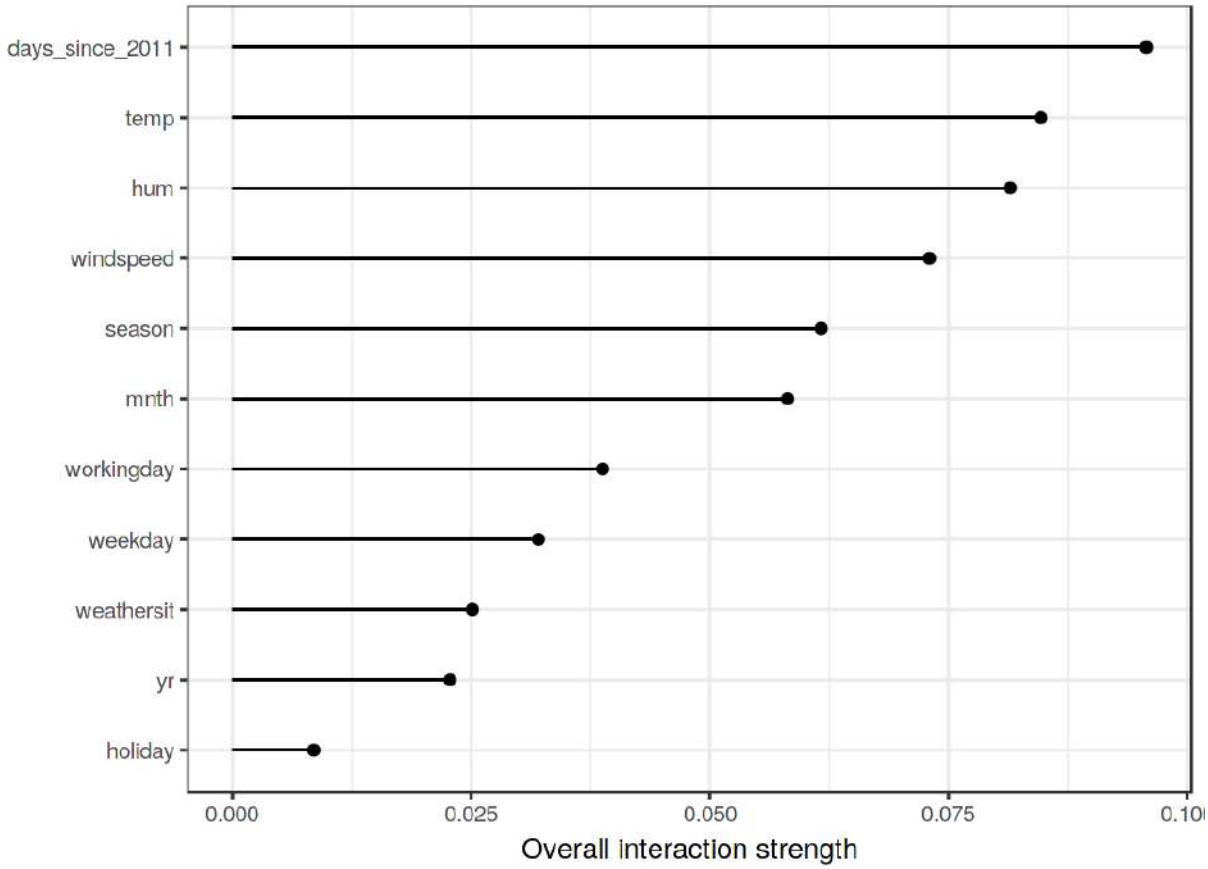
The H-statistic is expensive to evaluate, because it iterates over all data points and at each point the partial dependence has to be evaluated which in turn is done with all n data points. In the worst case, we need $2n^2$ calls to the machine learning models predict function to compute the two-way H-statistic (j vs. k) and $3n^2$ for the total H-statistic (j vs. all). To speed up the computation, we can sample from the n data points. This has the disadvantage of increasing the variance of the partial dependence estimates, which makes the H-statistic unstable. So if you are using sampling to reduce the computational burden, make sure to sample enough data points.

Friedman and Popescu also propose a test statistic to evaluate whether the H-statistic differs significantly from zero. The null hypothesis is the absence of interaction. To generate the interaction statistic under the null hypothesis, you must be able to adjust the model so that it has no interaction between feature j and k or all others. This is not possible for all types of models. Therefore this test is model-specific, not model-agnostic, and as such not covered here.

The interaction strength statistic can also be applied in a classification setting if the prediction is a probability.

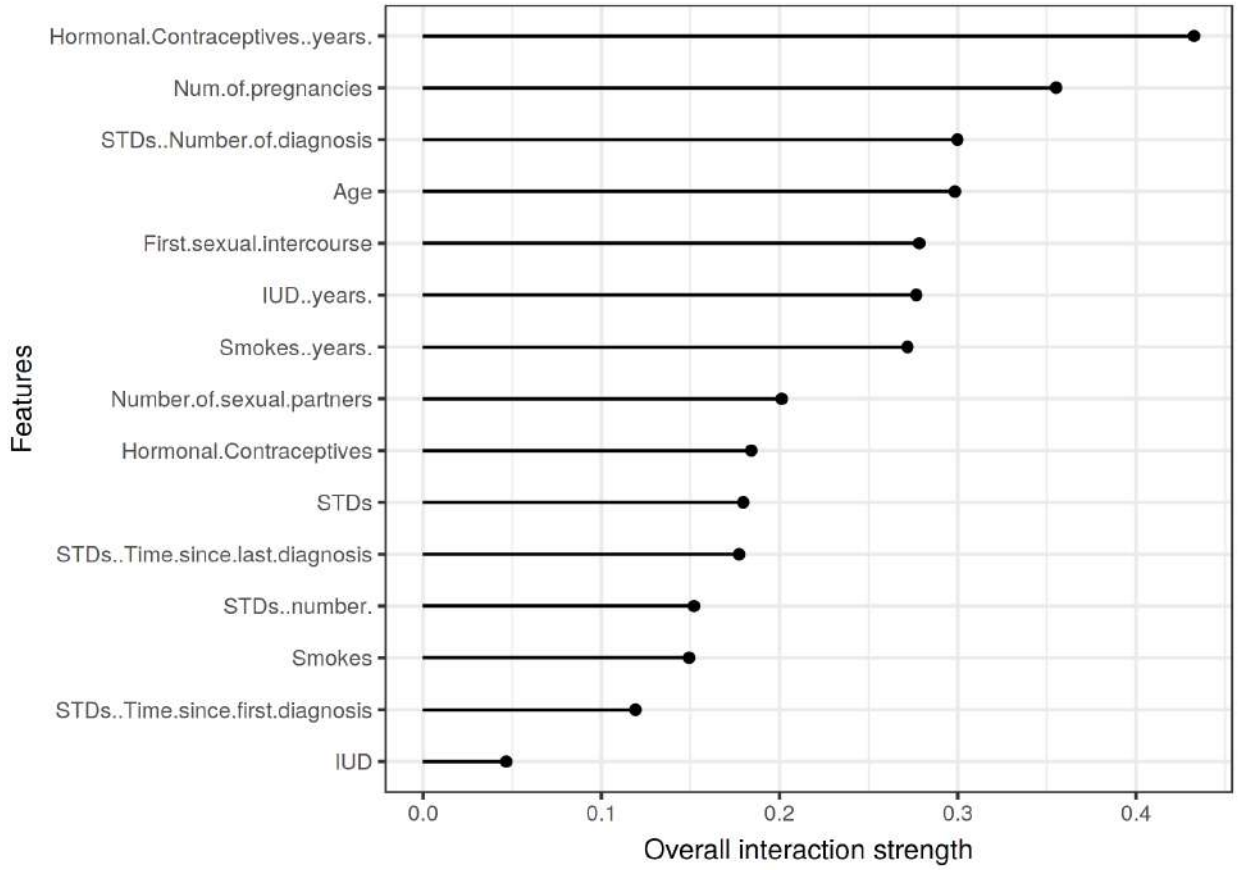
Examples

Let us see what feature interactions look like in practice! We measure the interaction strength of features in a support vector machine that predicts the number of rented bikes based on weather and calendrical features. The following plot shows the feature interaction H-statistic:



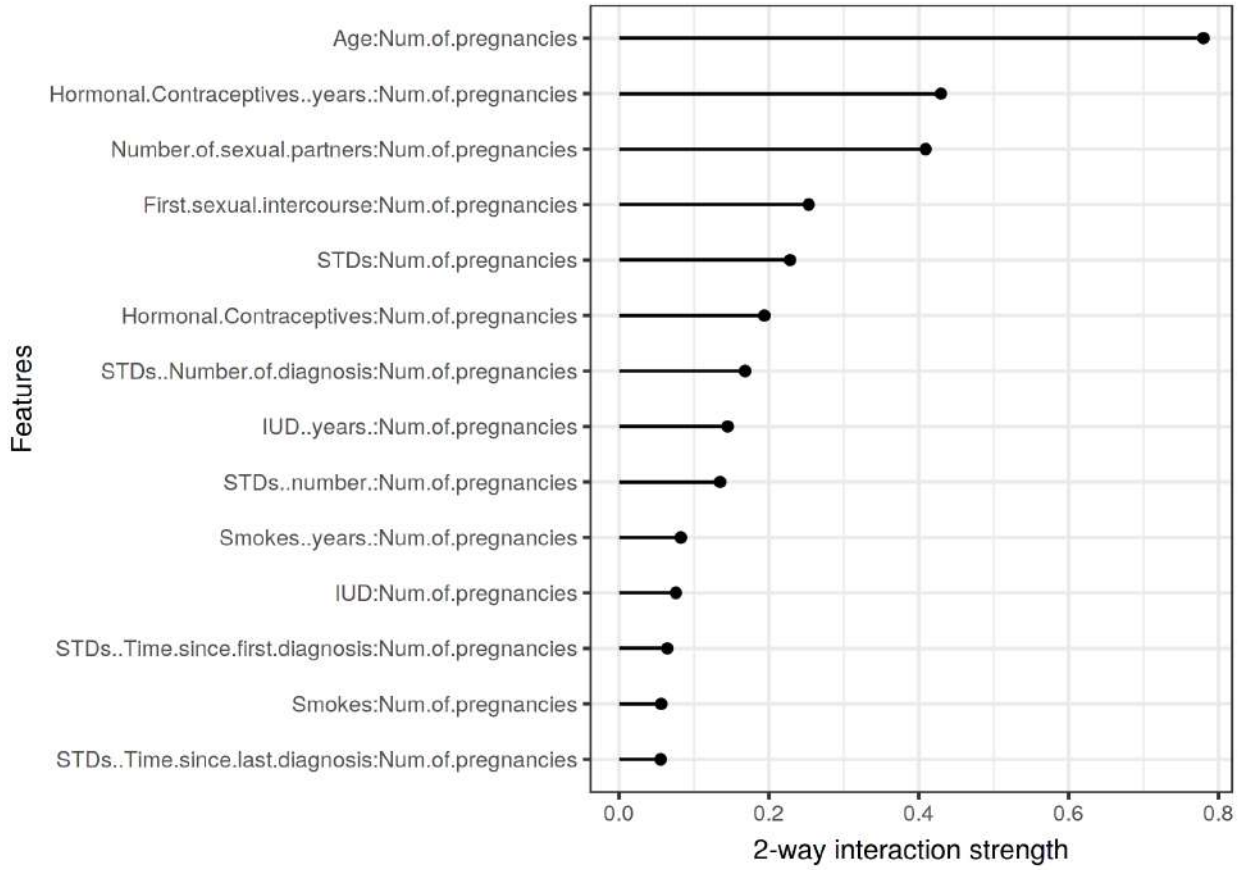
The interaction strength (H-statistic) for each feature with all other features for a support vector machine predicting bicycle rentals. Overall, the interaction effects between the features are very weak (below 10% of variance explained per feature).

In the next example, we calculate the interaction statistic for a classification problem. We analyze the interactions between features in a random forest trained to predict cervical cancer, given some risk factors.



The interaction strength (H-statistic) for each feature with all other features for a random forest predicting the probability of cervical cancer. The years on hormonal contraceptives has the highest relative interaction effect with all other features, followed by the number of pregnancies.

After looking at the feature interactions of each feature with all other features, we can select one of the features and dive deeper into all the 2-way interactions between the selected feature and the other features.



The 2-way interaction strengths (H-statistic) between number of pregnancies and each other feature. There is a strong interaction between the number of pregnancies and the age.

Advantages

The interaction H-statistic has an **underlying theory** through the partial dependence decomposition.

The H-statistic has a **meaningful interpretation**: The interaction is defined as the share of variance that is explained by the interaction.

Since the statistic is **dimensionless and always between 0 and 1**, it is comparable across features and even across models.

The statistic **detects all kinds of interactions**, regardless of their particular form.

With the H-statistic it is also possible to analyze arbitrary **higher interactions** such as the interaction strength between 3 or more features.

Disadvantages

The first thing you will notice: The interaction H-statistic takes a long time to compute, because it is **computationally expensive**.

The computation involves estimating marginal distributions. These **estimates also have a certain variance** if we do not use all data points. This means that as we sample points, the estimates also vary from run to run and the **results can be unstable**. I recommend repeating the H-statistic computation a few times to see if you have enough data to get a stable result.

It is unclear whether an interaction is significantly greater than 0. We would need to conduct a statistical test, but this **test is not (yet) available in a model-agnostic version**.

Concerning the test problem, it is difficult to say when the H-statistic is large enough for us to consider an interaction "strong".

The H-statistic tells us the strength of interactions, but it does not tell us how the interactions look like. That is what partial dependence plots are for. A meaningful workflow is to measure the interaction strengths and then create 2D-partial dependence plots for the interactions you are interested in.

The H-statistic cannot be used meaningfully if the inputs are pixels. So the technique is not useful for image classifier.

The interaction statistic works under the assumption that we can shuffle features independently. If the features correlate strongly, the assumption is violated and **we integrate over feature combinations that are very unlikely in reality**. That is the same problem that partial dependence plots have. You cannot say in general if it leads to overestimation or underestimation.

Sometimes the results are strange and for small simulations **do not yield the expected results**. But this is more of an anecdotal observation.

Implementations

For the examples in this book, I used the R package `iml`, which is available on CRAN[69] and the development version on Github[70]. There are other implementations, which focus on specific models: The R package pre[71] implements RuleFit and H-statistic. The R package gbm[72] implements gradient boosted models and H-statistic.

Alternatives

The H-statistic is not the only way to measure interactions:

[69] https://cran.r-project.org/web/packages/iml
[70] https://github.com/christophM/iml
[71] https://cran.r-project.org/web/packages/pre/index.html
[72] https://github.com/gbm-developers/gbm3

Variable Interaction Networks (VIN) by Hooker (2004)[73] is an approach that decomposes the prediction function into main effects and feature interactions. The interactions between features are then visualized as a network. Unfortunately no software is available yet.

Partial dependence based feature interaction by Greenwell et. al (2018)[74] measures the interaction between two features. This approach measures the feature importance (defined as the variance of the partial dependence function) of one feature conditional on different, fixed points of the other feature. If the variance is high, then the features interact with each other, if it is zero, they do not interact. The corresponding R package `vip` is available on Github[75]. The package also covers partial dependence plots and feature importance.

[73]Hooker, Giles. "Discovering additive structure in black box functions." Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining. (2004).

[74]Greenwell, Brandon M., Bradley C. Boehmke, and Andrew J. McCarthy. "A simple and effective model-based variable importance measure." arXiv preprint arXiv:1805.04755 (2018).

[75]https://github.com/koalaverse/vip

Feature Importance

The importance of a feature is the increase in the prediction error of the model after we permuted the feature's values, which breaks the relationship between the feature and the true outcome.

Theory

The concept is really straightforward: We measure the importance of a feature by calculating the increase in the model's prediction error after permuting the feature. A feature is "important" if shuffling its values increases the model error, because in this case the model relied on the feature for the prediction. A feature is "unimportant" if shuffling its values leaves the model error unchanged, because in this case the model ignored the feature for the prediction. The permutation feature importance measurement was introduced by Breiman (2001)[76] for random forests. Based on this idea, Fisher, Rudin, and Dominici (2018)[77] proposed a model-agnostic version of the feature importance and called it model reliance. They also introduced more advanced ideas about feature importance, for example a (model-specific) version that takes into account that many prediction models may predict the data well. Their paper is worth reading.

The permutation feature importance algorithm based on Fisher, Rudin, and Dominici (2018):

Input: Trained model f, feature matrix X, target vector y, error measure L(y,f).

1. Estimate the original model error $e^{orig} = L(y, f(X))$ (e.g. mean squared error)
2. For each feature j = 1,...,p do:
    - Generate feature matrix X^{perm} by permuting feature j in the data X. This breaks the association between feature j and true outcome y.
    - Estimate error $e^{perm} = L(Y,f(X^{perm}))$ based on the predictions of the permuted data.
    - Calculate permutation feature importance $FI^j = e^{perm}/e^{orig}$. Alternatively, the difference can be used: $FI^j = e^{perm} - e^{orig}$
3. Sort features by descending FI.

Fisher, Rudin, and Dominici (2018) suggest in their paper to split the dataset in half and swap the values of feature j of the two halves instead of permuting feature j. This is exactly the same as permuting feature j, if you think about it. If you want a more accurate estimate, you can estimate the error of permuting feature j by pairing each instance with the value of feature j of each other instance (except with itself). This gives you a dataset of size `n(n-1)` to estimate the permutation error, and it takes a large amount of computation time. I can only recommend using the `n(n-1)` -method if you are serious about getting extremely accurate estimates.

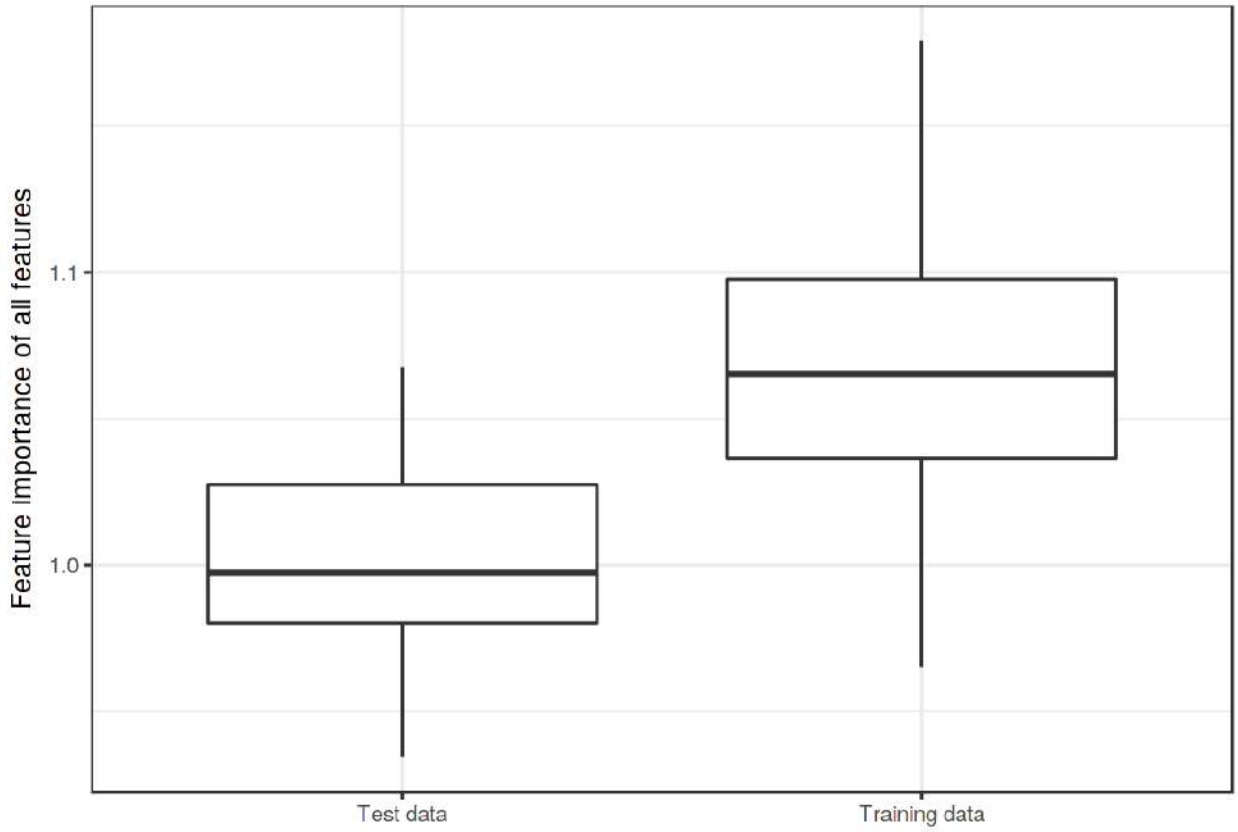
[76] Breiman, Leo."Random Forests." Machine Learning 45 (1). Springer: 5-32 (2001).

[77] Fisher, Aaron, Cynthia Rudin, and Francesca Dominici. "Model Class Reliance: Variable importance measures for any machine learning model class, from the 'Rashomon' perspective." http://arxiv.org/abs/1801.01489 (2018).

Should I Compute Importance on Training or Test Data?

tl;dr: I do not have a definite answer.

Answering the question about training or test data touches the fundamental question of what feature importance is. The best way to understand the difference between feature importance based on training vs. based on test data is an "extreme" example. I trained a support vector machine to predict a continuous, random target outcome given 50 random features (200 instances). By "random" I mean that the target outcome is independent of the 50 features. This is like predicting tomorrow's temperature given the latest lottery numbers. If the model "learns" any relationships, then it overfits. And in fact, the SVM did overfit on the training data. The mean absolute error (short: mae) for the training data is 0.29 and for the test data 0.82, which is also the error of the best possible model that always predicts the mean outcome of 0 (mae of 0.78). In other words, the SVM model is garbage. What values for the feature importance would you expect for the 50 features of this overfitted SVM? Zero because none of the features contribute to improved performance on unseen test data? Or should the importances reflect how much the model depends on each of the features, regardless whether the learned relationships generalize to unseen data? Let us take a look at how the distributions of feature importances for training and test data differ.



Distributions of feature importance values by data type. An SVM was trained on a regression dataset with 50 random features and 200 instances. The SVM overfits the data: Feature importance based on the training data shows many important features. Computed on unseen test data, the feature importances are close to a ratio of one (=unimportant).

It is unclear to me which of the two results is more desirable. So I will try to make a case for both versions and let you decide for yourself.

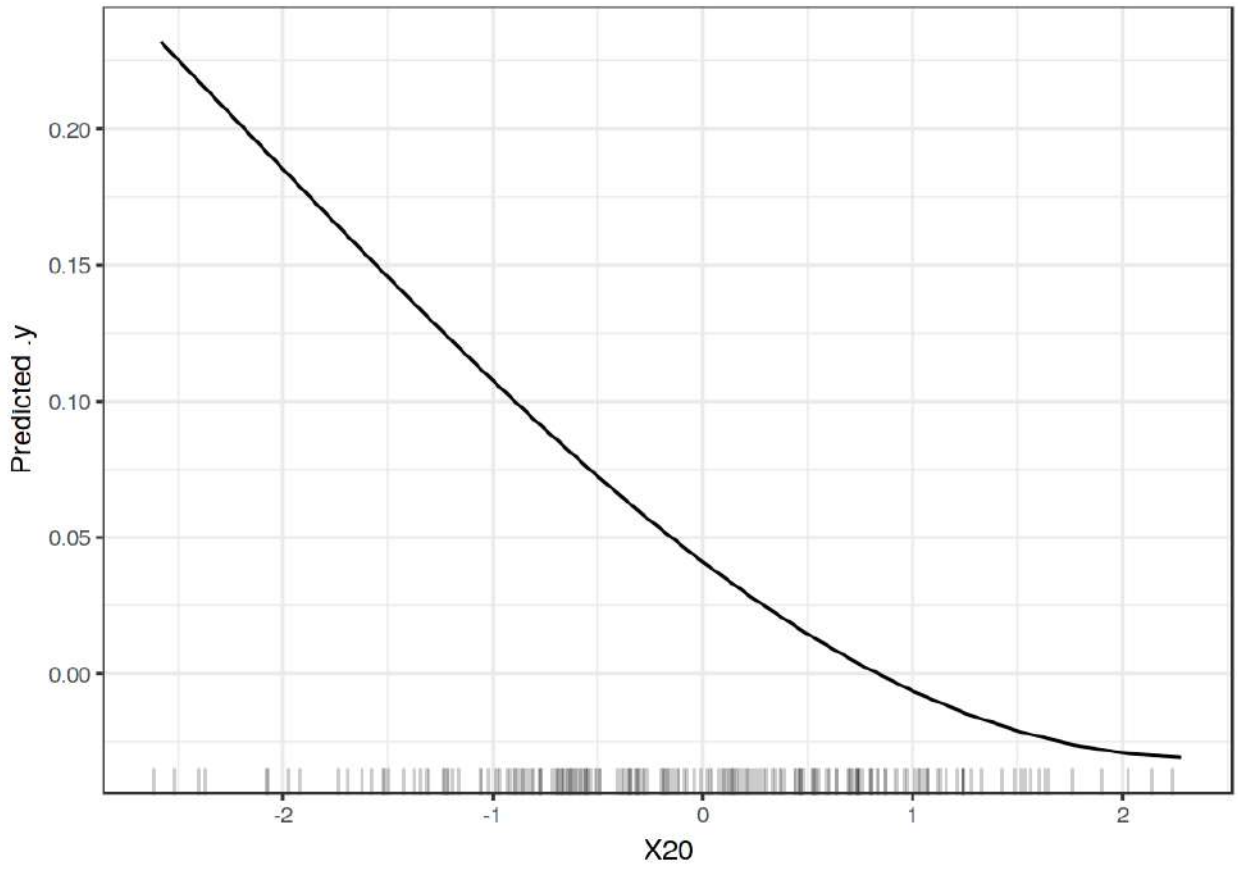
The case for test data

This is a simple case: Model error estimates based on training data are garbage -> feature importance relies on model error estimates -> feature importance based on training data is garbage.
Really, it is one of the first things you learn in machine learning: If you measure the model error (or performance) on the same data on which the model was trained, the measurement is usually too optimistic, which means that the model seems to work much better than it does in reality. And since the permutation feature importance relies on measurements of the model error, we should use unseen test data. The feature importance based on training data makes us mistakenly believe that features are important for the predictions, when in reality the model was just overfitting and the features were not important at all.

The case for training data

The arguments for using training data are somewhat more difficult to formulate, but are IMHO just as compelling as the arguments for using test data. We take another look at our garbage SVM. Based

on the training data, the most important feature was X20. Let us look at a partial dependence plot of feature X20. The partial dependence plot shows how the model output changes based on changes of the feature and does not rely on the generalization error. It does not matter whether the PDP is computed with training or test data.



PDP of feature X20, which is the most important feature according to the feature importance based on the training data. The plot shows how the SVM depends on this feature to make predictions

The plot clearly shows that the SVM has learned to rely on feature X20 for its predictions, but according to the feature importance based on the test data (1), it is not important. Based on the training data, the importance is 1.18, reflecting that the model has learned to use this feature. Feature importance based on the training data tells us which features are important for the model in the sense that it depends on them for making predictions.

As part of the case for using training data, I would like to introduce an argument against test data. In practice, you want to use all your data to train your model to get the best possible model in the end. This means no unused test data is left to compute the feature importance. You have the same problem when you want to estimate the generalization error of your model. If you would use (nested) cross-validation for the feature importance estimation, you would have the problem that the feature importance is not calculated on the final model with all the data, but on models with subsets of the data that might behave differently.

In the end, you need to decide whether you want to know how much the model relies on each feature for making predictions (-> training data) or how much the feature contributes to the performance of the model on unseen data (-> test data). To the best of my knowledge, there is no research addressing the question of training vs. test data. It will require more thorough examination than my "garbage-SVM" example. We need more research and more experience with these tools to gain a better understanding.

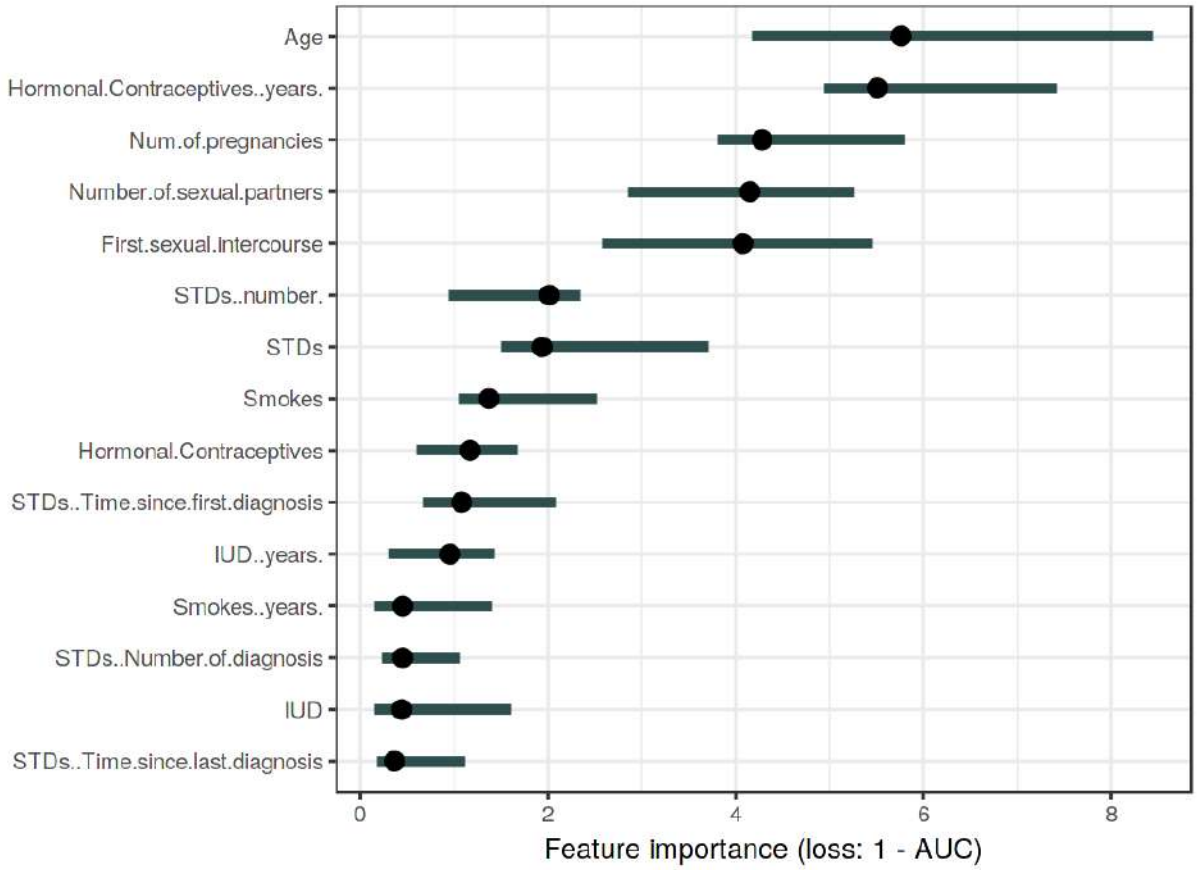
Next, we will look at some examples. I based the importance computation on the training data, because I had to choose one and using the training data needed a few lines less code.

Example and Interpretation

I show examples for classification and regression.

Cervical cancer (classification)

We fit a random forest model to predict cervical cancer. We measure the error increase by 1-AUC (1 minus the area under the ROC curve). Features associated with a model error increase by a factor of 1 (= no change) were not important for predicting cervical cancer.

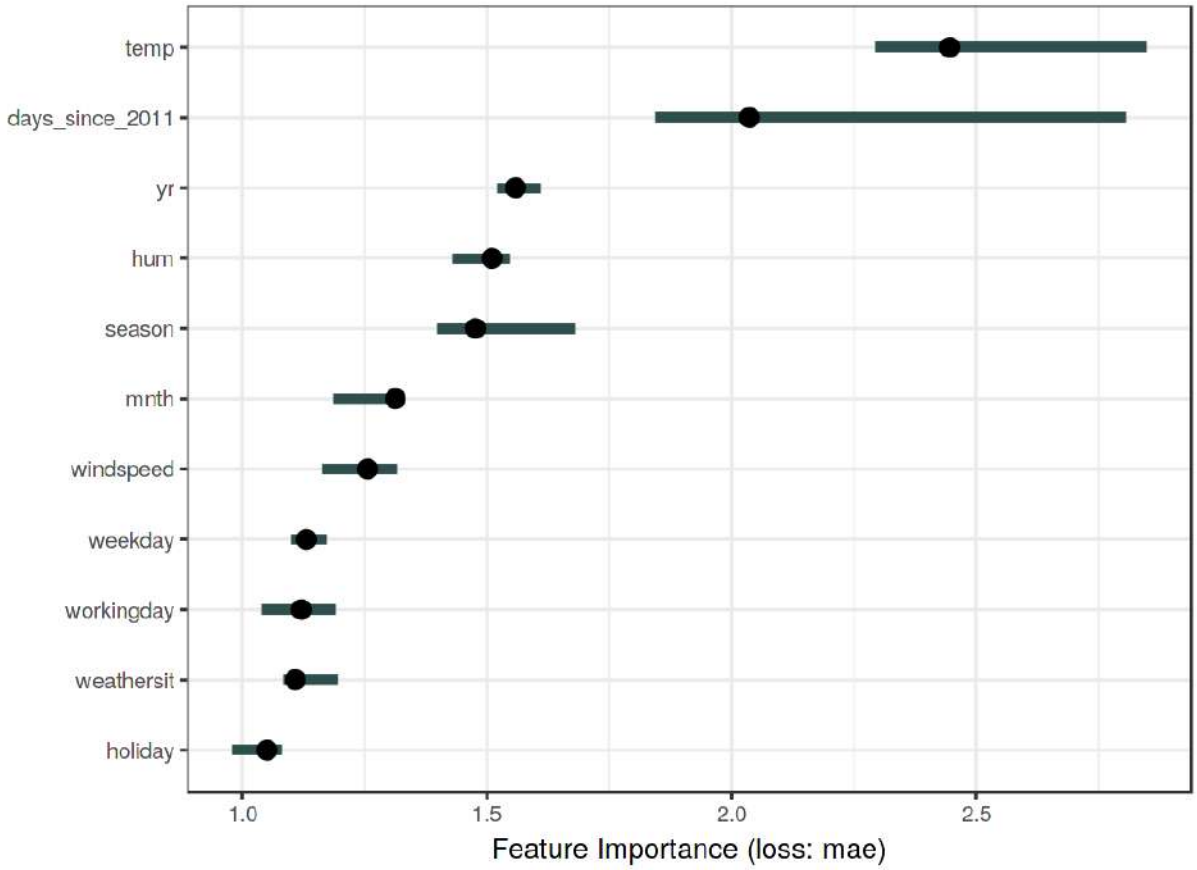


The importance of each of the features for predicting cervical cancer with a random forest. The most important feature was Age. Permuting Age resulted in an increase in 1-AUC by a factor of 5.76

The feature with the highest importance was Age associated with an error increase of 5.76 after permutation.

Bike sharing (regression)

We fit a support vector machine model to predict the number of rented bikes, given weather conditions and calendar information. As error measurement we use the mean absolute error.



The importance for each of the features in predicting bike counts with a support vector machine. The most important feature was temp, the least important was holiday.

Advantages

Nice interpretation: Feature importance is the increase in model error when the feature's information is destroyed.

Feature importance provides a **highly compressed**, **global insight** into the model's behavior.

A positive aspect of using the error ratio instead of the error difference is that the feature importance measurements are **comparable across different problems**.

The importance measure automatically **takes into account all interactions** with other features. By permuting the feature you also destroy the interaction effects with other features. This means that the permutation feature importance takes into account both the main feature effect and the interaction effects on model performance. This is also a disadvantage because the importance of the interaction between two features is included in the importance measurements of both features. This means that the feature importances do not add up to the total drop in performance, but the sum is larger. Only if there is no interaction between the features, as in a linear model, the importances add up approximately.

Permutation feature importance **does not require retraining the model**. Some other methods suggest deleting a feature, retraining the model and then comparing the model error. Since the retraining of a machine learning model can take a long time, "only" permuting a feature can save a lot of time. Importance methods that retrain the model with a subset of features appear intuitive at first glance, but the model with the reduced data is meaningless for the feature importance. We are interested in the feature importance of a fixed model. Retraining with a reduced dataset creates a different model than the one we are interested in. Suppose you train a sparse linear model (with Lasso) with a fixed number of features with a non-zero weight. The dataset has 100 features, you set the number of non-zero weights to 5. You analyze the importance of one of the features that have a non-zero weight. You remove the feature and retrain the model. The model performance remains the same because another equally good feature gets a non-zero weight and your conclusion would be that the feature was not important. Another example: The model is a decision tree and we analyze the importance of the feature that was chosen as the first split. You remove the feature and retrain the model. Since another feature is chosen as the first split, the whole tree can be very different, which means that we compare the error rates of (potentially) completely different trees to decide how important that feature is for one of the trees.

Disadvantages

It is very **unclear whether you should use training or test data** to compute the feature importance.

Permutation feature importance is **linked to the error of the model**. This is not inherently bad, but in some cases not what you need. In some cases, you might prefer to know how much the model's output varies for a feature without considering what it means for performance. For example, you want to find out how robust your model's output is when someone manipulates the features. In this case, you would not be interested in how much the model performance decreases when a feature is permuted, but how much of the model's output variance is explained by each feature. Model variance (explained by the features) and feature importance correlate strongly when the model generalizes well (i.e. it does not overfit).

You **need access to the true outcome**. If someone only provides you with the model and unlabeled data – but not the true outcome – you cannot compute the permutation feature importance.

The permutation feature importance depends on shuffling the feature, which adds randomness to the measurement. When the permutation is repeated, the **results might vary greatly**. Repeating the permutation and averaging the importance measures over repetitions stabilizes the measure, but increases the time of computation.

If features are correlated, the permutation feature importance **can be biased by unrealistic data instances**. The problem is the same as with partial dependence plots: The permutation of features produces unlikely data instances when two or more features are correlated. When they are positively correlated (like height and weight of a person) and I shuffle one of the features, I create new instances that are unlikely or even physically impossible (2 meter person weighing 30 kg for example), yet I use these new instances to measure the importance. In other words, for the permutation feature importance of a correlated feature, we consider how much the model performance decreases when

we exchange the feature with values we would never observe in reality. Check if the features are strongly correlated and be careful about the interpretation of the feature importance if they are.

Another tricky thing: **Adding a correlated feature can decrease the importance of the associated feature** by splitting the importance between both features. Let me give you an example of what I mean by "splitting" feature importance: We want to predict the probability of rain and use the temperature at 8:00 AM of the day before as a feature along with other uncorrelated features. I train a random forest and it turns out that the temperature is the most important feature and all is well and I sleep well the next night. Now imagine another scenario in which I additionally include the temperature at 9:00 AM as a feature that is strongly correlated with the temperature at 8:00 AM. The temperature at 9:00 AM does not give me much additional information if I already know the temperature at 8:00 AM. But having more features is always good, right? I train a random forest with the two temperature features and the uncorrelated features. Some of the trees in the random forest pick up the 8:00 AM temperature, others the 9:00 AM temperature, again others both and again others none. The two temperature features together have a bit more importance than the single temperature feature before, but instead of being at the top of the list of important features, each temperature is now somewhere in the middle. By introducing a correlated feature, I kicked the most important feature from the top of the importance ladder to mediocrity. On one hand this is fine, because it simply reflects the behavior of the underlying machine learning model, here the random forest. The 8:00 AM temperature has simply become less important because the model can now rely on the 9:00 AM measurement as well. On the other hand, it makes the interpretation of the feature importance considerably more difficult. Imagine you want to check the features for measurement errors. The check is expensive and you decide to check only the top 3 of the most important features. In the first case you would check the temperature, in the second case you would not include any temperature feature just because they now share the importance. Even though the importance values might make sense at the level of model behavior, it is confusing if you have correlated features.

Software and Alternatives

The `iml` R package was used for the examples. The `DALEX` R package and the Python `Skater` module also implement model-agnostic permutation feature importance.

An algorithm called PIMP[78] adapts the feature importance algorithm to provide p-values for the importances.

[78] https://academic.oup.com/bioinformatics/article/26/10/1340/193348

Global Surrogate

A global surrogate model is an interpretable model that is trained to approximate the predictions of a black box model. We can draw conclusions about the black box model by interpreting the surrogate model. Solving machine learning interpretability by using more machine learning!

Theory

Surrogate models are also used in engineering: If an outcome of interest is expensive, time-consuming or otherwise difficult to measure (e.g. because it comes from a complex computer simulation), a cheap and fast surrogate model of the outcome can be used instead. The difference between the surrogate models used in engineering and in interpretable machine learning is that the underlying model is a machine learning model (not a simulation) and that the surrogate model must be interpretable. The purpose of (interpretable) surrogate models is to approximate the predictions of the underlying model as accurately as possible and to be interpretable at the same time. The idea of surrogate models can be found under different names: Approximation model, metamodel, response surface model, emulator, ...

About the theory: There is actually not much theory needed to understand surrogate models. We want to approximate our black box prediction function f as closely as possible with the surrogate model prediction function g, under the constraint that g is interpretable. For the function g any interpretable model – for example from the interpretable models chapter – can be used.

For example a linear model:

$$g(x) = \beta_0 + \beta_1 x_1 + \ldots + \beta_p x_p$$

Or a decision tree:

$$g(x) = \sum_{m=1}^{M} c_m I\{x \in R_m\}$$

Training a surrogate model is a model-agnostic method, since it does not require any information about the inner workings of the black box model, only access to data and the prediction function is necessary. If the underlying machine learning model was replaced with another, you could still use the surrogate method. The choice of the black box model type and of the surrogate model type is decoupled.

Perform the following steps to obtain a surrogate model:

1. Select a dataset X. This can be the same dataset that was used for training the black box model or a new dataset from the same distribution. You could even select a subset of the data or a grid of points, depending on your application.

2. For the selected dataset X, get the predictions of the black box model.
3. Select an interpretable model type (linear model, decision tree, ...).
4. Train the interpretable model on the dataset X and its predictions.
5. Congratulations! You now have a surrogate model.
6. Measure how well the surrogate model replicates the predictions of the black box model.
7. Interpret the surrogate model.

You may find approaches for surrogate models that have some extra steps or differ a little, but the general idea is usually as described here.

One way to measure how well the surrogate replicates the black box model is the R-squared measure:

$$R^2 = 1 - \frac{SSE}{SST} = 1 - \frac{\sum_{i=1}^{n}(\hat{y}_*^{(i)} - \hat{y}^{(i)})^2}{\sum_{i=1}^{n}(\hat{y}^{(i)} - \bar{\hat{y}})^2}$$

where $\hat{y}_*^{(i)}$ is the prediction for the i-th instance of the surrogate model, $\hat{y}^{(i)}$ the prediction of the black box model and $\bar{\hat{y}}$ the mean of the black box model predictions. SSE stands for sum of squares error and SST for sum of squares total. The R-squared measure can be interpreted as the percentage of variance that is captured by the surrogate model. If R-squared is close to 1 (= low SSE), then the interpretable model approximates the behavior of the black box model very well. If the interpretable model is very close, you might want to replace the complex model with the interpretable model. If the R-squared is close to 0 (= high SSE), then the interpretable model fails to explain the black box model.

Note that we have not talked about the model performance of the underlying black box model, i.e. how good or bad it performs in predicting the actual outcome. The performance of the black box model does not play a role in training the surrogate model. The interpretation of the surrogate model is still valid because it makes statements about the model and not about the real world. But of course the interpretation of the surrogate model becomes irrelevant if the black box model is bad, because then the black box model itself is irrelevant.

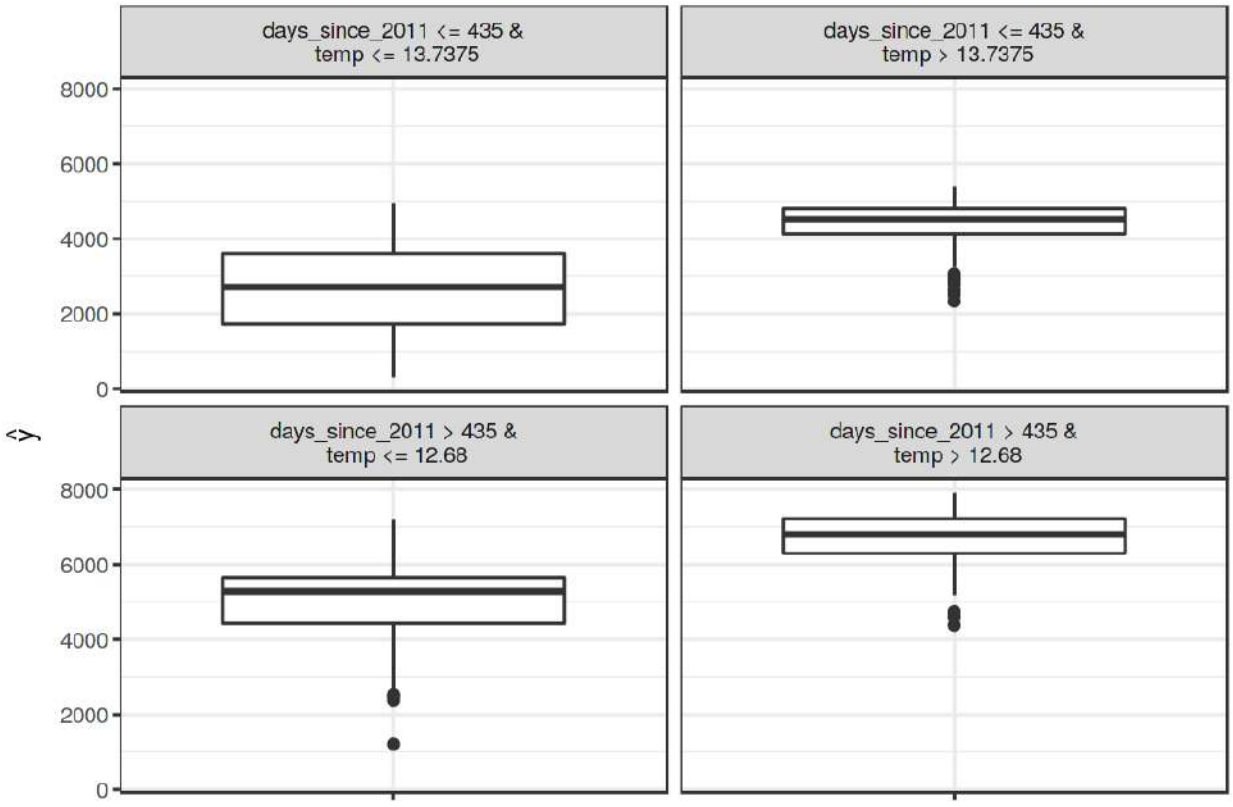
We could also build a surrogate model based on a subset of the original data or reweight the instances. In this way, we change the distribution of the surrogate model's input, which changes the focus of the interpretation (then it is no longer really global). If we weight the data locally by a specific instance of the data (the closer the instances to the selected instance, the higher their weight), we get a local surrogate model that can explain the individual prediction of the instance. Read more about local models in the following chapter.

Example

To demonstrate the surrogate models, we consider a regression and a classification example.

First, we train a support vector machine to predict the daily number of rented bikes given weather and calendar information. The support vector machine is not very interpretable, so we train a

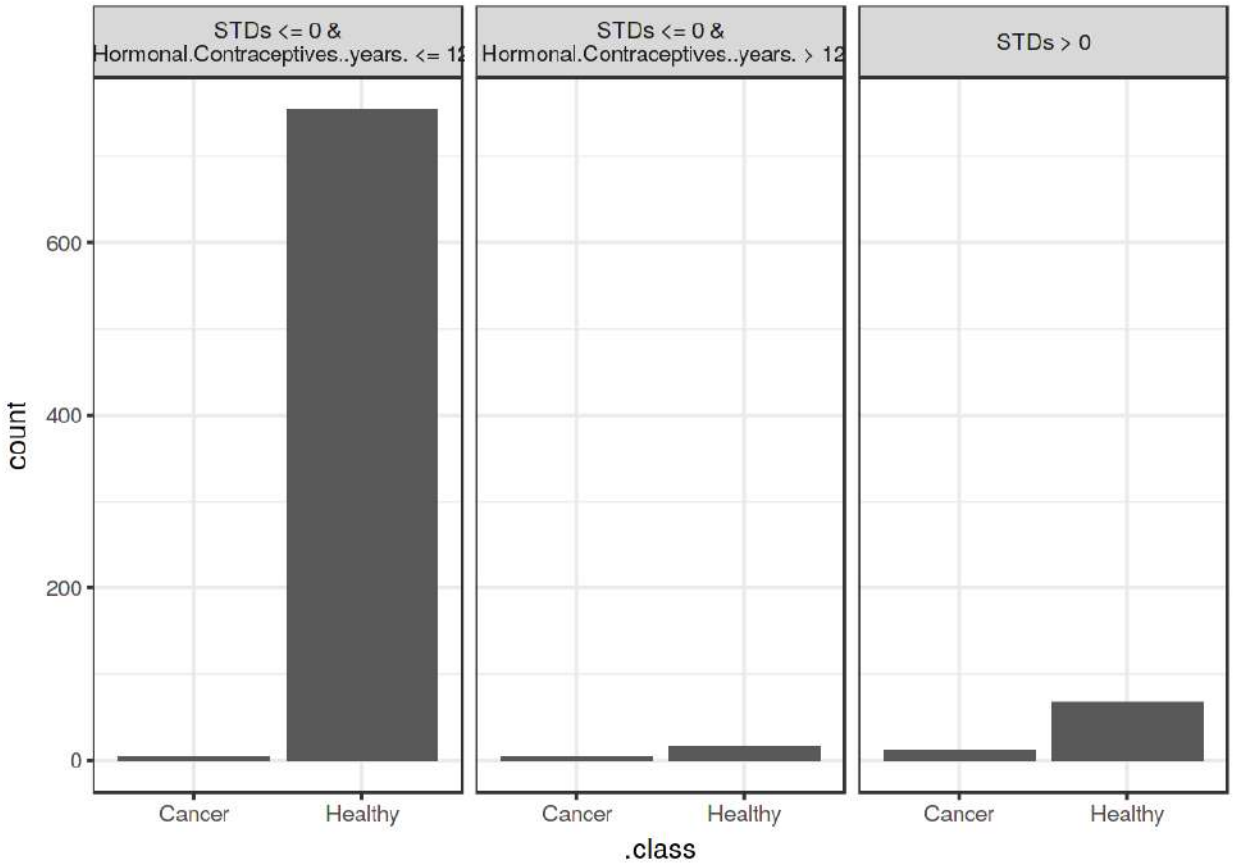
surrogate with a CART decision tree as interpretable model to approximate the behavior of the support vector machine.



The terminal nodes of a surrogate tree that approximates the predictions of a support vector machine trained on the bike rental dataset. The distributions in the nodes show that the surrogate tree predicts a higher number of rented bikes when temperature is above 13 degrees Celsius and when the day was later in the 2 year period (cut point at 435 days).

The surrogate model has a R-squared (variance explained) of 0.77 which means it approximates the underlying black box behavior quite well, but not perfectly. If the fit were perfect, we could throw away the support vector machine and use the tree instead.

In our second example, we predict the probability of cervical cancer with a random forest. Again we train a decision tree with the original dataset, but with the prediction of the random forest as outcome, instead of the real classes (healthy vs. cancer) from the data.



The terminal nodes of a surrogate tree that approximates the predictions of a random forest trained on the cervical cancer dataset. The counts in the nodes show the frequency of the black box models classifications in the nodes.

The surrogate model has an R-squared (variance explained) of 0.2, which means it does not approximate the random forest well and we should not overinterpret the tree when drawing conclusions about the complex model.

Advantages

The surrogate model method is **flexible**: Any model from the interpretable models chapter can be used. This also means that you can exchange not only the interpretable model, but also the underlying black box model. Suppose you create some complex model and explain it to different teams in your company. One team is familiar with linear models, the other team can understand decision trees. You can train two surrogate models (linear model and decision tree) for the original black box model and offer two kinds of explanations. If you find a better performing black box model, you do not have to change your method of interpretation, because you can use the same class of surrogate models.

I would argue that the approach is very **intuitive** and straightforward. This means it is easy to implement, but also easy to explain to people not familiar with data science or machine learning.

With the **R-squared measure**, we can easily measure how good our surrogate models are in approximating the black box predictions.

Disadvantages

You have to be aware that you draw **conclusions about the model and not about the data**, since the surrogate model never sees the real outcome.

It is not clear what the best **cut-off for R-squared** is in order to be confident that the surrogate model is close enough to the black box model. 80% of variance explained? 50%? 99%?

We can measure how close the surrogate model is to the black box model. Let us assume we are not very close, but close enough. It could happen that the interpretable model is very **close for one subset of the dataset**, **but widely divergent for another subset**. In this case the interpretation for the simple model would not be equally good for all data points.

The interpretable model you choose as a surrogate **comes with all its advantages and disadvantages**.

Some people argue that there are, in general, **no intrinsically interpretable models** (including even linear models and decision trees) and that it would even be dangerous to have an illusion of interpretability. If you share this opinion, then of course this method is not for you.

Software

I used the `iml` R package for the examples. If you can train a machine learning model, then you should be able to implement surrogate models yourself. Simply train an interpretable model to predict the predictions of the black box model.

Local Surrogate (LIME)

Local surrogate models are interpretable models that are used to explain individual predictions of black box machine learning models. Local interpretable model-agnostic explanations (LIME)[79] is a paper in which the authors propose a concrete implementation of local surrogate models. Surrogate models are trained to approximate the predictions of the underlying black box model. Instead of training a global surrogate model, LIME focuses on training local surrogate models to explain individual predictions.

The idea is quite intuitive. First, forget about the training data and imagine you only have the black box model where you can input data points and get the predictions of the model. You can probe the box as often as you want. Your goal is to understand why the machine learning model made a certain prediction. LIME tests what happens to the predictions when you give variations of your data into the machine learning model. LIME generates a new dataset consisting of permuted samples and the corresponding predictions of the black box model. On this new dataset LIME then trains an interpretable model, which is weighted by the proximity of the sampled instances to the instance of interest. The interpretable model can be anything from the interpretable models chapter, for example Lasso or a decision tree. The learned model should be a good approximation of the machine learning model predictions locally, but it does not have to be a good global approximation. This kind of accuracy is also called local fidelity.

Mathematically, local surrogate models with interpretability constraint can be expressed as follows:

$$\text{explanation}(x) = \arg\min_{g \in G} L(f, g, \pi_x) + \Omega(g)$$

The explanation model for instance x is the model g (e.g. linear regression model) that minimizes loss L (e.g. mean squared error), which measures how close the explanation is to the prediction of the original model f (e.g. an xgboost model), while the model complexity $\Omega(g)$ is kept low (e.g. prefer fewer features). G is the family of possible explanations, for example all possible linear regression models. The proximity measure π_x defines how large the neighborhood around instance x is that we consider for the explanation. In practice, LIME only optimizes the loss part. The user has to determine the complexity, e.g. by selecting the maximum number of features that the linear regression model may use.

The recipe for training local surrogate models:

- Select your instance of interest for which you want to have an explanation of its black box prediction.
- Perturb your dataset and get the black box predictions for these new points.
- Weight the new samples according to their proximity to the instance of interest.
- Train a weighted, interpretable model on the dataset with the variations.

[79] Ribeiro, Marco Tulio, Sameer Singh, and Carlos Guestrin. "Why should I trust you?: Explaining the predictions of any classifier." Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining. ACM (2016).

- Explain the prediction by interpreting the local model.

In the current implementations in R[80] and Python[81], for example, linear regression can be chosen as interpretable surrogate model. In advance, you have to select K, the number of features you want to have in your interpretable model. The lower K, the easier it is to interpret the model. A higher K potentially produces models with higher fidelity. There are several methods for training models with exactly K features. A good choice is Lasso. A Lasso model with a high regularization parameter λ yields a model without any feature. By retraining the Lasso models with slowly decreasing λ, one after the other, the features get weight estimates that differ from zero. If there are K features in the model, you have reached the desired number of features. Other strategies are forward or backward selection of features. This means you either start with the full model (= containing all features) or with a model with only the intercept and then test which feature would bring the biggest improvement when added or removed, until a model with K features is reached.

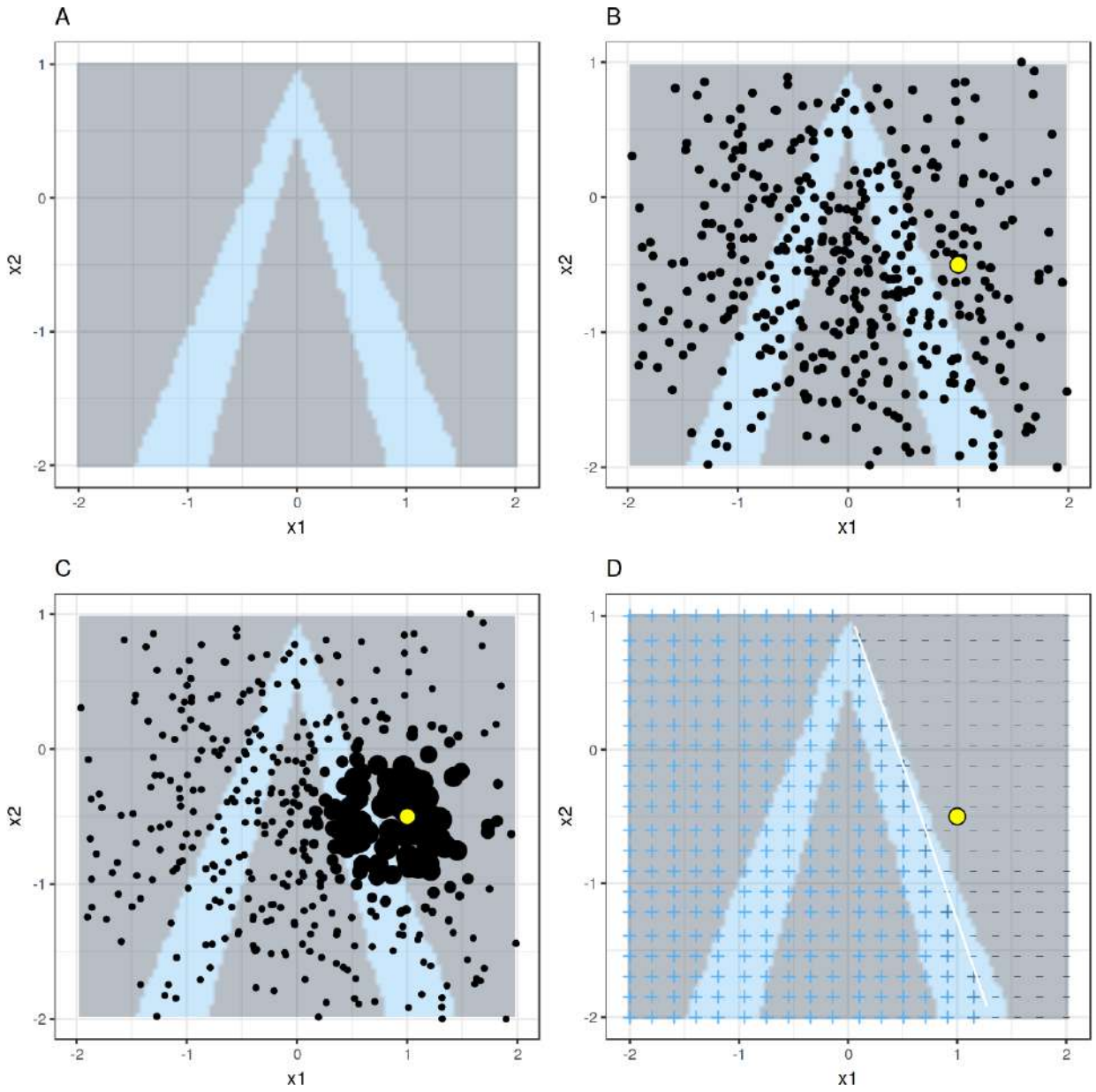
How do you get the variations of the data? This depends on the type of data, which can be either text, image or tabular data. For text and images, the solution is to turn single words or super-pixels on or off. In the case of tabular data, LIME creates new samples by perturbing each feature individually, drawing from a normal distribution with mean and standard deviation taken from the feature.

LIME for Tabular Data

Tabular data is data that comes in tables, with each row representing an instance and each column a feature. LIME samples are not taken around the instance of interest, but from the training data's mass center, which is problematic. But it increases the probability that the result for some of the sample points predictions differ from the data point of interest and that LIME can learn at least some explanation.

It is best to visually explain how sampling and local model training works:

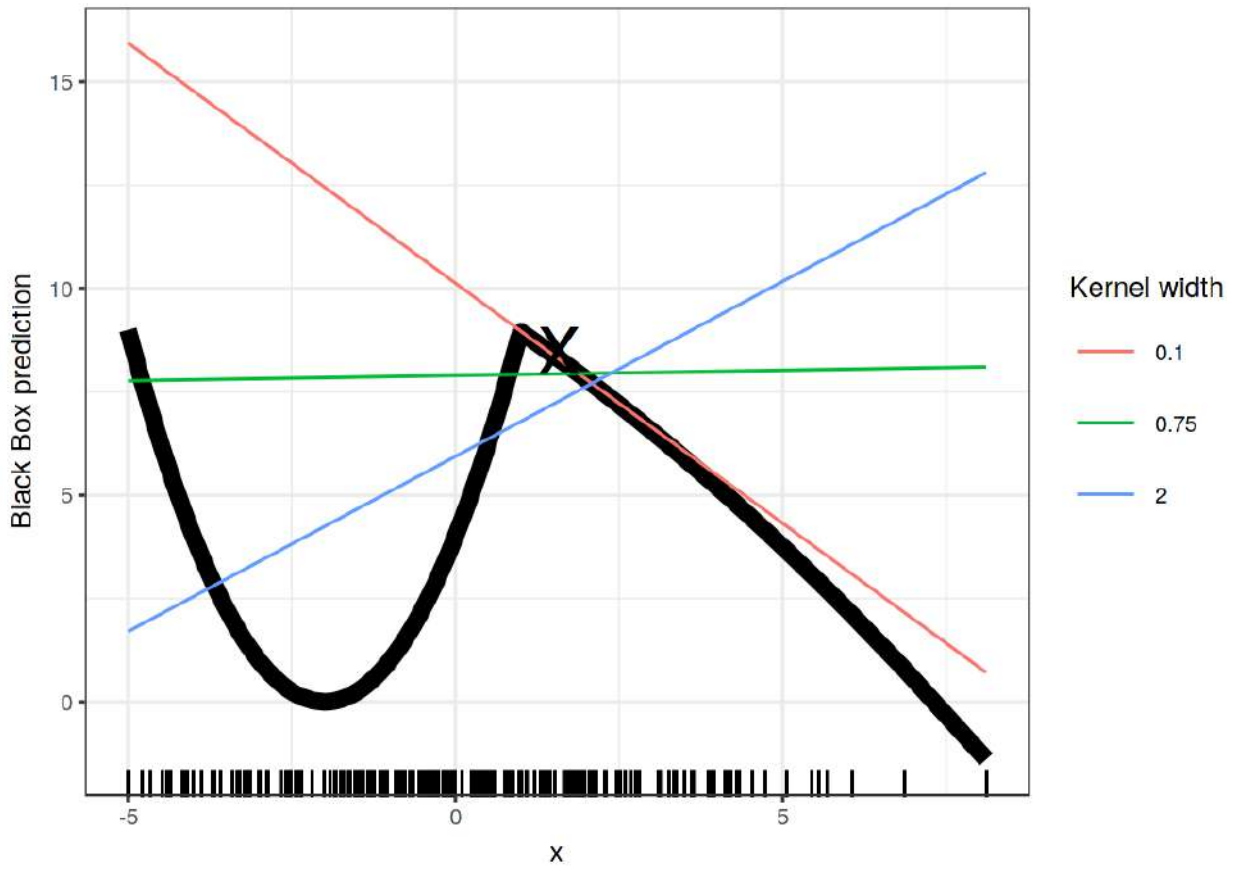
[80] https://github.com/thomasp85/lime
[81] https://github.com/marcotcr/lime



LIME algorithm for tabular data. A) Random forest predictions given features x1 and x2. Predicted classes: 1 (dark color) or 0 (light color). B) Instance of interest (yellow dot) and data sampled from a normal distribution (black dots). C) Assign higher weight to points near the instance of interest. D) Colors and signs of the grid show the classifications of the locally learned model from the weighted samples. The white line marks the decision boundary (P(class=1) = 0.5).

As always, the devil is in the detail. Defining a meaningful neighborhood around a point is difficult. LIME currently uses an exponential smoothing kernel to define the neighborhood. A smoothing kernel is a function that takes two data instances and returns a proximity measure. The kernel width determines how large the neighborhood is: A small kernel width means that an instance must be very close to influence the local model, a larger kernel width means that instances that are farther away

also influence the model. If you look at LIME's Python implementation (file lime/lime_tabular.py)[82] you will see that it uses an exponential smoothing kernel (on the normalized data) and the kernel width is 0.75 times the square root of the number of columns of the training data. It looks like an innocent line of code, but it is like an elephant sitting in your living room next to the good porcelain you got from your grandparents. The big problem is that we do not have a good way to find the best kernel or width. And where does the 0.75 even come from? In certain scenarios, you can easily turn your explanation around by changing the kernel width, as shown in the following figure:



Explanation of the prediction of instance x = 1.6. The predictions of the black box model depending on a single feature is shown as a black line and the distribution of the data is shown with rugs. Three local surrogate models with different kernel widths are computed. The resulting linear regression model depends on the kernel width: Does the feature have a negative, positive or no effect for x = 1.6?

The example shows only one feature. It gets worse in high-dimensional feature spaces. It is also very unclear whether the distance measure should treat all features equally. Is a distance unit for feature x1 identical to one unit for feature x2? Distance measures are quite arbitrary and distances in different dimensions (aka features) might not be comparable at all.

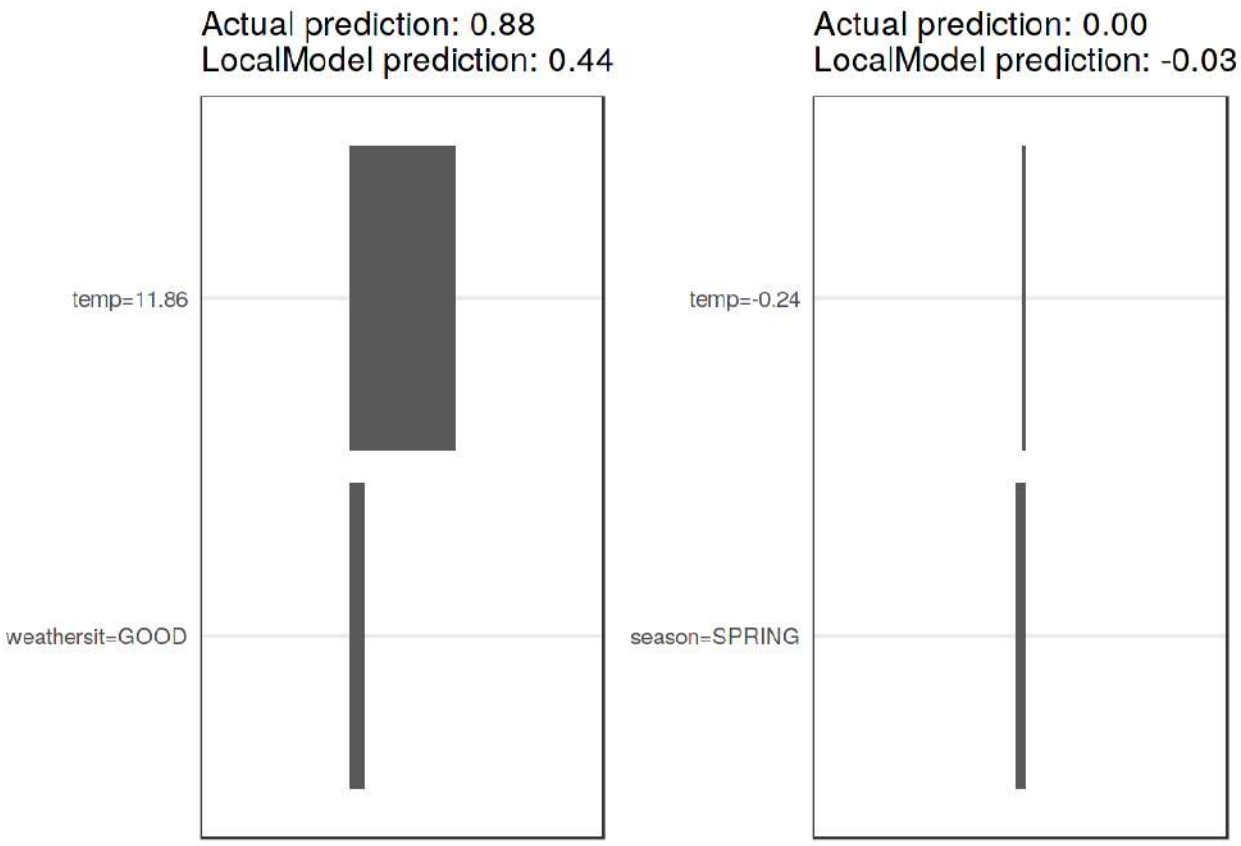
[82] https://github.com/marcotcr/lime/tree/ce2db6f20f47c3330beb107bb17fd25840ca4606

Example

Let us look at a concrete example. We go back to the bike rental data and turn the prediction problem into a classification: After taking into account the trend that the bicycle rental has become more popular over time, we want to know on a certain day whether the number of bicycles rented will be above or below the trend line. You can also interpret "above" as being above the average number of bicycles, but adjusted for the trend.

First we train a random forest with 100 trees on the classification task. On what day will the number of rental bikes be above the trend-free average, based on weather and calendar information?

The explanations are created with 2 features. The results of the sparse local linear models trained for two instances with different predicted classes:



LIME explanations for two instances of the bike rental dataset. Warmer temperature and good weather situation have a positive effect on the prediction. The x-axis shows the feature effect: The weight times the actual feature value.

From the figure it becomes clear that it is easier to interpret categorical features than numerical features. One solution is to categorize the numerical features into bins.

LIME for Text

LIME for text differs from LIME for tabular data. Variations of the data are generated differently: Starting from the original text, new texts are created by randomly removing words from the original text. The dataset is represented with binary features for each word. A feature is 1 if the corresponding word is included and 0 if it has been removed.

Example

In this example we classify YouTube comments as spam or normal.

The black box model is a deep decision tree trained on the document word matrix. Each comment is one document (= one row) and each column is the number of occurrences of a given word. Short decision trees are easy to understand, but in this case the tree is very deep. Also in place of this tree there could have been a recurrent neural network or a support vector machine trained on word embeddings (abstract vectors). Let us look at the two comments of this dataset and the corresponding classes (1 for spam, 0 for normal comment):

	CONTENT	CLASS
267	PSY is a good guy	0
173	For Christmas Song visit my channel! ;)	1

The next step is to create some variations of the datasets used in a local model. For example, some variations of one of the comments:

	For	Christmas	Song	visit	my	channel!	;)	prob	weight
2	1	0	1	1	0	0	1	0.09	0.57
3	0	1	1	1	1	0	1	0.09	0.71
4	1	0	0	1	1	1	1	0.99	0.71
5	1	0	1	1	1	1	1	0.99	0.86
6	0	1	1	1	0	0	1	0.09	0.57

Each column corresponds to one word in the sentence. Each row is a variation, 1 means that the word is part of this variation and 0 means that the word has been removed. The corresponding sentence for one of the variations is "`Christmas Song visit my ;)`". The "prob" column shows the predicted probability of spam for each of the sentence variations. The "weight" column shows the proximity of the variation to the original sentence, calculated as 1 minus the proportion of words that were removed, for example if 1 out of 7 words was removed, the proximity is 1 - 1/7 = 0.86.

Here are the two sentences (one spam, one no spam) with their estimated local weights found by the LIME algorithm:

case	label_prob	feature	feature_weight
1	0.0872151	good	0.000000
1	0.0872151	a	0.000000
1	0.0872151	PSY	0.000000
2	0.9939759	channel!	6.908755
2	0.9939759	visit	0.000000
2	0.9939759	Song	0.000000

The word "channel" indicates a high probability of spam. For the non-spam comment no non-zero weight was estimated, because no matter which word is removed, the predicted class remains the same.

LIME for Images

This section was written by Verena Haunschmid.

LIME for images works differently than LIME for tabular data and text. Intuitively, it would not make much sense to perturb individual pixels, since many more than one pixel contribute to one class. Randomly changing individual pixels would probably not change the predictions by much. Therefore, variations of the images are created by segmenting the image into "superpixels" and turning superpixels off or on. Superpixels are interconnected pixels with similar colors and can be turned off by replacing each pixel with a user-defined color such as gray. The user can also specify a probability for turning off a superpixel in each permutation.

Example

Since the computation of image explanations is rather slow, the lime R package[83] contains a precomputed example which we will also use to show the output of the method. The explanations can be displayed directly on the image samples. Since we can have several predicted labels per image (sorted by probability), we can explain the top `n_labels`. For the following image the top 3 predictions were *electric guitar*; *acoustic guitar*; and *Labrador*.

LIME explanations for the top 3 classes for image classification made by Google's Inception neural network. The example is taken from the LIME paper (Ribeiro et. al., 2016).

[83] https://github.com/thomasp85/lime

The prediction and explanation in the first case are very reasonable. The first prediction of *electric guitar* is of course wrong, but the explanation shows us that the neural network still behaved reasonably because the image part identified suggests that this could be an electric guitar.

Advantages

Even if you **replace the underlying machine learning model**, you can still use the same local, interpretable model for explanation. Suppose the people looking at the explanations understand decision trees best. Because you use local surrogate models, you use decision trees as explanations without actually having to use a decision tree to make the predictions. For example, you can use a SVM. And if it turns out that an xgboost model works better, you can replace the SVM and still use as decision tree to explain the predictions.

Local surrogate models benefit from the literature and experience of training and interpreting interpretable models.

When using Lasso or short trees, the resulting **explanations are short (= selective) and possibly contrastive**. Therefore, they make human-friendly explanations. This is why I see LIME more in applications where the recipient of the explanation is a lay person or someone with very little time. It is not sufficient for complete attributions, so I do not see LIME in compliance scenarios where you might be legally required to fully explain a prediction. Also for debugging machine learning models, it is useful to have all the reasons instead of a few.

LIME is one of the few methods that **works for tabular data, text and images**.

The **fidelity measure** (how well the interpretable model approximates the black box predictions) gives us a good idea of how reliable the interpretable model is in explaining the black box predictions in the neighborhood of the data instance of interest.

LIME is implemented in Python (lime[84] and Skater[85]) and R (lime package[86] and iml package[87]) and is **very easy to use**.

The explanations created with local surrogate models **can use other features than the original model**. This can be a big advantage over other methods, especially if the original features cannot bet interpreted. A text classifier can rely on abstract word embeddings as features, but the explanation can be based on the presence or absence of words in a sentence. A regression model can rely on a non-interpretable transformation of some attributes, but the explanations can be created with the original attributes.

Disadvantages

The correct definition of the neighborhood is a very big, unsolved problem when using LIME with tabular data. In my opinion it is the biggest problem with LIME and the reason why I would

[84] https://github.com/marcotcr/lime
[85] https://github.com/datascienceinc/Skater
[86] https://cran.r-project.org/web/packages/lime/index.html
[87] https://cran.r-project.org/web/packages/iml/index.html

recommend to use LIME only with great care. For each application you have to try different kernel settings and see for yourself if the explanations make sense. Unfortunately, this is the best advice I can give to find good kernel widths.

Sampling could be improved in the current implementation of LIME. Data points are sampled from a Gaussian distribution, ignoring the correlation between features. This can lead to unlikely data points which can then be used to learn local explanation models.

The complexity of the explanation model has to be defined in advance. This is just a small complaint, because in the end the user always has to define the compromise between fidelity and sparsity.

Another really big problem is the instability of the explanations. In an article [88] the authors showed that the explanations of two very close points varied greatly in a simulated setting. Also, in my experience, if you repeat the sampling process, then the explantions that come out can be different. Instability means that it is difficult to trust the explanations, and you should be very critical.

Conclusion: Local surrogate models, with LIME as a concrete implementation, are very promising. But the method is still in development phase and many problems need to be solved before it can be safely applied.

[88] Alvarez-Melis, David, and Tommi S. Jaakkola. "On the robustness of interpretability methods." arXiv preprint arXiv:1806.08049 (2018).

Shapley Values

A prediction can be explained by assuming that each feature value of the instance is a "player" in a game where the prediction is the payout. The Shapley value – a method from coalitional game theory – tells us how to fairly distribute the "payout" among the features.

General Idea

Assume the following scenario:

You have trained a machine learning model to predict apartment prices. For a certain apartment it predicts €300,000 and you need to explain this prediction. The apartment has a size of 50 m², is located on the 2nd floor, has a park nearby and cats are banned:

The predicted price for a 50 m² 2nd floor apartment with a nearby park and cat ban is €300,000. Our goal is to explain how each of these feature values contributed to the prediction.

The average prediction for all apartments is €310,000. How much has each feature value contributed to the prediction compared to the average prediction?

The answer is simple for linear regression models. The effect of each feature is the weight of the feature times the feature value. This only works because of the linearity of the model. For more complex models, we need a different solution. For example, LIME suggests local models to estimate effects. Another solution comes from cooperative game theory: The Shapley value, coined by Shapley (1953)[89], is a method for assigning payouts to players depending on their contribution to the total payout. Players cooperate in a coalition and receive a certain profit from this cooperation.

[89]Shapley, Lloyd S. "A value for n-person games." Contributions to the Theory of Games 2.28 (1953): 307-317.

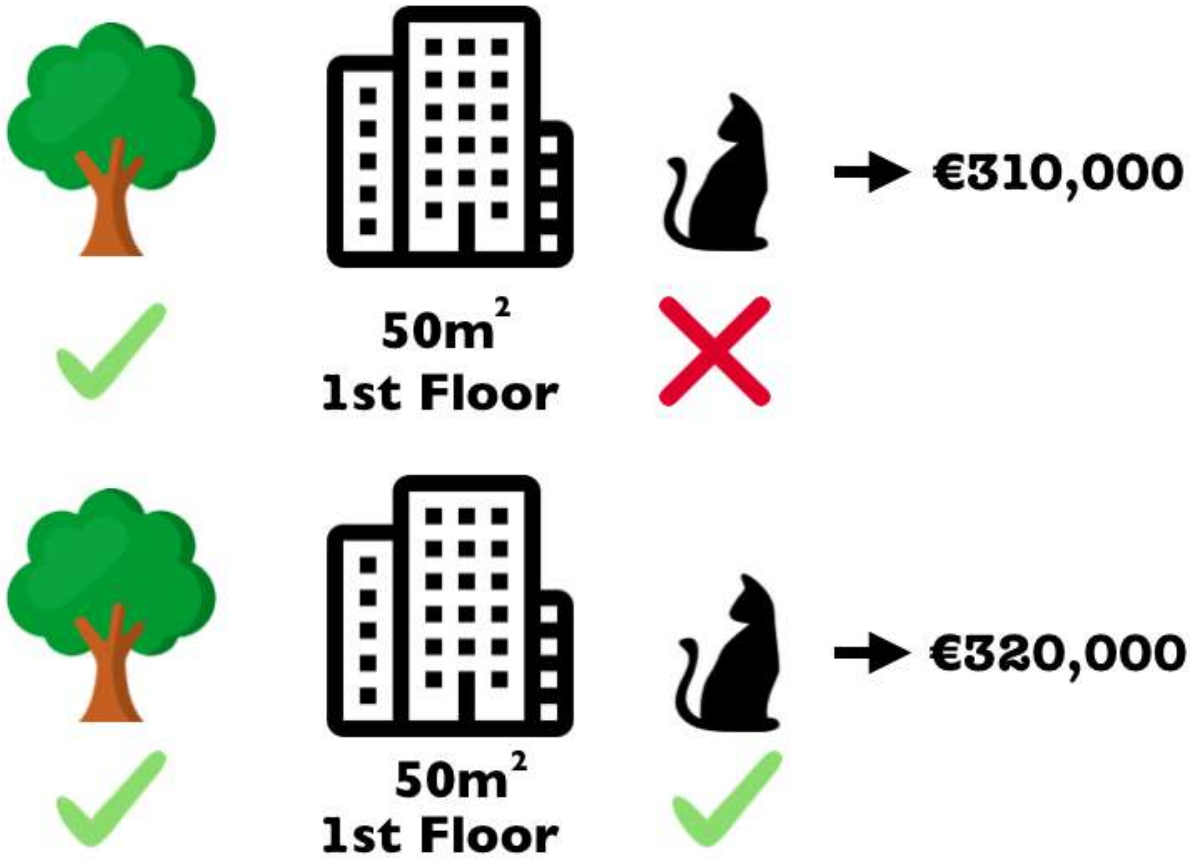
Players? Game? Payout? What is the connection to machine learning predictions and interpretability? The "game" is the prediction task for a single instance of the dataset. The "gain" is the actual prediction for this instance minus the average prediction for all instances. The "players" are the feature values of the instance that collaborate to receive the gain (= predict a certain value). In our apartment example, the feature values `park-nearby`, `cat-banned`, `area-50` and `floor-2nd` worked together to achieve the prediction of €300,000. Our goal is to explain the difference between the actual prediction (€300,000) and the average prediction (€310,000): a difference of -€10,000.

The answer could be: The `park-nearby` contributed €30,000; `size-50` contributed €10,000; `floor-2nd` contributed €0; `cat-banned` contributed -€50,000. The contributions add up to -€10,000, the final prediction minus the average predicted apartment price.

How do we calculate the Shapley value for one feature?

The Shapley value is the average marginal contribution of a feature value across all possible coalitions. All clear now?

In the following figure we evaluate the contribution of the `cat-banned` feature value when it is added to a coalition of `park-nearby` and `size-50`. We simulate that only `park-nearby`, `cat-banned` and `size-50` are in a coalition by randomly drawing another apartment from the data and using its value for the floor feature. The value `floor-2nd` was replaced by the randomly drawn `floor-1st`. Then we predict the price of the apartment with this combination (€310,000). In a second step, we remove `cat-banned` from the coalition by replacing it with a random value of the cat allowed/banned feature from the randomly drawn apartment. In the example it was `cat-allowed`, but it could have been `cat-banned` again. We predict the apartment price for the coalition of `park-nearby` and `size-50` (€320,000). The contribution of `cat-banned` was €310,000 - €320,000 = -€10.000. This estimate depends on the values of the randomly drawn apartment that served as a "donor" for the cat and floor feature values. We will get better estimates if we repeat this sampling step and average the contributions.



One sample repetition to estimate the contribution of `cat-banned` to the prediction when added to the coalition of `park-nearby` and `area-50`.

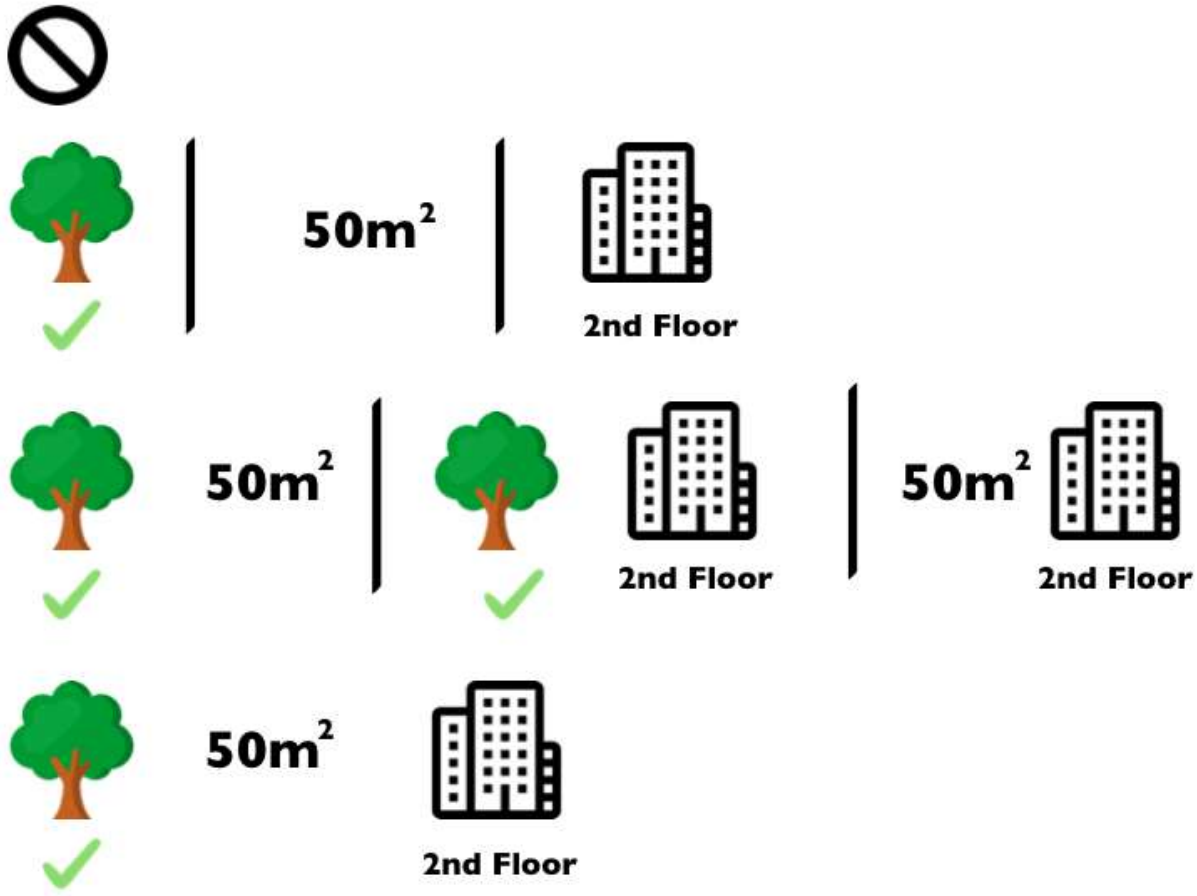
We repeat this computation for all possible coalitions. The Shapley value is the average of all the marginal contributions to all possible coalitions. The computation time increases exponentially with the number of features. One solution to keep the computation time manageable is to compute contributions for only a few samples of the possible coalitions.

The following figure shows all coalitions of feature values that are needed to determine the Shapley value for `cat-banned`. The first row shows the coalition without any feature values. The second, third and fourth rows show different coalitions with increasing coalition size, separated by "|". All in all, the following coalitions are possible:

- `No feature values`
- `park-nearby`
- `size-50`
- `floor-2nd`
- `park-nearby+size-50`
- `park-nearby+floor-2nd`
- `size-50+floor-2nd`

- `park-nearby+size-50+floor-2nd`.

For each of these coalitions we compute the predicted apartment price with and without the feature value `cat-banned` and take the difference to get the marginal contribution. The Shapley value is the (weighted) average of marginal contributions. We replace the feature values of features that are not in a coalition with random feature values from the apartment dataset to get a prediction from the machine learning model.



All 8 coalitions needed for computing the exact Shapley value of the `cat-banned` feature value.

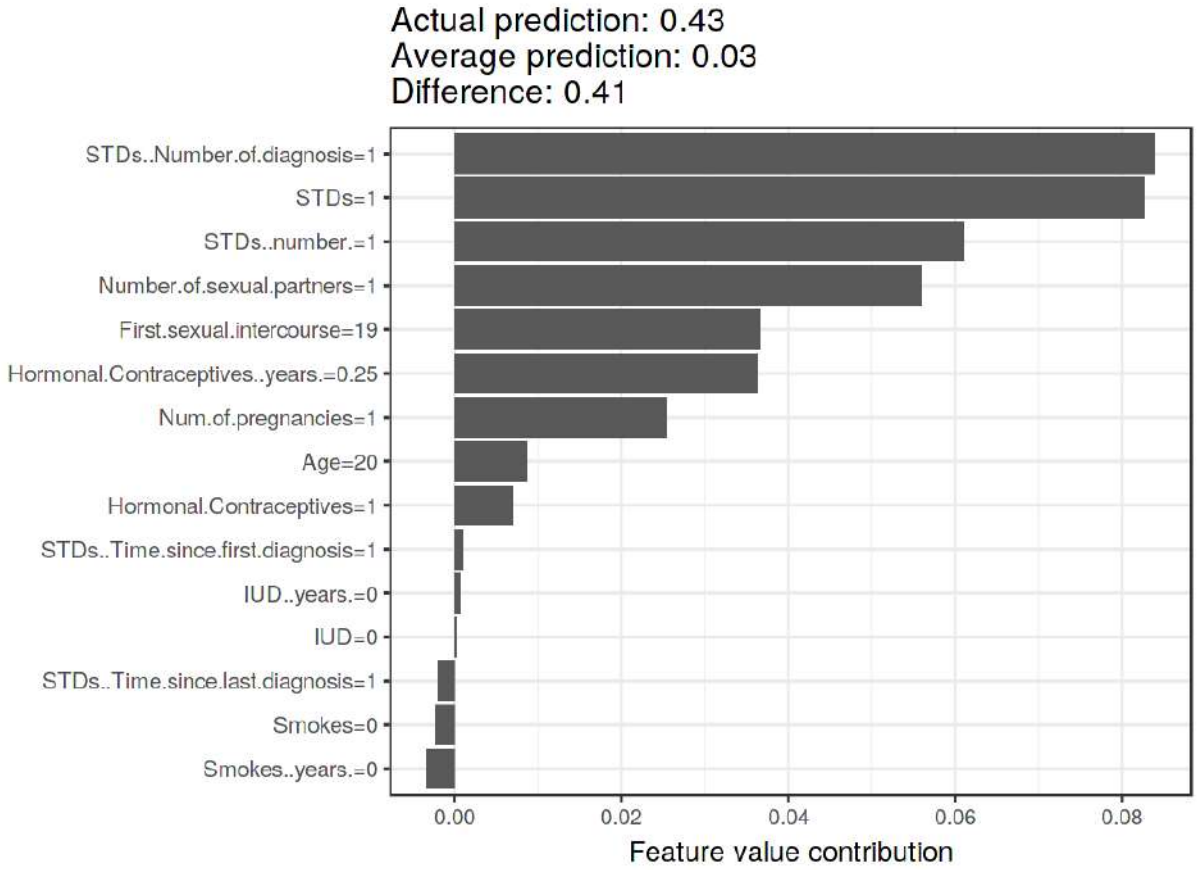
If we estimate the Shapley values for all feature values, we get the complete distribution of the prediction (minus the average) among the feature values.

Examples and Interpretation

The interpretation of the Shapley value for feature value j is: The value of the j-th feature contributed ϕ_j to the prediction of this particular instance compared to the average prediction for the dataset.

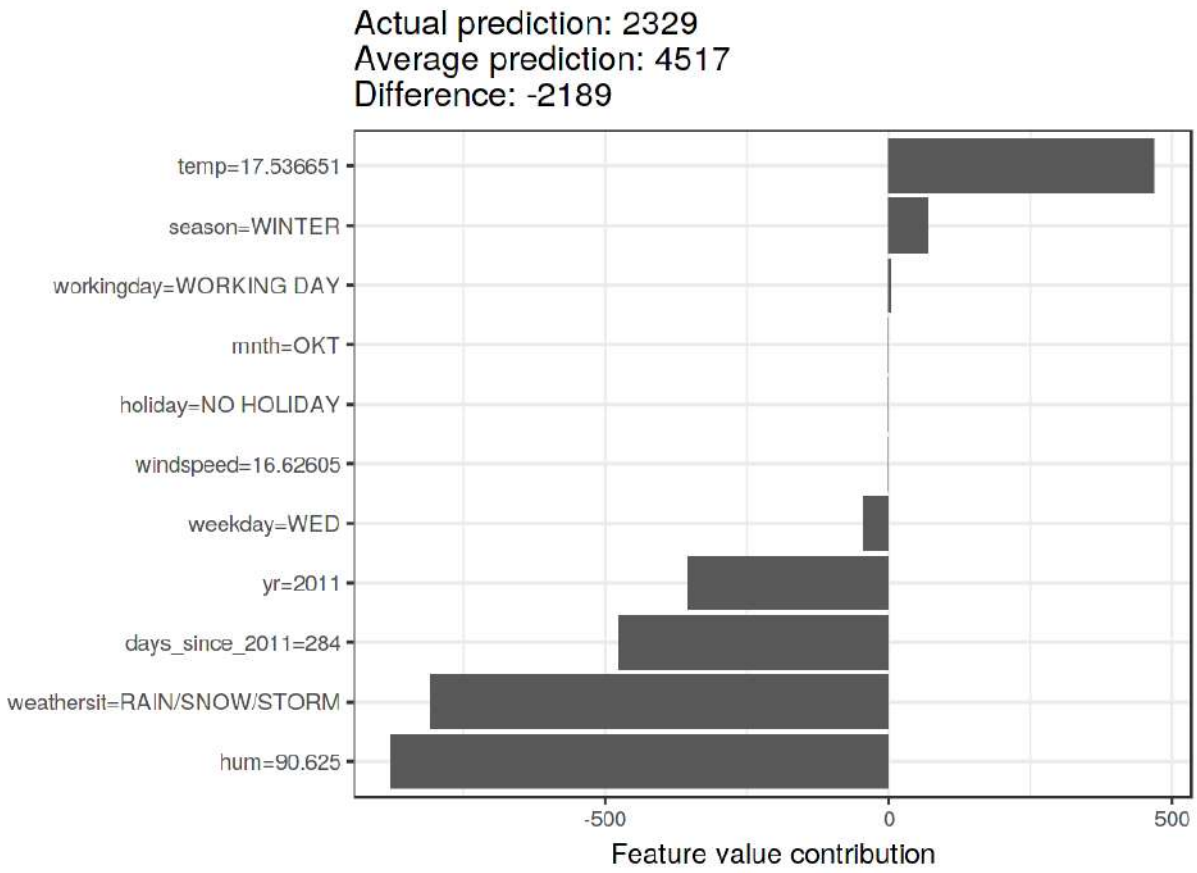
The Shapley value works for both classification (if we are dealing with probabilities) and regression.

We use the Shapley value to analyze the predictions of a random forest model predicting cervical cancer:



Shapley values for a woman in the cervical cancer dataset. With a prediction of 0.43, this woman's cancer probability is 0.41 above the average prediction of 0.03. The number of diagnosed STDs increased the probability the most. The sum of contributions yields the difference between actual and average prediction (0.41).

For the bike rental dataset, we also train a random forest to predict the number of rented bikes for a day, given weather and calendar information. The explanations created for the random forest prediction of a particular day:



Shapley values for day 285. With a predicted 2329 rental bikes, this day is -2189 below the average prediction of 4517. The weather situation and humidity and had the largest negative contributions. The temperature on this day had a positive contribution. The sum of Shapley values yields the difference of actual and average prediction (-2189).

Be careful to interpret the Shapley value correctly: The Shapley value is the average contribution of a feature value to the prediction in different coalitions. The Shapley value is NOT the difference in prediction when we would remove the feature from the model.

The Shapley Value in Detail

This section goes deeper into the definition and computation of the Shapley value for the curious reader. Skip this section and go directly to "Advantages and Disadvantages" if you are not interested in the technical details.

We are interested in how each feature affects the prediction of a data point. In a linear model it is easy to calculate the individual effects. Here is what a linear model prediction looks like for one data instance:

$$\hat{f}(x) = \beta_0 + \beta_1 x_1 + \ldots + \beta_p x_p$$

where x is the instance for which we want to compute the contributions. Each x_j is a feature value, with j = 1,...,p. The β_j is the weight corresponding to feature j.

The contribution ϕ_j of the j-th feature on the prediction $\hat{f}(x)$ is:

$$\phi_j(\hat{f}) = \beta_j x_j - E(\beta_j X_j) = \beta_j x_j - \beta_j E(X_j)$$

where $E(\beta_j X_j)$ is the mean effect estimate for feature j. The contribution is the difference between the feature effect minus the average effect. Nice! Now we know how much each feature contributed to the prediction. If we sum all the feature contributions for one instance, the result is the following:

$$\begin{aligned}\sum_{j=1}^{p} \phi_j(\hat{f}) &= \sum_{j=1}^{p}(\beta_j x_j - E(\beta_j X_j)) \\ &= (\beta_0 + \sum_{j=1}^{p} \beta_j x_j) - (\beta_0 + \sum_{j=1}^{p} E(\beta_j X_j)) \\ &= \hat{f}(x) - E(\hat{f}(X))\end{aligned}$$

This is the predicted value for the data point x minus the average predicted value. Feature contributions can be negative.

Can we do the same for any type of model? It would be great to have this as a model-agnostic tool. Since we usually do not have similar weights in other model types, we need a different solution.

Help comes from unexpected places: cooperative game theory. The Shapley value is a solution for computing feature contributions for single predictions for any machine learning model.

The Shapley Value

The Shapley value is defined via a value function val of players in S.

The Shapley value of a feature value is its contribution to the payout, weighted and summed over all possible feature value combinations:

$$\phi_j(val) = \sum_{S \subseteq \{x_1,\ldots,x_p\} \setminus \{x_j\}} \frac{|S|!\,(p - |S| - 1)!}{p!} \left(val\left(S \cup \{x_j\}\right) - val(S)\right)$$

where S is a subset of the features used in the model, x is the vector of feature values of the instance to be explained and p the number of features. $val_x(S)$ is the prediction for feature values in set S that are marginalized over features that are not included in set S:

$$val_x(S) = \int \hat{f}(x_1, \ldots, x_p) d\mathbb{P}_{x \notin S} - E_X(\hat{f}(X))$$

You actually perform multiple integrations for each feature that is not contained S. A concrete example: The machine learning model works with 4 features x1, x2, x3 and x4 and we evaluate the prediction for the coalition S consisting of feature values x1 and x3:

$$val_x(S) = val_x(\{x_1, x_3\}) = \int_{\mathbb{R}} \int_{\mathbb{R}} \hat{f}(x_1, X_2, x_3, X_4) d\mathbb{P}_{X_2 X_4} - E_X(\hat{f}(X))$$

This looks similar to the feature contributions in the linear model!

Do not get confused by the many uses of the word "value": The feature value is the numerical or categorical value of a feature and instance; the Shapley value is the feature contribution to the prediction; the value function is the payout function for coalitions of players (feature values).

The Shapley value is the only attribution method that satisfies the properties **Efficiency**, **Symmetry**, **Dummy** and **Additivity**, which together can be considered a definition of a fair payout.

Efficiency
The feature contributions must add up to the difference of prediction for x and the average.

$$\sum_{j=1}^{p} \phi_j = \hat{f}(x) - E_X(\hat{f}(X))$$

Symmetry
The contributions of two feature values j and k should be the same if they contribute equally to all possible coalitions. If

$$val(S \cup \{x_j\}) = val(S \cup \{x_k\})$$

for all

$$S \subseteq \{x_1, \ldots, x_p\} \setminus \{x_j, x_k\}$$

then

$$\phi_j = \phi_k$$

Dummy
A feature j that does not change the predicted value – regardless of which coalition of feature values it is added to – should have a Shapley value of 0. If

$$val(S \cup \{x_j\}) = val(S)$$

for all

$$S \subseteq \{x_1, \ldots, x_p\}$$

then

$$\phi_j = 0$$

Additivity
For a game with combined payouts val+val⁺ the respective Shapley values are as follows:

$$\phi_j + \phi_j^+$$

Suppose you trained a random forest, which means that the prediction is an average of many decision trees. The Additivity property guarantees that for a feature value, you can calculate the Shapley value for each tree individually, average them, and get the Shapley value for the feature value for the random forest.

Intuition

An intuitive way to understand the Shapley value is the following illustration: The feature values enter a room in random order. All feature values in the room participate in the game (= contribute to the prediction). The Shapley value of a feature value is the average change in the prediction that the coalition already in the room receives when the feature value joins them.

Estimating the Shapley Value

All possible coalitions (sets) of feature values have to be evaluated with and without the j-th feature to calculate the exact Shapley value. For more than a few features, the exact solution to this problem becomes problematic as the number of possible coalitions exponentially increases as more features are added. Strumbelj et al. (2014)[90] propose an approximation with Monte-Carlo sampling:

$$\hat{\phi}_j = \frac{1}{M} \sum_{m=1}^{M} \left(\hat{f}(x_{+j}^m) - \hat{f}(x_{-j}^m) \right)$$

where $\hat{f}(x_{+j}^m)$ is the prediction for x, but with a random number of feature values replaced by feature values from a random data point z, except for the respective value of feature j. The x-vector x_{-j}^m is almost identical to x_{+j}^m, but the value x_j^m is also taken from the sampled x. Each of these M new instances is a kind of "Frankenstein Monster" assembled from two instances.

Approximate Shapley estimation for single feature value:

[90] Štrumbelj, Erik, and Igor Kononenko. "Explaining prediction models and individual predictions with feature contributions." Knowledge and information systems 41.3 (2014): 647-665.

- Output: Shapley value for the value of the j-th feature
- Required: Number of iterations M, instance of interest x, feature index j, data matrix X, and machine learning model f
  - For all m = 1,...,M:
    * Draw random instance z from the data matrix X
    * Choose a random permutation o of the feature values
    * Order instance x: $x_o = (x_{(1)}, \ldots, x_{(j)}, \ldots, x_{(p)})$
    * Order instance z: $z_o = (z_{(1)}, \ldots, z_{(j)}, \ldots, z_{(p)})$
    * Construct two new instances
      .
      $$x_{+j} = (x_{(1)}, \ldots, x_{(j-1)}, x_{(j)}, z_{(j+1)}, \ldots, z_{(p)})$$
      .
      $$x_{-j} = (x_{(1)}, \ldots, x_{(j-1)}, z_{(j)}, z_{(j+1)}, \ldots, z_{(p)})$$
    *
      $$\phi_j^m = \hat{f}(x_{+j}) - \hat{f}(x_{-j})$$
- Compute Shapley value as the average: $\phi_j(x) = \frac{1}{M} \sum_{m=1}^{M} \phi_j^m$

First, select an instance of interest x, a feature j and the number of iterations M. For each iteration, a random instance z is selected from the data and a random order of the features is generated. Two new instances are created by combining values from the instance of interest x and the sample z. The first instance x_{+j} is the instance of interest, but all values in the order before and including value of feature j are replaced by feature values from the sample z. The second instance x_{-j} is similar, but has all the values in the order before, but excluding feature j replaced by values of feature j from the sample z. The difference in the prediction from the black box is computed:

$$\phi_j^m = \hat{f}(x_{+j}^m) - \hat{f}(x_{-j}^m)$$

All these differences are averaged and result in:

$$\phi_j(x) = \frac{1}{M} \sum_{m=1}^{M} \phi_j^m$$

Averaging implicitly weighs samples by the probability distribution of X.

The procedure has to be repeated for each of the features to get all Shapley values.

Advantages

The difference between the prediction and the average prediction is **fairly distributed** among the feature values of the instance – the Efficiency property of Shapley values. This property distinguishes the Shapley value from other methods such as LIME. LIME does not guarantee that the prediction is

fairly distributed among the features. The Shapley value might be the only method to deliver a full explanation. In situations where the law requires explainability – like EU's "right to explanations" – the Shapley value might be the only legally compliant method, because it is based on a solid theory and distributes the effects fairly. I am not a lawyer, so this reflects only my intuition about the requirements.

The Shapley value allows **contrastive explanations**. Instead of comparing a prediction to the average prediction of the entire dataset, you could compare it to a subset or even to a single data point. This contrastiveness is also something that local models like LIME do not have.

The Shapley value is the only explanation method with a **solid theory**. The axioms – efficiency, symmetry, dummy, additivity – give the explanation a reasonable foundation. Methods like LIME assume linear behavior of the machine learning model locally, but there is no theory as to why this should work.

It is mind-blowing to **explain a prediction as a game** played by the feature values.

Disadvantages

The Shapley value requires **a lot of computing time**. In 99.9% of real-world problems, only the approximate solution is feasible. An exact computation of the Shapley value is computationally expensive because there are 2^k possible coalitions of the feature values and the "absence" of a feature has to be simulated by drawing random instances, which increases the variance for the estimate of the Shapley values estimation. The exponential number of the coalitions is dealt with by sampling coalitions and limiting the number of iterations M. Decreasing M reduces computation time, but increases the variance of the Shapley value. It is unclear how to choose a sensitive M. It should be possible to choose M based on Chernoff bounds, but I have not seen any paper on doing this for Shapley values for machine learning predictions.

The Shapley value **can be misinterpreted**. The Shapley value of a feature value is not the difference of the predicted value after removing the feature from the model training. The interpretation of the Shapley value is: Given the current set of feature values, the contribution of a feature value to the difference between the actual prediction and the mean prediction is the estimated Shapley value.

The Shapley value is the wrong explanation method if you seek sparse explanations (explanations that contain few features). Explanations created with the Shapley value method **always use all the features**. Humans prefer selective explanations, such as those produced by LIME. LIME might be the better choice for explanations lay-persons have to deal with. Another solution is SHAP[91] introduced by Lundberg and Lee (2016)[92], which is based on the Shapley value, but can also provide explanations with few features.

The Shapley value returns a simple value per feature, but **no prediction model** like LIME. This means it cannot be used to make statements about changes in prediction for changes in the input, such as: "If I were to earn €300 more a year, my credit score would increase by 5 points."

[91] https://github.com/slundberg/shap

[92] Lundberg, Scott, and Su-In Lee. "An unexpected unity among methods for interpreting model predictions." arXiv preprint arXiv:1611.07478 (2016).

Another disadvantage is that **you need access to the data** if you want to calculate the Shapley value for a new data instance. It is not sufficient to access the prediction function because you need the data to replace parts of the instance of interest with values from randomly drawn instances of the data. This can only be avoided if you can create data instances that look like real data instances but are not actual instances from the training data.

Like many other permutation-based interpretation methods, the Shapley value method suffers from **inclusion of unrealistic data instances** when features are correlated. To simulate that a feature value is missing from a coalition, we marginalize the feature. This is achieved by sampling values from the feature's marginal distribution. This is fine as long as the features are independent. When features are dependent, then we might sample feature values that do not make sense for this instance. But we would use those to compute the feature's Shapley value. To the best of my knowledge, there is no research on what that means for the Shapley values, nor a suggestion on how to fix it. One solution might be to permute correlated features together and get one mutual Shapley value for them. Or the sampling procedure might have to be adjusted to account for dependence of features.

Software and Alternatives

Shapley values are implemented in the `iml` R package.

SHAP, an alternative formulation of the Shapley values, is implemented in the Python package `shap`. SHAP turns the Shapley values method into an optimization problem and uses a special kernel function to measure proximity of data instances. The results of SHAP are sparse (many Shapley values are estimated to be zero), which is the biggest difference from the classic Shapley values.

Another approach is called breakDown, which is implemented in the `breakDown` R package[93]. BreakDown also shows the contributions of each feature to the prediction, but computes them step by step. Let us reuse the game analogy: We start with an empty team, add the feature value that would contribute the most to the prediction and iterate until all feature values are added. How much each feature value contributes depends on the respective feature values that are already in the "team", which is the big drawback of the breakDown method. It is faster than the Shapley value method, and for models without interactions, the results are the same.

[93]Staniak, Mateusz, and Przemyslaw Biecek. "Explanations of model predictions with live and breakDown packages." arXiv preprint arXiv:1804.01955 (2018).

Example-Based Explanations

Example-based explanation methods select particular instances of the dataset to explain the behavior of machine learning models or to explain the underlying data distribution.

Example-based explanations are mostly model-agnostic, because they make any machine learning model more interpretable. The difference to model-agnostic methods is that the example-based methods explain a model by selecting instances of the dataset and not by creating summaries of features (such as feature importance or partial dependence). Example-based explanations only make sense if we can represent an instance of the data in a humanly understandable way. This works well for images, because we can view them directly. In general, example-based methods work well if the feature values of an instance carry more context, meaning the data has a structure, like images or texts do. It is more challenging to represent tabular data in a meaningful way, because an instance can consist of hundreds or thousands of (less structured) features. Listing all feature values to describe an instance is usually not useful. It works well if there are only a handful of features or if we have a way to summarize an instance.

Example-based explanations help humans construct mental models of the machine learning model and the data the machine learning model has been trained on. It especially helps to understand complex data distributions. But what do I mean by example-based explanations? We often use them in our jobs and daily lives. Let us start with some examples[94].

A physician sees a patient with an unusual cough and a mild fever. The patient's symptoms remind her of another patient she had years ago with similar symptoms. She suspects that her current patient could have the same disease and she takes a blood sample to test for this specific disease.

A data scientist works on a new project for one of his clients: Analysis of the risk factors that lead to the failure of production machines for keyboards. The data scientist remembers a similar project he worked on and reuses parts of the code from the old project because he thinks the client wants the same analysis.

A kitten sits on the window ledge of a burning and uninhabited house. The fire department has already arrived and one of the firefighters ponders for a second whether he can risk going into the building to save the kitten. He remembers similar cases in his life as a firefighter: Old wooden houses that have been burning slowly for some time were often unstable and eventually collapsed. Because of the similarity of this case, he decides not to enter, because the risk of the house collapsing is too great. Fortunately, the kitty jumps out of the window, lands safely and nobody is harmed in the fire. Happy end.

These stories illustrate how we humans think in examples or analogies. The blueprint of example-based explanations is: Thing B is similar to thing A and A caused Y, so I predict that B will cause Y

[94]Aamodt, Agnar, and Enric Plaza. "Case-based reasoning: Foundational issues, methodological variations, and system approaches." AI communications 7.1 (1994): 39-59.

as well. Implicitly, some machine learning approaches work example-based. Decision trees partition the data into nodes based on the similarities of the data points in the features that are important for predicting the target. A decision tree gets the prediction for a new data instance by finding the instances that are similar (= in the same terminal node) and returning the average of the outcomes of those instances as the prediction. The k-nearest neighbors (knn) method works explicitly with example-based predictions. For a new instance, a knn model locates the k-nearest neighbors (e.g. the k=3 closest instances) and returns the average of the outcomes of those neighbors as a prediction. The prediction of a knn can be explained by returning the k neighbors, which – again – is only meaningful if we have a good way to represent a single instance.

The chapters in this part cover the following example-based interpretation methods:

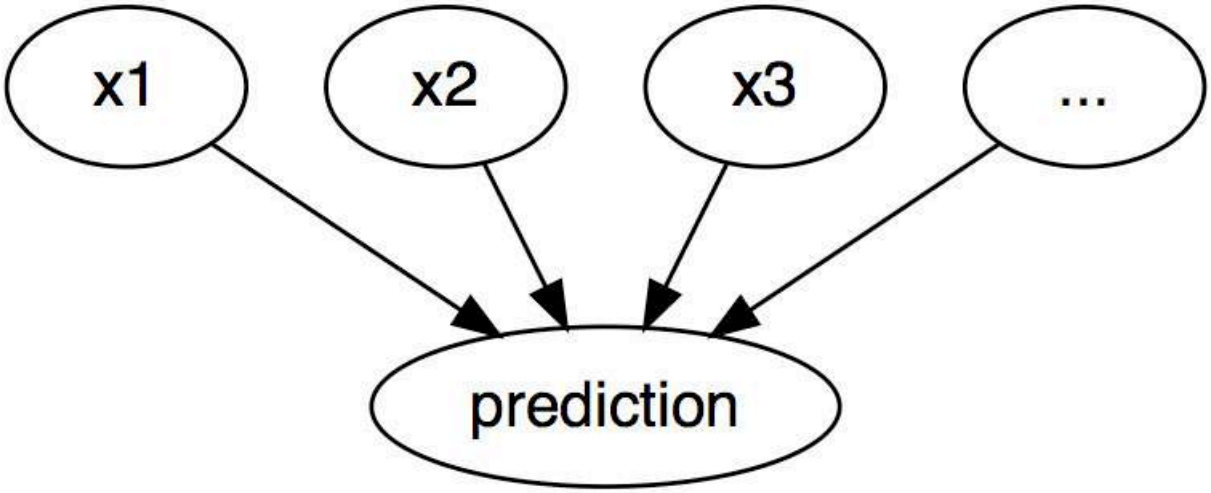
- Counterfactual explanations tell us how an instance has to change to significantly change its prediction. By creating counterfactual instances, we learn about how the model makes its predictions and can explain individual predictions.
- Adversarial examples are counterfactuals used to fool machine learning models. The emphasis is on flipping the prediction and not explaining it.
- Prototypes are a selection of representative instances from the data and criticisms are instances that are not well represented by those prototypes. [95]
- Influential instances are the training data points that were the most influential for the parameters of a prediction model or the predictions themselves. Identifying and analysing influential instances helps to find problems with the data, debug the model and understand the model's behavior better.
- k-nearest neighbors model: An (interpretable) machine learning model based on examples.

[95] Kim, Been, Rajiv Khanna, and Oluwasanmi O. Koyejo. "Examples are not enough, learn to criticize! Criticism for interpretability." Advances in Neural Information Processing Systems (2016).

Counterfactual Explanations

A counterfactual explanation describes a causal situation in the form: "If X had not occurred, Y would not have occurred". For example: "If I hadn't taken a sip of this hot coffee, I wouldn't have burned my tongue". Event Y is that I burned my tongue; cause X is that I had a hot coffee. Thinking in counterfactuals requires imagining a hypothetical reality that contradicts the observed facts (e.g. a world in which I have not drunk the hot coffee), hence the name "counterfactual". The ability to think in counterfactuals makes us humans so smart compared to other animals.

In interpretable machine learning, counterfactual explanations can be used to explain predictions of individual instances. The "event" is the predicted outcome of an instance, the "causes" are the particular feature values of this instance that were input to the model and "caused" a certain prediction. Displayed as a graph, the relationship between the inputs and the prediction is very simple: The feature values cause the prediction.



The causal relationships between inputs of a machine learning model and the predictions, when the model is merely seen as a black box. The inputs cause the prediction (not necessarily reflecting the real causal relation of the data).

Even if in reality the relationship between the inputs and the outcome to be predicted might not be causal, we can see the inputs of a model as the cause of the prediction.

Given this simple graph, it is easy to see how we can simulate counterfactuals for predictions of machine learning models: We simply change the feature values of an instance before making the predictions and we analyze how the prediction changes. We are interested in scenarios in which the prediction changes in a relevant way, like a flip in predicted class (e.g. credit application accepted or rejected) or in which the prediction reaches a certain threshold (e.g. the probability for cancer reaches 10%). **A counterfactual explanation of a prediction describes the smallest change to the feature values that changes the prediction to a predefined output.**

The counterfactual explanation method is model-agnostic, since it only works with the model

inputs and output. This method would also feel at home in the model-agnostic chapter, since the interpretation can be expressed as a summary of the differences in feature values ("change features A and B to change the prediction"). But a counterfactual explanation is itself a new instance, so it lives in this chapter ("starting from instance X, change A and B to get a counterfactual instance"). Unlike prototypes, counterfactuals do not have to be actual instances from the training data, but can be a new combination of feature values.

Before discussing how to create counterfactuals, I would like to discuss some use cases for counterfactuals and how a good counterfactual explanation looks like.

In this first example, Peter applies for a loan and gets rejected by the (machine learning powered) banking software. He wonders why his application was rejected and how he might improve his chances to get a loan. The question of "why" can be formulated as a counterfactual: What is the smallest change to the features (income, number of credit cards, age, ...) that would change the prediction from rejected to approved? One possible answer could be: If Peter would earn 10,000 Euro more per year, he would get the loan. Or if Peter had fewer credit cards and had not defaulted on a loan 5 years ago, he would get the loan. Peter will never know the reasons for the rejection, as the bank has no interest in transparency, but that is another story.

In our second example we want to explain a model that predicts a continuous outcome with counterfactual explanations. Anna wants to rent out her apartment, but she is not sure how much to charge for it, so she decides to train a machine learning model to predict the rent. Of course, since Anna is a data scientist, that is how she solves her problems. After entering all the details about size, location, whether pets are allowed and so on, the model tells her that she can charge 900 Euro. She expected 1000 Euro or more, but she trusts her model and decides to play with the feature values of the apartment to see how she can improve the value of the apartment. She finds out that the apartment could be rented out for over 1000 Euro, if it were 15 m^2 larger. Interesting, but non-actionable knowledge, because she cannot enlarge her apartment. Finally, by tweaking only the feature values under her control (built-in kitchen yes/no, pets allowed yes/no, type of floor, etc.), she finds out that if she allows pets and installs windows with better insulation, she can charge 1000 Euro. Anna had intuitively worked with counterfactuals to change the outcome.

Counterfactuals are human-friendly explanations, because they are contrastive to the current instance and because they are selective, meaning they usually focus on a small number of feature changes. But counterfactuals suffer from the 'Rashomon effect'. Rashomon is a Japanese movie in which the murder of a Samurai is told by different people. Each of the stories explains the outcome equally well, but the stories contradict each other. The same can also happen with counterfactuals, since there are usually multiple different counterfactual explanations. Each counterfactual tells a different "story" of how a certain outcome was reached. One counterfactual might say to change feature A, the other counterfactual might say to leave A the same but change feature B, which is a contradiction. This issue of multiple truths can be addressed either by reporting all counterfactual explanations or by having a criterion to evaluate counterfactuals and select the best one.

Speaking of criteria, how do we define a good counterfactual explanation? First, the user of a counterfactual explanation defines a relevant change in the prediction of an instance (= the alternative reality), so an obvious first requirement is that **a counterfactual instance produces**

the predefined prediction as closely as possible. It is not always possible to match the predefined output exactly. In a classification setting with two classes, a rare class and a frequent class, the model could always classify an instance as the frequent class. Changing the feature values so that the predicted label would flip from the common class to the rare class might be impossible. We therefore want to relax the requirement that the predicted output of the counterfactual must correspond exactly to the defined outcome. In the classification example, we could look for a counterfactual where the predicted probability of the rare class is increased to 10% instead of the current 2%. The question then is, what are the minimum changes to the features so that the predicted probability changes from 2% to 10% (or close to 10%)? Another quality criterion is that **a counterfactual should be as similar as possible to the instance regarding feature values**. This requires a distance measure between two instances. The counterfactual should not only be close to the original instance, but should also **change as few features as possible**. This can be achieved by selecting an appropriate distance measure like the Manhattan distance. The last requirement is that **a counterfactual instance should have feature values that are likely**. It would not make sense to generate a counterfactual explanation for the rent example where the size of an apartment is negative or the number of rooms is set to 200. It is even better when the counterfactual is likely according to the joint distribution of the data, e.g. an apartment with 10 rooms and 20 m² should not be regarded as counterfactual explanation.

Generating Counterfactual Explanations

A simple and naive approach to generating counterfactual explanations is searching by trial and error. This approach involves randomly changing feature values of the instance of interest and stopping when the desired output is predicted. Like the example where Anna tried to find a version of her apartment for which she could charge more rent. But there are better approaches than trial and error. First, we define a loss function that takes as input the instance of interest, a counterfactual and the desired (counterfactual) outcome. The loss measures how far the predicted outcome of the counterfactual is from the predefined outcome and how far the counterfactual is from the instance of interest. We can either optimize the loss directly with an optimization algorithm or by searching around the instance, as suggested in the "Growing Spheres" method (see Software and Alternatives).

In this section, I will present the approach suggested by Wachter et. al (2017)[96]. They suggest minimizing the following loss.

$$L(x, x', y', \lambda) = \lambda \cdot (\hat{f}(x') - y')^2 + d(x, x')$$

The first term is the quadratic distance between the model prediction for the counterfactual x' and the desired outcome y', which the user must define in advance. The second term is the distance d between the instance x to be explained and the counterfactual x', but more about this later. The parameter λ balances the distance in prediction (first term) against the distance in feature values (second term). The loss is solved for a given λ and returns a counterfactual x'. A higher value of λ

[96]Wachter, Sandra, Brent Mittelstadt, and Chris Russell. "Counterfactual explanations without opening the black box: Automated decisions and the GDPR." (2017).

means that we prefer counterfactuals that come close to the desired outcome y', a lower value means that we prefer counterfactuals x' that are very similar to x in the feature values. If λ is very large, the instance with the prediction that comes closest to y' will be selected, regardless how far it is away from x. Ultimately, the user must decide how to balance the requirement that the prediction for the counterfactual matches the desired outcome with the requirement that the counterfactual is similar to x. The authors of the method suggest instead of selecting a value for λ to select a tolerance ϵ for how far away the prediction of the counterfactual instance is allowed to be from y'. This constraint can be written as:

$$|\hat{f}(x') - y'| \leq \epsilon$$

To minimize this loss function, any suitable optimization algorithm can be used, e.g. Nelder-Mead. If you have access to the gradients of the machine learning model, you can use gradient-based methods like ADAM. The instance x to be explained, the desired output y' and the tolerance parameter ϵ must be set in advance. The loss function is minimized for x' and the (locally) optimal counterfactual x' returned while increasing λ until a sufficiently close solution is found (= within the tolerance parameter).

$$\arg\min_{x'} \max_{\lambda} L(x, x', y', \lambda)$$

The function d for measuring the distance between instance x and counterfactual x' is the Manhattan distance weighted feature-wise with the inverse median absolute deviation (MAD).

$$d(x, x') = \sum_{j=1}^{p} \frac{|x_j - x'_j|}{MAD_j}$$

The total distance is the sum of all p feature-wise distances, that is, the absolute differences of feature values between instance x and counterfactual x'. The feature-wise distances are scaled by the inverse of the median absolute deviation of feature j over the dataset defined as:

$$MAD_j = \text{median}_{i \in \{1,\ldots,n\}}(|x_{i,j} - \text{median}_{l \in \{1,\ldots,n\}}(x_{l,j})|)$$

The median of a vector is the value at which half of the vector values are greater and the other half smaller. The MAD is the equivalent of the variance of a feature, but instead of using the mean as the center and summing over the square distances, we use the median as the center and sum over the absolute distances. The proposed distance function has the advantage over the Euclidean distance that it introduces sparsity. This means that two points are closer to each other when less features are different. And it is more robust to outliers. Scaling with the MAD is necessary to bring all the features to the same scale – it should not matter whether you measure the size of an apartment in square meters or square feet.

The recipe for producing the counterfactuals is simple:

1. Select an instance x to be explained, the desired outcome y', a tolerance ϵ and a (low) initial value for λ.
2. Sample a random instance as initial counterfactual.
3. Optimize the loss with the initially sampled counterfactual as starting point.
4. While $|\hat{f}(x') - y'| > \epsilon$:
    - Increase λ.
    - Optimize the loss with the current counterfactual as starting point.
    - Return the counterfactual that minimizes the loss.
5. Repeat steps 2-3 and return the list of counterfactuals or the one that minimizes the loss.

Examples

Both examples are from the work of Wachter et. al (2017).

In the first example, the authors train a three-layer fully-connected neural network to predict a student's average grade of the first year at law school, based on grade point average (GPA) prior to law school, race and law school entrance exam scores. The goal is to find counterfactual explanations for each student that answer the following question: How would the input features need to be changed, to get a predicted score of 0? Since the scores have been normalized before, a student with a score of 0 is as good as the average of the students. A negative score means a below-average result, a positive score an above-average result.

The following table shows the learned counterfactuals:

Score	GPA	LSAT	Race	GPA x'	LSAT x'	Race x'
0.17	3.1	39.0	0	3.1	34.0	0
0.54	3.7	48.0	0	3.7	32.4	0
-0.77	3.3	28.0	1	3.3	33.5	0
-0.83	2.4	28.5	1	2.4	35.8	0
-0.57	2.7	18.3	0	2.7	34.9	0

The first column contains the predicted score, the next 3 columns the original feature values and the last 3 columns the counterfactual feature values that result in a score close to 0. The first two rows are students with above-average predictions, the other three rows below-average. The counterfactuals for the first two rows describe how the student features would have to change to decrease the predicted score and for the other three cases how they would have to change to increase the score to the average. The counterfactuals for increasing the score always change the race from black (coded with 1) to white (coded with 0) which shows a racial bias of the model. The GPA is not changed in the counterfactuals, but LSAT is.

The second example shows counterfactual explanations for predicted risk of diabetes. A three-layer fully-connected neural network is trained to predict the risk for diabetes depending on age, BMI, number of pregnancies and so on for women of Pima heritage. The counterfactuals answer the question: Which feature values must be changed to increase or decrease the risk score of diabetes to

0.5? The following counterfactuals were found:

- Person 1: If your 2-hour serum insulin level was 154.3, you would have a score of 0.51
- Person 2: If your 2-hour serum insulin level was 169.5, you would have a score of 0.51
- Person 3: If your Plasma glucose concentration was 158.3 and your 2-hour serum insulin level was 160.5, you would have a score of 0.51

Advantages

The interpretation of counterfactual explanations is very clear. If the feature values of an instance are changed according to the counterfactual, the prediction changes to the predefined prediction. There are no additional assumptions and no magic in the background. This also means it is not as dangerous as methods like LIME, where it is unclear how far we can extrapolate the local model for the interpretation.

The counterfactual method creates a new instance, but we can also summarize a counterfactual by reporting which feature values have changed. This gives us **two options for reporting our results**. You can either report the counterfactual instance or highlight which features have been changed between the instance of interest and the counterfactual instance.

The **counterfactual method does not require access to the data or the model**. It only requires access to the model's prediction function, which would also work via a web API, for example. This is attractive for companies which are audited by third parties or which are offering explanations for users without disclosing the model or data. A company has an interest in protecting model and data because of trade secrets or data protection reasons. Counterfactual explanations offer a balance between explaining model predictions and protecting the interests of the model owner.

The method **works also with systems that do not use machine learning**. We can create counterfactuals for any system that receives inputs and returns outputs. The system that predicts apartment rents could also consist of handwritten rules, and counterfactual explanations would still work.

The counterfactual explanation method is relatively easy to implement, since it is essentially a loss function that can be optimized with standard optimizer libraries. Some additional details must be taken into account, such as limiting feature values to meaningful ranges (e.g. only positive apartment sizes).

Disadvantages

For each instance you will usually find multiple counterfactual explanations (Rashomon effect). This is inconvenient – most people prefer simple explanations over the complexity of the real world. It is also a practical challenge. Let us say we generated 23 counterfactual explanations for one instance. Are we reporting them all? Only the best? What if they are all relatively "good", but very different? These questions must be answered anew for each project. It can also be advantageous to

have multiple counterfactual explanations, because then humans can select the ones that correspond to their previous knowledge.

There is **no guarantee that for a given tolerance ϵ a counterfactual instance is found**. That is not necessarily the fault of the method, but rather depends on the data.

The proposed method **does not handle categorical features** with many different levels well. The authors of the method suggested running the method separately for each combination of feature values of the categorical features, but this will lead to a combinatorial explosion if you have multiple categorical features with many values. For example, 6 categorical features with 10 unique levels would mean 1 million runs. A solution for only categorical features was proposed by Martens et. al (2014)[97]. A good solution would be to use an optimizer that solves problems with a mix of continuous and discrete inputs.

The counterfactuals method **lacks a general software implementation**. And a method is only useful if it is implemented. Fortunately, it should be easy to implement and hopefully I can remove this statement here soon.

Software and Alternatives

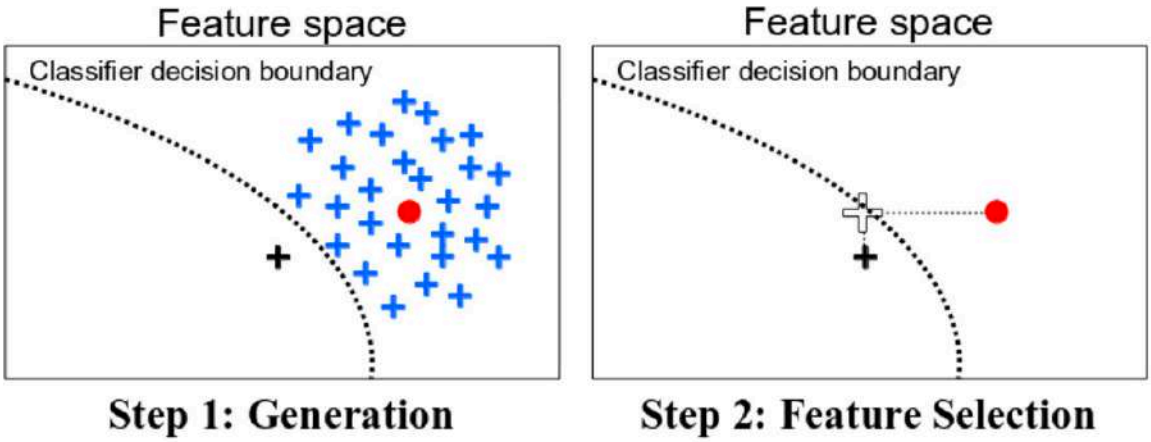
Unfortunately there is currently no software available for counterfactual explanations.

A very similar approach was proposed by Martens et. al (2014) for explaining document classifications. In their work, they focus on explaining why a document was or was not classified as a particular class. The difference to the method presented in this chapter is that Martens et. al (2014) focus on text classifiers, which have word occurrences as inputs.

An alternative way to search counterfactuals is the Growing Spheres algorithm by Laugel et. al (2017)[98]. The method first draws a sphere around the point of interest, samples points within that sphere, checks whether one of the sampled points yields the desired prediction, contracts or expands the sphere accordingly until a (sparse) counterfactual is found and finally returned. They do not use the word counterfactual in their paper, but the method is quite similar. They also define a loss function that favors counterfactuals with as few changes in the feature values as possible. Instead of directly optimizing the function, they suggest the above-mentioned search with spheres.

[97] Martens, David, and Foster Provost. "Explaining data-driven document classifications." (2014).

[98] Laugel, Thibault, et al. "Inverse classification for comparison-based interpretability in machine learning." arXiv preprint arXiv:1712.08443 (2017).



An illustration of Growing Spheres and selecting sparse counterfactuals by Laugel et. al (2017).

Anchors by Ribeiro et. al (2018)[99] are the opposite of counterfactuals. Anchors answer the question: Which features are sufficient to anchor a prediction, i.e. changing the other features cannot change the prediction? Once we have found features that serve as anchors for a prediction, we will no longer find counterfactual instances by changing the features not used in the anchor.

(a) Original image (b) Anchor for "beagle" (c) Images where Inception predicts $P(\text{beagle}) > 90\%$

What animal is featured in this picture ?	**dog**
What floor is featured in this picture?	dog
What toenail is paired in this flowchart ?	dog
What animal is shown on this depiction ?	dog

(d) **VQA:** Anchor (bold) and samples from $\mathcal{D}(z|A)$

Where is the **dog**?	on the floor
What color is the **wall?**	white
When was this picture taken?	during the day
Why is he lifting his paw?	to play

(e) **VQA:** More example anchors (in bold)

Examples for anchors by Ribeiro et. al (2018).

[99] Ribeiro, Marco Tulio, Sameer Singh, and Carlos Guestrin. "Anchors: High-precision model-agnostic explanations." AAAI Conference on Artificial Intelligence (2018).

Adversarial Examples

An adversarial example is an instance with small, intentional feature perturbations that cause a machine learning model to make a false prediction. I recommend reading the chapter about Counterfactual Explanations first, as the concepts are very similar. Adversarial examples are counterfactual examples with the aim to deceive the model, not interpret it.

Why are we interested in adversarial examples? Are they not just curious by-products of machine learning models without practical relevance? The answer is a clear "no". Adversarial examples make machine learning models vulnerable to attacks, as in the following scenarios.

A self-driving car crashes into another car because it ignores a stop sign. Someone had placed a picture over the sign, which looks like a stop sign with a little dirt for humans, but was designed to look like a parking prohibition sign for the sign recognition software of the car.

A spam detector fails to classify an email as spam. The spam mail has been designed to resemble a normal email, but with the intention of cheating the recipient.

A machine-learning powered scanner scans suitcases for weapons at the airport. A knife was developed to avoid detection by making the system think it is an umbrella.

Let us take a look at some ways to create adversarial examples.

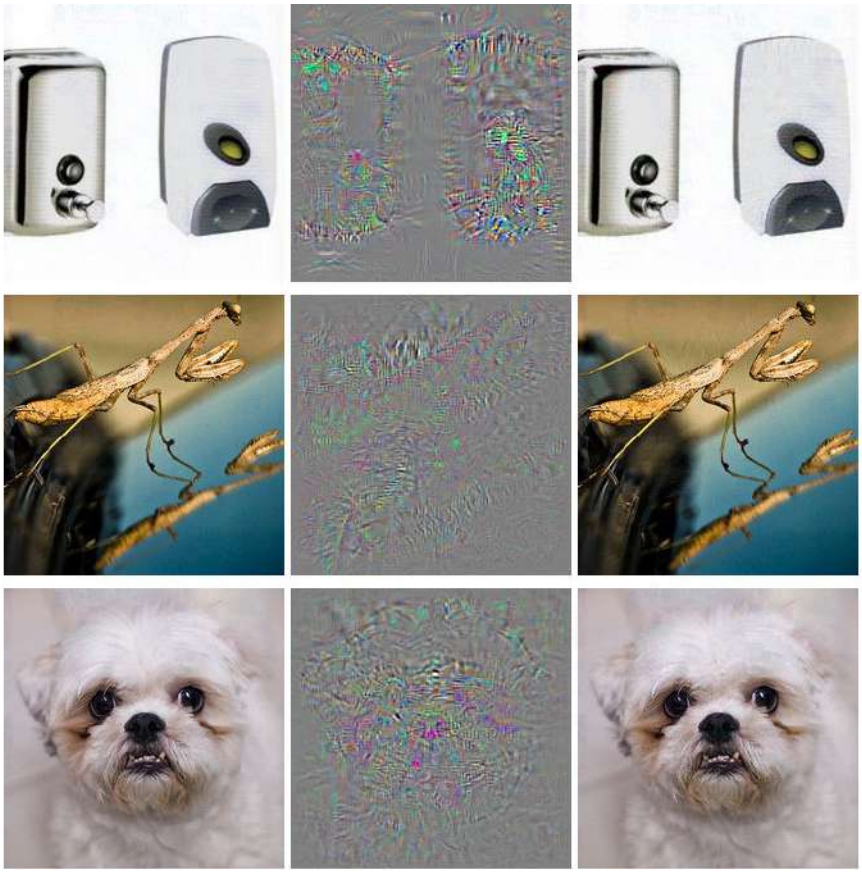
Methods and Examples

There are many techniques to create adversarial examples. Most approaches suggest minimizing the distance between the adversarial example and the instance to be manipulated, while shifting the prediction to the desired (adversarial) outcome. Some methods require access to the gradients of the model, which of course only works with gradient based models such as neural networks, other methods only require access to the prediction function, which makes these methods model-agnostic. The methods in this section focus on image classifiers with deep neural networks, as a lot of research is done in this area and the visualization of adversarial images is very educational. Adversarial examples for images are images with intentionally perturbed pixels with the aim to deceive the model during application time. The examples impressively demonstrate how easily deep neural networks for object recognition can be deceived by images that appear harmless to humans. If you have not yet seen these examples, you might be surprised, because the changes in predictions are incomprehensible for a human observer. Adversarial examples are like optical illusions but for machines.

Something is Wrong With My Dog

Szegedy et. al (2013)[100] used a gradient based optimization approach in their work "Intriguing Properties of Neural Networks" to find adversarial examples for deep neural networks.

[100]Szegedy, Christian, et al. "Intriguing properties of neural networks." arXiv preprint arXiv:1312.6199 (2013).



Adversarial examples for AlexNet by Szegedy et. al (2013). All images in the left column are correctly classified. The middle column shows the (magnified) error added to the images to produce the images in the right column all categorized (incorrectly) as 'Ostrich'.

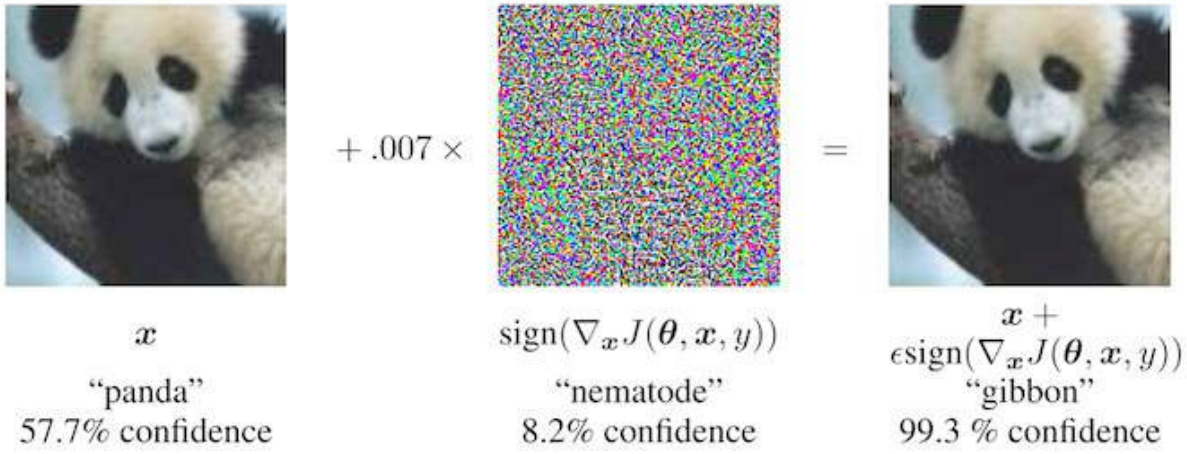
These adversarial examples were generated by minimizing the following function with respect to r:

$$loss(\hat{f}(x+r), l) + c \cdot |r|$$

In this formula, x is an image (represented as a vector of pixels), r is the changes to the pixels to create an adversarial image (x+r produces a new image), l is the desired outcome class, and the parameter c is used to balance the distance between images and the distance between predictions. The first term is the distance between the predicted outcome of the adversarial example and the desired class l, the second term measures the distance between the adversarial example and the original image. This formulation is almost identical to the loss function to generate counterfactual explanations. There are additional constraints for r so that the pixel values remain between 0 and 1. The authors suggest to solve this optimization problem with a box-constrained L-BFGS, an optimization algorithm that works with gradients.

Disturbed panda: Fast gradient sign method

Goodfellow et. al (2014)[101] invented the fast gradient sign method for generating adversarial images. The gradient sign method uses the gradient of the underlying model to find adversarial examples. The original image x is manipulated by adding or subtracting a small error ϵ to each pixel. Whether we add or subtract ϵ depends on whether the sign of the gradient for a pixel is positive or negative. Adding errors in the direction of the gradient means that the image is intentionally altered so that the model classification fails.



Goodfellow et. al (2014) make a panda look like a gibbon for a neural network. By adding small perturbations (middle image) to the original panda pixels (left image), the authors create an adversarial example that is classified as a gibbon (right image) but looks like a panda to humans.

The following formula describes the core of the fast gradient sign method:

$$x' = x + \epsilon \cdot sign(\bigtriangledown_x J(\theta, x, y))$$

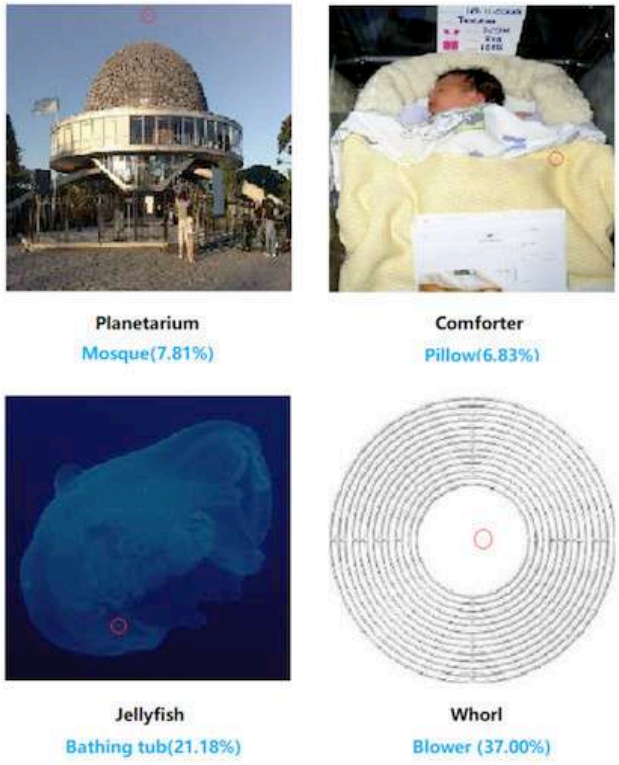
where $\bigtriangledown_x J$ is the gradient of the models loss function with respect to the original input pixel vector x, y is the true label vector for x and θ is the model parameter vector. From the gradient vector (which is as long as the vector of the input pixels) we only need the sign: The sign of the gradient is positive (+1) if an increase in pixel intensity increases the loss (the error the model makes) and negative (-1) if a decrease in pixel intensity increases the loss. This vulnerability occurs when a neural network treats a relationship between an input pixel intensity and the class score linearly. In particular, neural network architectures that favor linearity, such as LSTMs, maxout networks, networks with ReLU activation units or other linear machine learning algorithms such as logistic regression are vulnerable to the gradient sign method. The attack is carried out by extrapolation. The linearity between the input pixel intensity and the class scores leads to vulnerability to outliers, i.e. the model can be deceived by moving pixel values into areas outside the data distribution. I expected these adversarial examples to be quite specific to a given neural network architecture. But it turns out that you can reuse adversarial examples to deceive networks with a different architecture trained on the same task.

[101]Goodfellow, Ian J., Jonathon Shlens, and Christian Szegedy. "Explaining and harnessing adversarial examples." arXiv preprint arXiv:1412.6572 (2014).

Goodfellow et. al (2014) suggested adding adversarial examples to the training data to learn robust models.

A jellyfish … No, wait. A bathtub: 1-pixel attacks

The approach presented by Goodfellow and colleagues (2014) requires many pixels to be changed, if only by a little. But what if you can only change a single pixel? Would you be able to deceive a machine learning model? Su et. al (2019) [102] showed that it is actually possible to deceive image classifiers by changing a single pixel.



By intentionally changing a single pixel (marked with red circles) a neural network trained on ImageNet is deceived to predict the wrong class (in blue letters) instead of the original class (in black letters). Work by Su et. al (2019).

Similar to counterfactuals, the 1-pixel attack looks for a modified example x' which comes close to the original image x, but changes the prediction to an adversarial outcome. However, the definition of closeness differs: Only a single pixel may change. The 1-pixel attack uses differential evolution to find out which pixel is to be changed and how. Differential evolution is loosely inspired by biological evolution of species. A population of individuals called candidate solutions recombines generation by generation until a solution is found. Each candidate solution encodes a pixel modification and is represented by a vector of five elements: the x- and y-coordinates and the red, green and blue (RGB) values. The search starts with, for example, 400 candidate solutions (= pixel modification suggestions)

[102] Su, Jiawei, Danilo Vasconcellos Vargas, and Kouichi Sakurai. "One pixel attack for fooling deep neural networks." IEEE Transactions on Evolutionary Computation (2019).

and creates a new generation of candidate solutions (children) from the parent generation using the following formula:

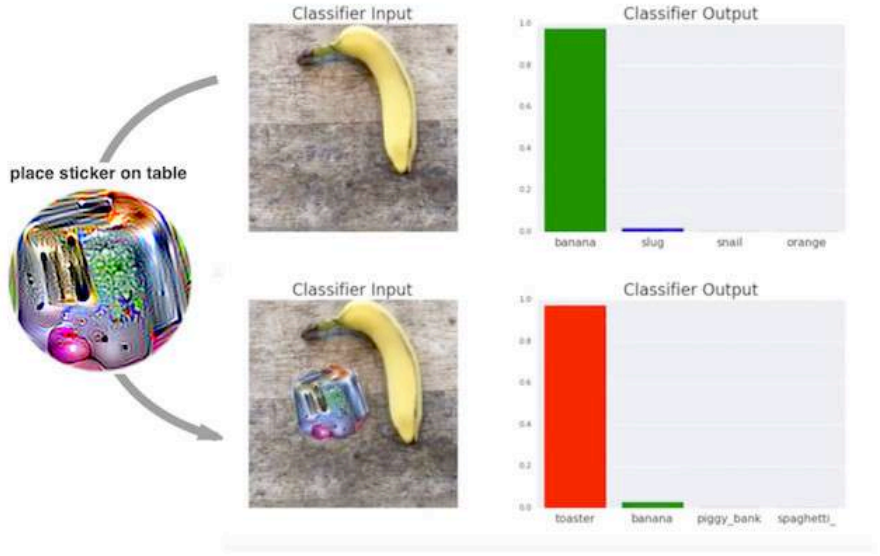
$$x_i(g+1) = x_{r1}(g) + F \cdot (x_{r2}(g) + x_{r3}(g))$$

where each x_i is an element of a candidate solution (either x-coordinate, y-coordinate, red, green or blue), g is the current generation, F is a scaling parameter (set to 0.5) and r1, r2 and r3 are different random numbers. Each new child candidate solution is in turn a pixel with the five attributes for location and color and each of those attributes is a mixture of three random parent pixels.

The creation of children is stopped if one of the candidate solutions is an adversarial example, meaning it is classified as an incorrect class, or if the number of maximum iterations specified by the user is reached.

Everything is a toaster: Adversarial patch

One of my favorite methods brings adversarial examples into physical reality. Brown et. al (2017)[103] designed a printable label that can be stuck next to objects to make them look like toasters for an image classifier. Brilliant work!



A sticker that makes a VGG16 classifier trained on ImageNet categorize an image of a banana as a toaster. Work by Brown et. al (2017).

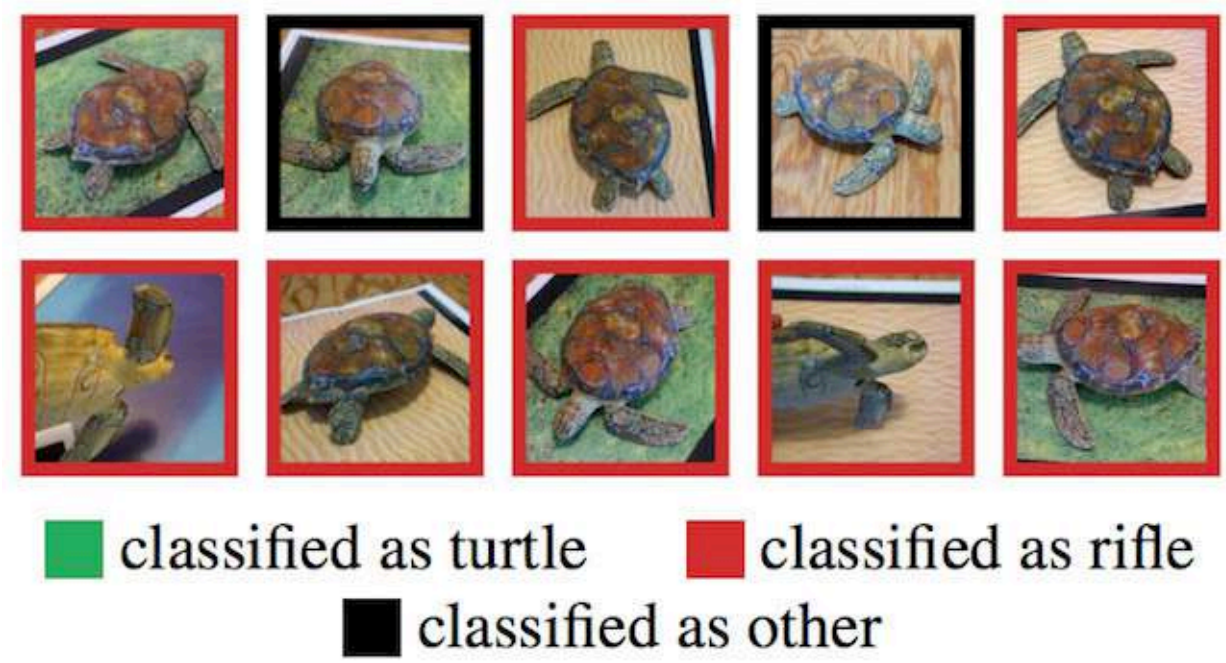
This method differs from the methods presented so far for adversarial examples, since the restriction that the adversarial image must be very close to the original image is removed. Instead, the method completely replaces a part of the image with a patch that can take on any shape. The image of the patch is optimized over different background images, with different positions of the patch on the images, sometimes moved, sometimes larger or smaller and rotated, so that the patch works in many

[103] Brown, Tom B., et al. "Adversarial patch." arXiv preprint arXiv:1712.09665 (2017).

situations. In the end, this optimized image can be printed and used to deceive image classifiers in the wild.

Never bring a 3D-printed turtle to a gunfight – even if your computer thinks it is a good idea: Robust adversarial examples

The next method is literally adding another dimension to the toaster: Athalye et. al (2017)[104] 3D-printed a turtle that was designed to look like a rifle to a deep neural network from almost all possible angles. Yeah, you read that right. A physical object that looks like a turtle to humans looks like a rifle to the computer!



A 3D-printed turtle that is recognized as a rifle by TensorFlowâ€™s standard pre-trained InceptionV3 classifier. Work by Athalye et. al (2017)

The authors have found a way to create an adversarial example in 3D for a 2D classifier that is adversarial over transformations, such as all possibilities to rotate the turtle, zoom in and so on. Other approaches such as the fast gradient method no longer work when the image is rotated or viewing angle changes. Athalye et. al (2017) propose the Expectation Over Transformation (EOT) algorithm, which is a method for generating adversarial examples that even work when the image is transformed. The main idea behind EOT is to optimize adversarial examples across many possible transformations. Instead of minimizing the distance between the adversarial example and the original image, EOT keeps the expected distance between the two below a certain threshold, given a selected distribution of possible transformations. The expected distance under transformation can be written as:

[104]Athalye, Anish, and Ilya Sutskever. "Synthesizing robust adversarial examples." arXiv preprint arXiv:1707.07397 (2017).

$$\mathbb{E}_{t\sim T}[d(t(x'), t(x))]$$

where x is the original image, t(x) the transformed image (e.g. rotated), x' the adversarial example and t(x') its transformed version. Apart from working with a distribution of transformations, the EOT method follows the familiar pattern of framing the search for adversarial examples as an optimization problem. We try to find an adversarial example x' that maximizes the probability for the selected class y_t (e.g. "rifle") across the distribution of possible transformations T:

$$\arg\max_{x'} \mathbb{E}_{t\sim T}[logP(y_t|t(x'))]$$

With the constraint that the expected distance over all possible transformations between adversarial example x' and original image x remains below a certain threshold:

$$\mathbb{E}_{t\sim T}[d(t(x'), t(x))] < \epsilon \quad \text{and} \quad x \in [0,1]^d$$

I think we should be concerned about the possibilities this method enables. The other methods are based on the manipulation of digital images. However, these 3D-printed, robust adversarial examples can be inserted into any real scene and deceive a computer to wrongly classify an object. Let us turn it around: What if someone creates a rifle which looks like a turtle?

The blindfolded adversary: Black box attack

Imagine the following scenario: I give you access to my great image classifier via Web API. You can get predictions from the model, but you do not have access to the model parameters. From the convenience of your couch, you can send data and my service answers with the corresponding classifications. Most adversarial attacks are not designed to work in this scenario because they require access to the gradient of the underlying deep neural network to find adversarial examples. Papernot and colleagues (2017)[105] showed that it is possible to create adversarial examples without internal model information and without access to the training data. This type of (almost) zero-knowledge attack is called black box attack.

How it works:

1. Start with a few images that come from the same domain as the training data, e.g. if the classifier to be attacked is a digit classifier, use images of digits. The knowledge of the domain is required, but not the access to the training data.
2. Get predictions for the current set of images from the black box.
3. Train a surrogate model on the current set of images (for example a neural network).
4. Create a new set of synthetic images using a heuristic that examines for the current set of images in which direction to manipulate the pixels to make the model output have more variance.

[105] Papernot, Nicolas, et al. "Practical black-box attacks against machine learning." Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security. ACM (2017).

5. Repeat steps 2 to 4 for a predefined number of epochs.
6. Create adversarial examples for the surrogate model using the fast gradient method (or similar).
7. Attack the original model with adversarial examples.

The aim of the surrogate model is to approximate the decision boundaries of the black box model, but not necessarily to achieve the same accuracy.

The authors tested this approach by attacking image classifiers trained on various cloud machine learning services. These services train image classifiers on user uploaded images and labels. The software trains the model automatically – sometimes with an algorithm unknown to the user – and deploys it. The classifier then gives predictions for uploaded images, but the model itself cannot be inspected or downloaded. The authors were able to find adversarial examples for various providers, with up to 84% of the adversarial examples being misclassified.

The method even works if the black box model to be deceived is not a neural network. This includes machine learning models without gradients such as a decision trees.

The Cybersecurity Perspective

Machine learning deals with known unknowns: predicting unknown data points from a known distribution. The defense against attacks deals with unknown unknowns: robustly predicting unknown data points from an unknown distribution of adversarial inputs. As machine learning is integrated into more and more systems, such as autonomous vehicles or medical devices, they are also becoming entry points for attacks. Even if the predictions of a machine learning model on a test dataset are 100% correct, adversarial examples can be found to deceive the model. The defense of machine learning models against cyber attacks is a new part of the field of cybersecurity.

Biggio et. al (2018)[106] give a nice review of ten years of research on adversarial machine learning, on which this section is based. Cybersecurity is an arms-race in which attackers and defenders outwit each other time and again.

There are three golden rules in cybersecurity: 1) know your adversary 2) be proactive and 3) protect yourself.

Different applications have different adversaries. People who try to defraud other people via email for their money are adversary agents of users and providers of email services. The providers want to protect their users, so that they can continue using their mail program, the attackers want to get people to give them money. Knowing your adversaries means knowing their goals. Assuming you do not know that these spammers exist and the only abuse of the email service is sending pirated copies of music, then the defense would be different (e.g. scanning the attachments for copyrighted material instead of analyzing the text for spam indicators).

Being proactive means actively testing and identifying weak points of the system. You are proactive when you actively try to deceive the model with adversarial examples and then defend against

[106]Biggio, Battista, and Fabio Roli. "Wild Patterns: Ten years after the rise of adversarial machine learning." Pattern Recognition 84 (2018): 317-331.

them. Using interpretation methods to understand which features are important and how features affect the prediction is also a proactive step in understanding the weaknesses of a machine learning model. As the data scientist, do you trust your model in this dangerous world without ever having looked beyond the predictive power on a test dataset? Have you analyzed how the model behaves in different scenarios, identified the most important inputs, checked the prediction explanations for some examples? Have you tried to find adversarial inputs? The interpretability of machine learning models plays a major role in cybersecurity. Being reactive, the opposite of proactive, means waiting until the system has been attacked and only then understanding the problem and installing some defensive measures.

How can we protect our machine learning systems against adversarial examples? A proactive approach is the iterative retraining of the classifier with adversarial examples, also called adversarial training. Other approaches are based on game theory, such as learning invariant transformations of the features or robust optimization (regularization). Another proposed method is to use multiple classifiers instead of just one and have them vote the prediction (ensemble), but that has no guarantee to work, since they could all suffer from similar adversarial examples. Another approach that does not work well either is gradient masking, which constructs a model without useful gradients by using a nearest neighbor classifier instead of the original model.

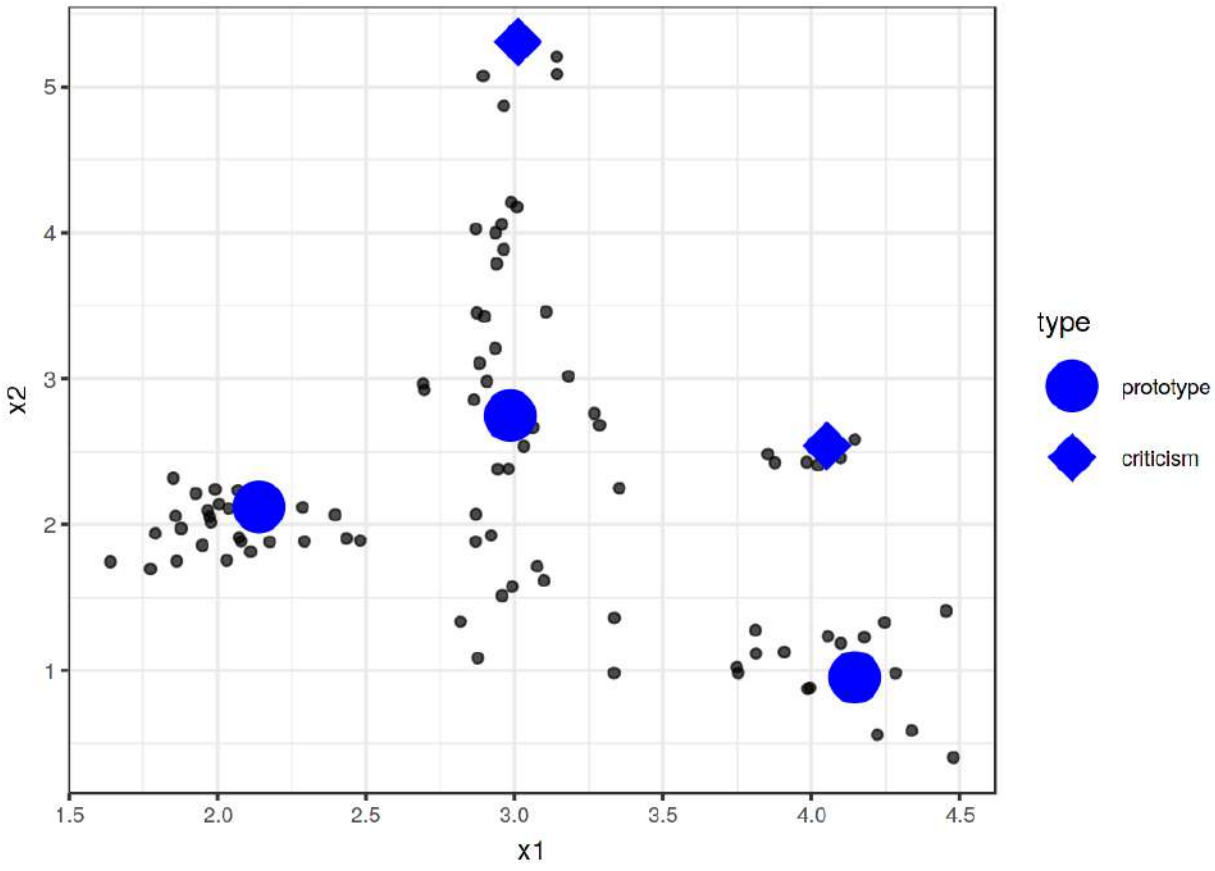
We can distinguish types of attacks by how much an attacker knows about the system. The attackers may have perfect knowledge (white box attack), meaning they know everything about the model like the type of model, the parameters and the training data; the attackers may have partial knowledge (gray box attack), meaning they might only know the feature representation and the type of model that was used, but have no access to the training data or the parameters; the attackers may have zero knowledge (black box attack), meaning they can only query the model in a black box manner but have no access to the training data or information about the model parameters. Depending on the level of information, the attackers can use different techniques to attack the model. As we have seen in the examples, even in the black box case adversarial examples can be created, so that hiding information about data and model is not sufficient to protect against attacks.

Given the nature of the cat-and-mouse game between attackers and defenders, we will see a lot of development and innovation in this area. Just think of the many different types of spam emails that are constantly evolving. New methods of attacks against machine learning models are invented and new defensive measures are proposed against these new attacks. More powerful attacks are developed to evade the latest defenses and so on, ad infinitum. With this chapter I hope to sensitize you to the problem of adversarial examples and that only by proactively studying the machine learning models are we able to discover and remedy weaknesses.

Prototypes and Criticisms

A **prototype** is a data instance that is representative of all the data. A **criticism** is a data instance that is not well represented by the set of prototypes. The purpose of criticisms is to provide insights together with prototypes, especially for data points which the prototypes do not represent well. Prototypes and criticisms can be used independently from a machine learning model to describe the data, but they can also be used to create an interpretable model or to make a black box model interpretable.

In this chapter I use the expression "data point" to refer to a single instance, to emphasize the interpretation that an instance is also a point in a coordinate system where each feature is a dimension. The following figure shows a simulated data distribution, with some of the instances chosen as prototypes and some as criticisms. The small points are the data, the large points the prototypes and the large squares the criticisms. The prototypes are selected (manually) to cover the centers of the data distribution and the criticisms are points in a cluster without a prototype. Prototypes and criticisms are always actual instances from the data.



Prototypes and criticisms for a data distribution with two features x1 and x2.

I selected the prototypes manually, which does not scale well and probably leads to poor results.

There are many approaches to find prototypes in the data. One of these is k-medoids, a clustering algorithm related to the k-means algorithm. Any clustering algorithm that returns actual data points as cluster centers would qualify for selecting prototypes. But most of these methods find only prototypes, but no criticisms. This chapter presents MMD-critic by Kim et. al (2016)[107], an approach that combines prototypes and criticisms in a single framework.

MMD-critic compares the distribution of the data and the distribution of the selected prototypes. This is the central concept for understanding the MMD-critic method. MMD-critic selects prototypes that minimize the discrepancy between the two distributions. Data points in areas with high density are good prototypes, especially when points are selected from different "data clusters". Data points from regions that are not well explained by the prototypes are selected as criticisms.

Let us delve deeper into the theory.

Theory

The MMD-critic procedure on a high-level can be summarized briefly:

1. Select the number of prototypes and criticisms you want to find.
2. Find prototypes with greedy search. Prototypes are selected so that the distribution of the prototypes is close to the data distribution.
3. Find criticisms with greedy search. Points are selected as criticisms where the distribution of prototypes differs from the distribution of the data.

We need a couple of ingredients to find prototypes and criticisms for a dataset with MMD-critic. As the most basic ingredient, we need a **kernel function** to estimate the data densities. A kernel is a function that weighs two data points according to their proximity. Based on density estimates, we need a measure that tells us how different two distributions are so that we can determine whether the distribution of the prototypes we select is close to the data distribution. This is solved by measuring the **maximum mean discrepancy (MMD)**. Also based on the kernel function, we need the **witness function** to tell us how different two distributions are at a particular data point. With the witness function, we can select criticisms, i.e. data points at which the distribution of prototypes and data diverges and the witness function takes on large absolute values. The last ingredient is a search strategy for good prototypes and criticisms, which is solved with a simple **greedy search**.

Let us start with the **maximum mean discrepancy (MMD)**, which measures the discrepancy between two distributions. The selection of prototypes creates a density distribution of prototypes. We want to evaluate whether the prototypes distribution differs from the data distribution. We estimate both with kernel density functions. The maximum mean discrepancy measures the difference between two distributions, which is the supremum over a function space of differences between the expectations according to the two distributions. All clear? Personally, I understand these

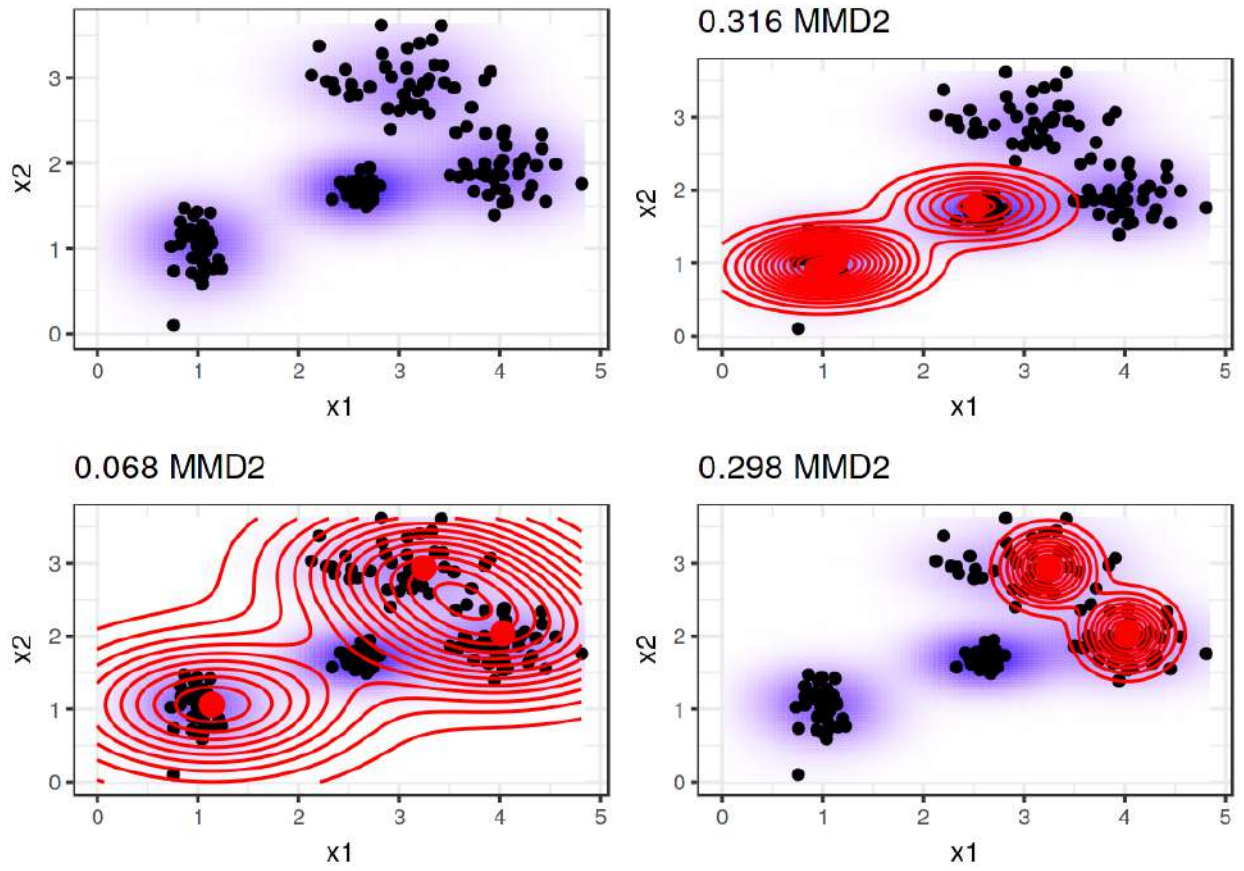
[107]Kim, Been, Rajiv Khanna, and Oluwasanmi O. Koyejo. "Examples are not enough, learn to criticize! Criticism for interpretability." Advances in Neural Information Processing Systems (2016).

concepts much better when I see how something is calculated with data. The following formula shows how to calculate the squared MMD measure (MMD2):

$$MMD^2 = \frac{1}{m^2}\sum_{i,j=1}^{m} k(z_i, z_j) - \frac{2}{mn}\sum_{i,j=1}^{m,n} k(z_i, x_j) + \frac{1}{n^2}\sum_{i,j=1}^{n} k(x_i, x_j)$$

k is a kernel function that measures the similarity of two points, but more about this later. m is the number of prototypes z, and n is the number of data points x in our original dataset. The prototypes z are a selection of data points x. Each point is multidimensional, that is it can have multiple features. The goal of MMD-critic is to minimize MMD2. The closer MMD2 is to zero, the better the distribution of the prototypes fits the data. The key to bringing MMD2 down to zero is the term in the middle, which calculates the average proximity between the prototypes and all other data points (multiplied by 2). If this term adds up to the first term (the average proximity of the prototypes to each other) plus the last term (the average proximity of the data points to each other), then the prototypes explain the data perfectly. Try out what would happen to the formula if you used all n data points as prototypes.

The following graphic illustrates the MMD2 measure. The first plot shows the data points with two features, whereby the estimation of the data density is displayed with a shaded background. Each of the other plots shows different selections of prototypes, along with the MMD2 measure in the plot titles. The prototypes are the large red dots and their distribution is shown as contour lines. The selection of the prototypes that best covers the data in these scenarios (bottom left) has the lowest discrepancy value.



The squared maximum mean discrepancy measure (MMD2) for a dataset with two features and different selections of prototypes.

A choice for the kernel is the radial basis function kernel:

$$k(x, x') = exp\left(\gamma||x - x'||^2\right)$$

where ||x-x'||² is the Euclidean distance between two points and γ is a scaling parameter. The value of the kernel decreases with the distance between the two points and ranges between zero and one: Zero when the two points are infinitely far apart; one when the two points are equal.

We combine the MMD2 measure, the kernel and greedy search in an algorithm for finding prototypes:

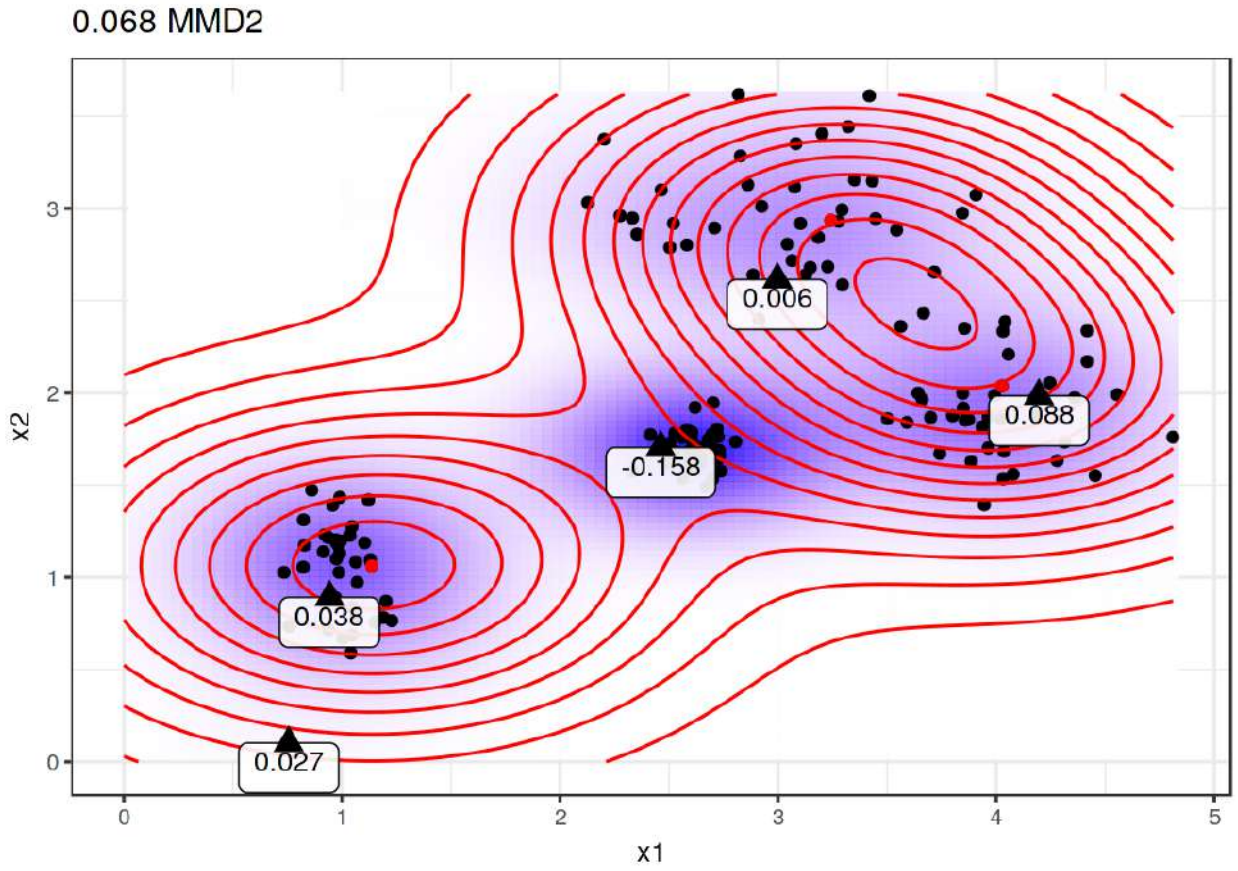
- Start with an empty list of prototypes.
- While the number of prototypes is below the chosen number m:
  - For each point in the dataset, check how much MMD2 is reduced when the point is added to the list of prototypes. Add the data point that minimizes the MMD2 to the list.
- Return the list of prototypes.

The remaining ingredient for finding criticisms is the witness function, which tells us how much two density estimates differ at a particular point. It can be estimated using:

$$witness(x) = \frac{1}{n}\sum_{i=1}^{n} k(x, x_i) - \frac{1}{m}\sum_{j=1}^{m} k(x, z_j)$$

For two datasets (with the same features), the witness function gives you the means of evaluating in which empirical distribution the point x fits better. To find criticisms, we look for extreme values of the witness function in both negative and positive directions. The first term in the witness function is the average proximity between point x and the prototypes, and, respectively, the second term is the average proximity between point x and the data. If the witness function for a point x is close to zero, the density function of the data and the prototypes are close together, which means that the distribution of prototypes resembles the distribution of the data at point x. A positive witness function at point x means that the prototype distribution overestimates the data distribution (for example if we select a prototype but there are only few data points nearby); a negative witness function at point x means that the prototype distribution underestimates the data distribution (for example if there are many data points around x but we have not selected any prototypes nearby).

To give you more intuition, let us reuse the prototypes from the plot beforehand with the lowest MMD2 and display the witness function for a few manually selected points. The labels in the following plot show the value of the witness function for various points marked as squares. Only the point in the middle has a high absolute value and is therefore a good candidate for a criticism.



Evaluations of the witness function at different points.

The witness function allows us to explicitly search for data instances that are not well represented by the prototypes. Criticisms are points with high absolute value in the witness function. Like prototypes, criticisms are also found through greedy search. But instead of reducing the overall MMD2, we are looking for points that maximize a cost function that includes the witness function and a regularizer term. The additional term in the optimization function enforces diversity in the points, which is needed so that the points come from different clusters.

This second step is independent of how the prototypes are found. I could also have handpicked some prototypes and used the procedure described here to learn criticisms. Or the prototypes could come from any clustering procedure, like k-medoids.

That is it with the important parts of MMD-critic theory. One question remains: **How can MMD-critic be used for interpretable machine learning?**

MMD-critic can add interpretability in three ways: By helping to better understand the data distribution; by building an interpretable model; by making a black box model interpretable.

If you apply MMD-critic to your data to find prototypes and criticisms, it will improve your understanding of the data, especially if you have a complex data distribution with edge cases. But with MMD-critic you can achieve more!

For example, you can create an interpretable prediction model: a so-called "nearest prototype model". The prediction function is defined as:

$$\hat{f}(x) = argmax_{i \in S} k(x, x_i)$$

which means that we select the prototype i from the set of prototypes S that is closest to the new data point, in the sense that it yields the highest value of the kernel function. The prototype itself is returned as an explanation for the prediction. This procedure has three tuning parameters: The type of kernel, the kernel scaling parameter and the number of prototypes. All parameters can be optimized within a cross validation loop. The criticisms are not used in this approach.

As a third option, we can use MMD-critic to make any machine learning model globally explainable by examining prototypes and criticisms along with their model predictions. The procedure is as follows:

1. Find prototypes and criticisms with MMD-critic.
2. Train a machine learning model as usual.
3. Predict outcomes for the prototypes and criticisms with the machine learning model.
4. Analyse the predictions: In which cases was the algorithm wrong? Now you have a number of examples that represent the data well and help you to find the weaknesses of the machine learning model.

How does that help? Remember when Google's image classifier identified black people as gorillas? Perhaps they should have used the procedure described here before deploying their image recognition model. It is not enough just to check the performance of the model, because if it were 99% correct, this issue could still be in the 1%. And labels can also be wrong! Going through all the training data and performing a sanity check if the prediction is problematic might have revealed the problem, but would be infeasible. But the selection of – say a few thousand – prototypes and criticisms is feasible and could have revealed a problem with the data: It might have shown that there is a lack of images of people with dark skin, which indicates a problem with the diversity in the dataset. Or it could have shown one or more images of a person with dark skin as a prototype or (probably) as a criticism with the notorious "gorilla" classification. I do not promise that MMD-critic would certainly intercept these kind of mistakes, but it is a good sanity check.

Examples

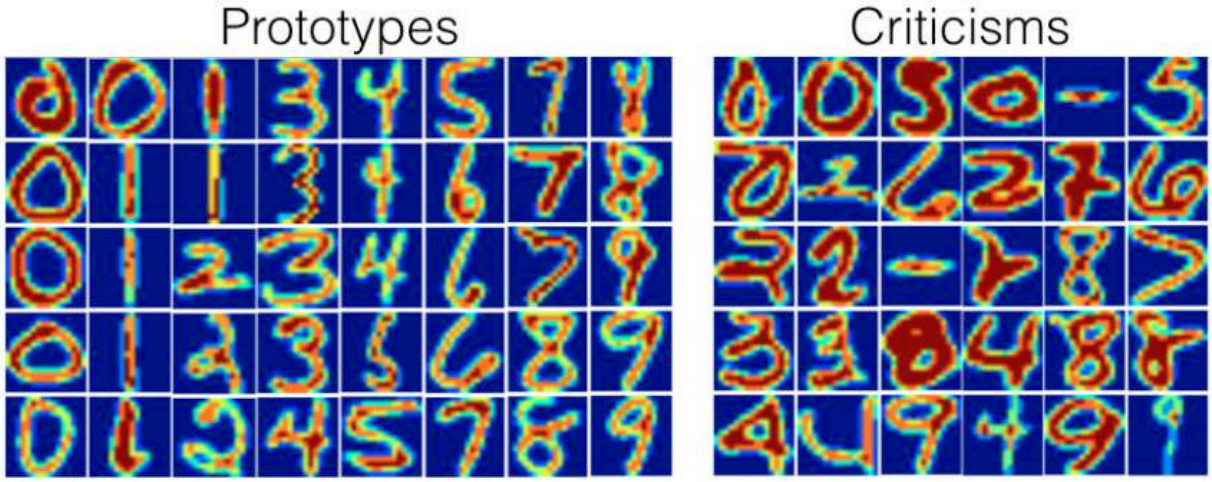
I have taken the examples from the MMD-critic paper. Both applications are based on image datasets. Each image was represented by image embeddings with 2048 dimensions. An image embedding is a vector with numbers which capture abstract attributes of an image. Embedding vectors are usually extracted from neural networks which are trained to solve an image recognition task, in this case the ImageNet challenge. The kernel distances between the images were calculated using these embedding vectors.

The first dataset contains different dog breeds from the ImageNet dataset. MMD-critic is applied on data from two dog breed classes. With the dogs on the left, the prototypes usually show the face of the dog, while the criticisms are images without the dog faces or in different colors (like black and white). On the right side, the prototypes contain outdoor images of dogs. The criticisms contain dogs in costumes and other unusual cases.

Prototypes and criticisms for two types of dog breeds from the ImageNet dataset.

Another illustration of MMD-critic uses a handwritten digit dataset.

Looking at the actual prototypes and criticisms, you might notice that the number of images per digit is different. This is because a fixed number of prototypes and criticisms were searched across the entire dataset and not with a fixed number per class. As expected, the prototypes show different ways of writing the digits. The criticisms include examples with unusually thick or thin lines, but also unrecognizable digits.



Prototypes and criticisms for a handwritten digits dataset.

Advantages

In a user study the authors of MMD-critic gave images to the participants, which they had to visually match to one of two sets of images, each representing one of two classes (e.g. two dog breeds). The **participants performed best when the sets showed prototypes and criticisms** instead of random images of a class.

You are free to **choose the number of prototypes and criticisms**.

MMD-critic works with density estimates of the data. This **works with any type of data and any type of machine learning model**.

The algorithm is **easy to implement**.

MMD-critic is very flexible in the way it is used to increase interpretability. It can be used to understand complex data distributions. It can be used to build an interpretable machine learning model. Or it can shed light on the decision making of a black box machine learning model.

Finding criticisms is independent of the selection process of the prototypes. But it makes sense to select prototypes according to MMD-critic, because then both prototypes and criticisms are created using the same method of comparing prototypes and data densities.

Disadvantages

You have to **choose the number of prototypes and criticisms**. As much as this can be nice-to-have, it is also a disadvantage. How many prototypes and criticisms do we actually need? The more the better? The less the better? One solution is to select the number of prototypes and criticisms by measuring how much time humans have for the task of looking at the images, which depends on the particular application. Only when using MMD-critic to build a classifier do we have a way to optimize it directly. One solution could be a screeplot showing the number of prototypes on the x-axis and the MMD2 measure on the y-axis. We would choose the number of prototypes where the MMD2 curve flattens.

The other parameters are the choice of the kernel and the kernel scaling parameter. We have the same problem as with the number of prototypes and criticisms: **How do we select a kernel and its scaling parameter?** Again, when we use MMD-critic as a nearest prototype classifier, we can tune the kernel parameters. For the unsupervised use cases of MMD-critic, however, it is unclear. (Maybe I am a bit harsh here, since all unsupervised methods have this problem.)

It takes all the features as input, **disregarding the fact that some features might not be relevant** for predicting the outcome of interest. One solution is to use only relevant features, for example image embeddings instead of raw pixels. This works as long as we have a way to project the original instance onto a representation that contains only relevant information.

There is some code available, but it is **not yet implemented as nicely packaged and documented software**.

Code and Alternatives

An implementation of MMD-critic can be found here: https://github.com/BeenKim/MMD-critic[108].

The simplest alternative to finding prototypes is k-medoids[109] by Kaufman et. al (1987).[110]

[108]https://github.com/BeenKim/MMD-critic
[109]https://en.wikipedia.org/wiki/K-medoids
[110]Kaufman, Leonard, and Peter Rousseeuw. "Clustering by means of medoids". North-Holland (1987).

Influential Instances

Machine learning models are ultimately a product of training data and deleting one of the training instances can affect the resulting model. We call a training instance "influential" when its deletion from the training data considerably changes the parameters or predictions of the model. By identifying influential training instances, we can "debug" machine learning models and better explain their behaviors and predictions.

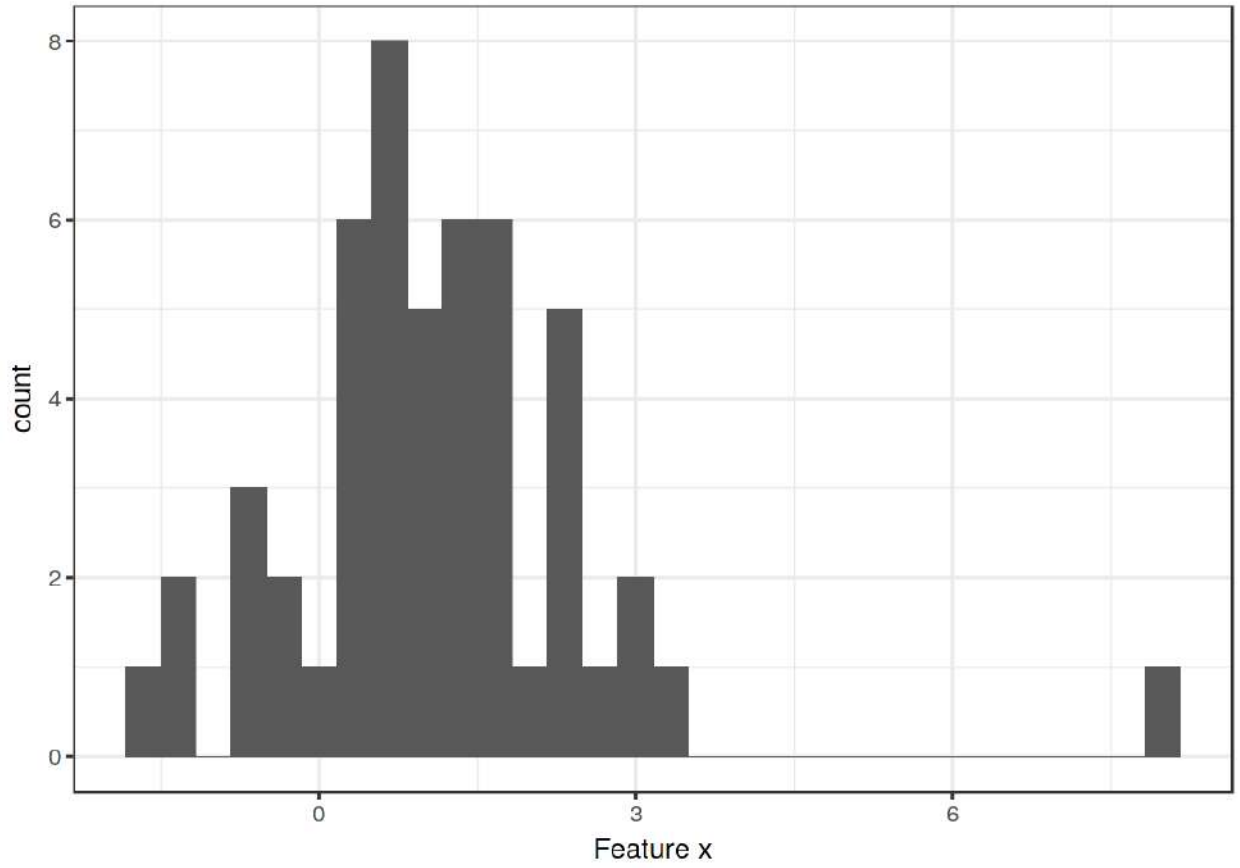
This chapter shows you two approaches for identifying influential instances, namely deletion diagnostics and influence functions. Both approaches are based on robust statistics, which provides statistical methods that are less affected by outliers or violations of model assumptions. Robust statistics also provides methods to measure how robust estimates from data are (such as a mean estimate or the weights of a prediction model).

Imagine you want to estimate the average income of the people in your city and ask ten random people on the street how much they earn. Apart from the fact that your sample is probably really bad, how much can your average income estimate be influenced by a single person? To answer this question, you can recalculate the mean value by omitting individual answers or derive mathematically via "influence functions" how the mean value can be influenced. With the deletion approach, we recalculate the mean value ten times, omitting one of the income statements each time, and measure how much the mean estimate changes. A big change means that an instance was very influential. The second approach upweights one of the persons by an infinitesimally small weight, which corresponds to the calculation of the first derivative of a statistic or model. This approach is also known as "infinitesimal approach" or "influence function". The answer is, by the way, that your mean estimate can be very strongly influenced by a single answer, since the mean scales linearly with single values. A more robust choice is the median (the value at which one half of people earn more and the other half less), because even if the person with the highest income in your sample would earn ten times more, the resulting median would not change.

Deletion diagnostics and influence functions can also be applied to the parameters or predictions of machine learning models to understand their behavior better or to explain individual predictions. Before we look at these two approaches for finding influential instances, we will examine the difference between an outlier and an influential instance.

Outlier

An outlier is an instance that is far away from the other instances in the dataset. "Far away" means that the distance, for example the Euclidean distance, to all the other instances is very large. In a dataset of newborns, a newborn weighting 6 kg would be considered an outlier. In a dataset of bank accounts with mostly checking accounts, a dedicated loan account (large negative balance, few transactions) would be considered an outlier. The following figure shows an outlier for a 1-dimensional distribution.

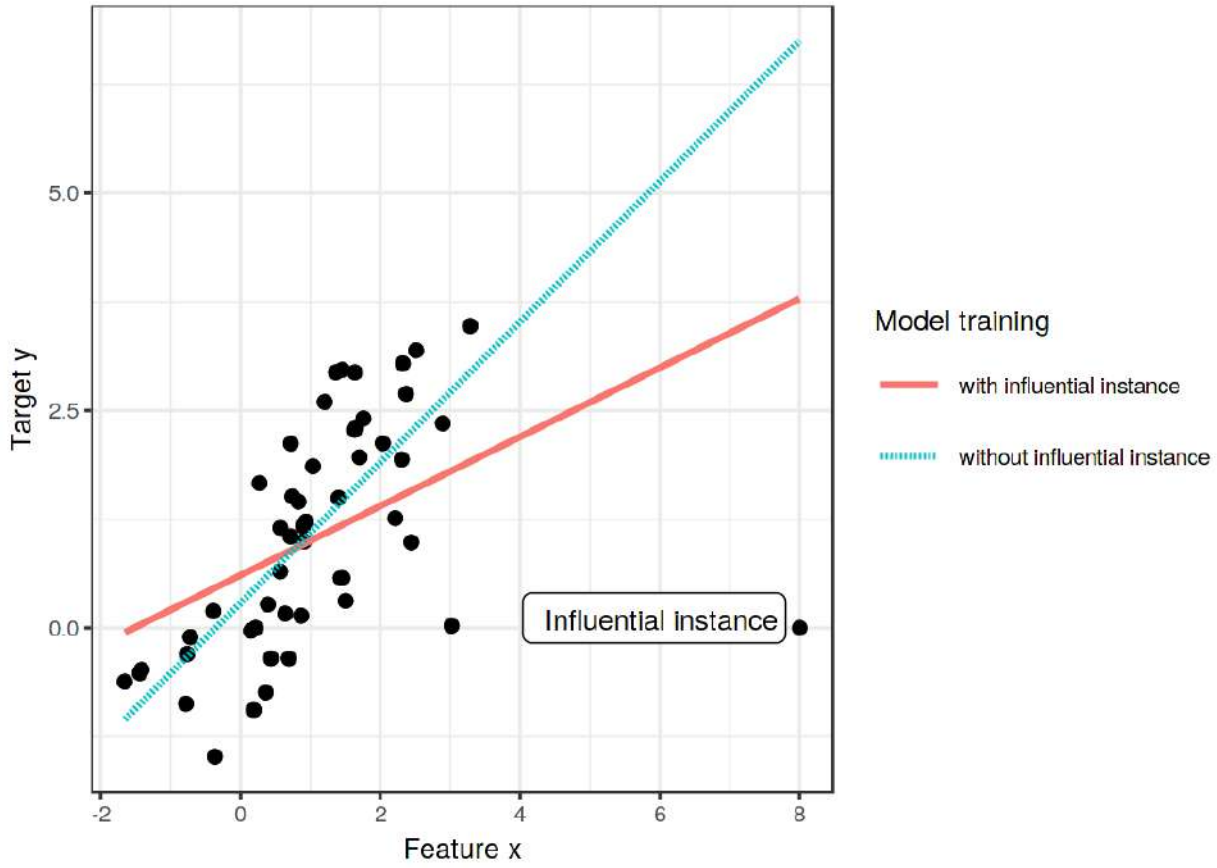


Feature x follows a Gaussian distribution with an outlier at x=8.

Outliers can be interesting data points (e.g. criticisms). When an outlier influences the model it is also an influential instance.

Influential instance

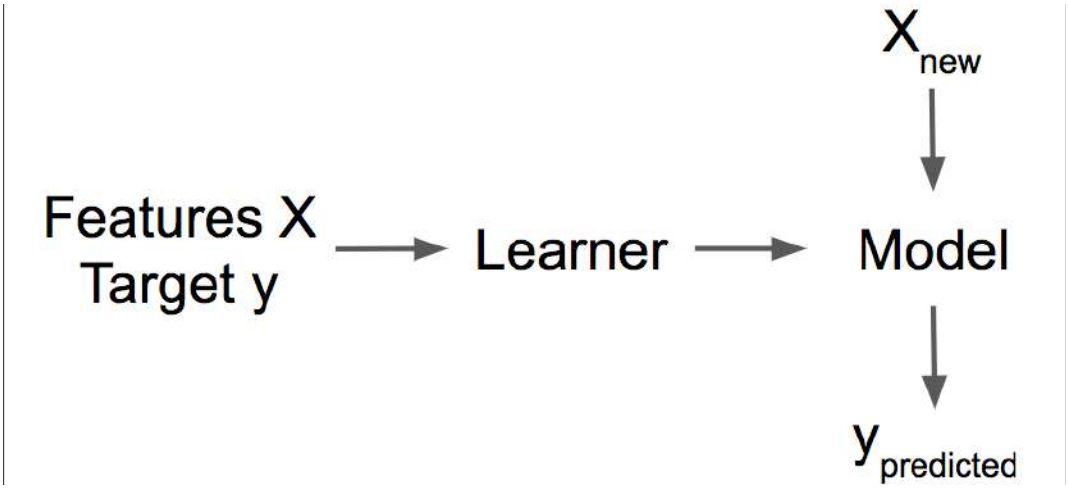
An influential instance is a data instance whose removal has a strong effect on the trained model. The more the model parameters or predictions change when the model is retrained with a particular instance removed from the training data, the more influential that instance is. Whether an instance is influential for a trained model also depends on its value for the target y. The following figure shows an influential instance for a linear regression model.



A linear model with one feature. Trained once on the full data and once without the influential instance. Removing the influential instance changes the fitted slope (weight/coefficient) drastically.

Why do influential instances help to understand the model?

The key idea behind influential instances for interpretability is to trace model parameters and predictions back to where it all began: the training data. A learner, that is, the algorithm that generates the machine learning model, is a function that takes training data consisting of features X and target y and generates a machine learning model. For example, the learner of a decision tree is an algorithm that selects the split features and the values at which to split. A learner for a neural network uses backpropagation to find the best weights.



A learner learns a model from training data (features plus target). The model makes predictions for new data.

We ask how the model parameters or the predictions would change if we removed instances from the training data in the training process. This is in contrast to other interpretability approaches that analyze how the prediction changes when we manipulate the features of the instances to be predicted, such as partial dependence plots or feature importance. With influential instances, we do not treat the model as fixed, but as a function of the training data. Influential instances help us answer questions about global model behavior and about individual predictions. Which were the most influential instances for the model parameters or the predictions overall? Which were the most influential instances for a particular prediction? Influential instances tell us for which instances the model could have problems, which training instances should be checked for errors and give an impression of the robustness of the model. We might not trust a model if a single instance has a strong influence on the model predictions and parameters. At least that would make us investigate further.

How can we find influential instances? We have two ways of measuring influence: Our first option is to delete the instance from the training data, retrain the model on the reduced training dataset and observe the difference in the model parameters or predictions (either individually or over the complete dataset). The second option is to upweight a data instance by approximating the parameter changes based on the gradients of the model parameters. The deletion approach is easier to understand and motivates the upweighting approach, so we start with the former.

Deletion Diagnostics

Statisticians have already done a lot of research in the area of influential instances, especially for (generalized) linear regression models. When you search for "influential observations", the first search results are about measures like DFBETA and Cook's distance. **DFBETA** measures the effect of deleting an instance on the model parameters. **Cook's distance** (Cook, 1977[111]) measures the effect of deleting an instance on model predictions. For both measures we have to retrain the model

[111]Cook, R. Dennis. "Detection of influential observation in linear regression." Technometrics 19.1 (1977): 15-18.

repeatedly, omitting individual instances each time. The parameters or predictions of the model with all instances is compared with the parameters or predictions of the model with one of the instances deleted from the training data.

DFBETA is defined as:

$$DFBETA_i = \beta - \beta^{(-i)}$$

where β is the weight vector when the model is trained on all data instances, and $\beta^{(-i)}$ the weight vector when the model is trained without instance i. Quite intuitive I would say. DFBETA works only for models with weight parameters, such as logistic regression or neural networks, but not for models such as decision trees, tree ensembles, some support vector machines and so on.

Cook's distance was invented for linear regression models and approximations for generalized linear regression models exist. Cook's distance for a training instance is defined as the (scaled) sum of the squared differences in the predicted outcome when the i-th instance is removed from the model training.

$$D_i = \frac{\sum_{i=1}^{n}(\hat{y}_j - \hat{y}_j^{(-i)})^2}{p \cdot MSE}$$

where the numerator is the squared difference between prediction of the model with and without the i-th instance, summed over the dataset. The denominator is the number of features p times the mean squared error. The denominator is the same for all instances no matter which instance i is removed. Cook's distance tells us how much the predicted output of a linear model changes when we remove the i-th instance from the training.

Can we use Cook's distance and DFBETA for any machine learning model? DFBETA requires model parameters, so this measure works only for parameterized models. Cook's distance does not require any model parameters. Interestingly, Cook's distance is usually not seen outside the context of linear models and generalized linear models, but the idea of taking the difference between model predictions before and after removal of a particular instance is very general. A problem with the definition of Cook's distance is the MSE, which is not meaningful for all types of prediction models (e.g. classification).

The simplest influence measure for the effect on the model predictions can be written as follows:

$$\text{Influence}^{(-i)} = \frac{1}{n}\sum_{j=1}^{n}\left|\hat{y}_j - \hat{y}_j^{(-i)}\right|$$

This expression is basically the numerator of Cook's distance, with the difference that the absolute difference is added up instead of the squared differences. This was a choice I made, because it makes sense for the examples later. The general form of deletion diagnostic measures consists of choosing a measure (such as the predicted outcome) and calculating the difference of the measure for the model trained on all instances and when the instance is deleted.

We can easily break the influence down to explain for the prediction of instance j what the influence of the i-th training instance was:

$$\text{Influence}_j^{(-i)} = \left|\hat{y}_j - \hat{y}_j^{(-i)}\right|$$

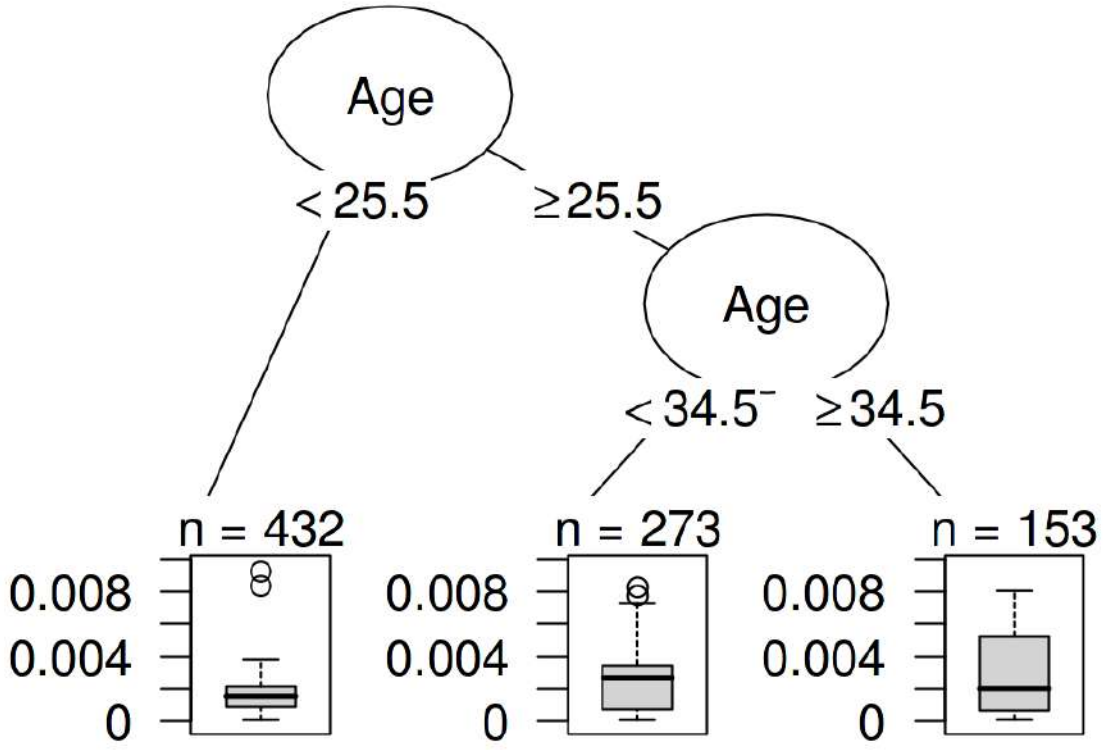
This would also work for the difference in model parameters or the difference in the loss. In the following example we will use these simple influence measures.

Deletion diagnostics example

In the following example, we train a support vector machine to predict cervical cancer given risk factors and measure which training instances were most influential overall and for a particular prediction. Since the prediction of cancer is a classification problem, we measure the influence as the difference in predicted probability for cancer. An instance is influential if the predicted probability strongly increases or decreases on average in the dataset when the instance is removed from model training. The measurement of the influence for all 858 training instances requires to train the model once on all data and retrain it 858 times (= size of training data) with one of the instances removed each time.

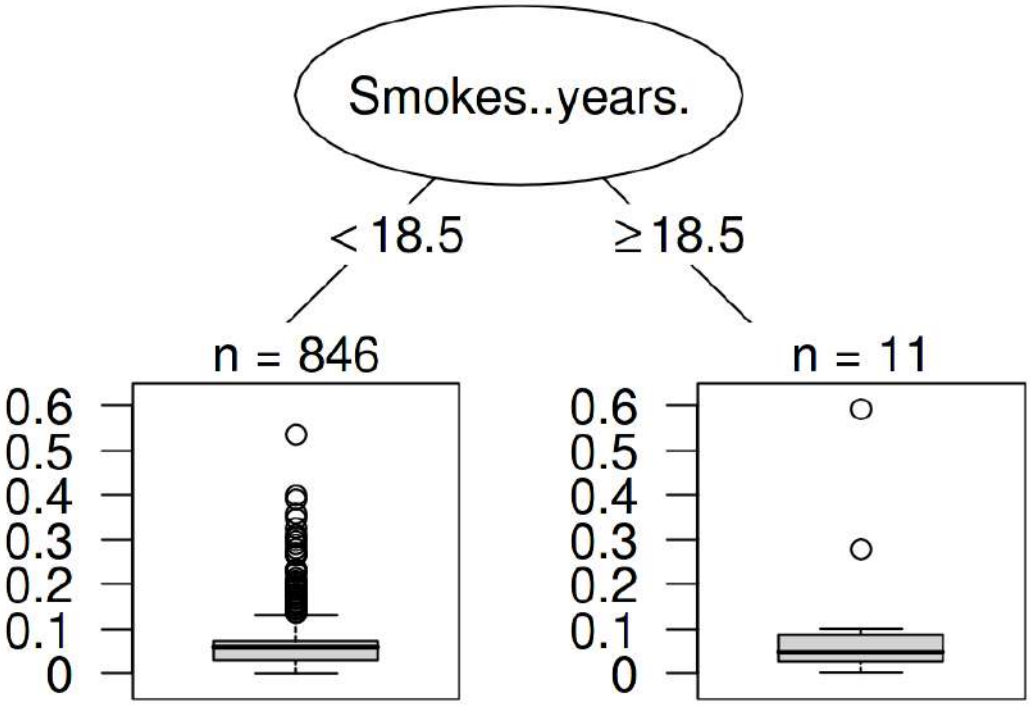
The most influential instance has an influence measure of about 0.01. An influence of 0.01 means that if we remove the 540-th instance, the predicted probability changes by 1 percentage point on average. This is quite substantial considering the average predicted probability for cancer is 6.4%. The mean value of influence measures over all possible deletions is 0.2 percentage points. Now we know which of the data instances were most influential for the model. This is already useful to know for debugging the data. Is there a problematic instance? Are there measurement errors? The influential instances are the first ones that should be checked for errors, because each error in them strongly influences the model predictions.

Apart from model debugging, can we learn something to better understand the model? Just printing out the top 10 most influential instances is not very useful, because it is just a table of instances with many features. All methods that return instances as output only make sense if we have a good way of representing them. But we can better understand what kind of instances are influential when we ask: What distinguishes an influential instance from a non-influential instance? We can turn this question into a regression problem and model the influence of an instance as a function of its feature values. We are free to choose any model from the chapter on Interpretable Machine Learning Models. For this example I chose a decision tree (following figure) that shows that data from women of age 35 and older were the most influential for the support vector machine. Of all the women in the dataset 153 out of 858 were older than 35. In the chapter on Partial Dependence Plots we have seen that after 40 there is a sharp increase in the predicted probability of cancer and the Feature Importance has also detected age as one of the most important features. The influence analysis tells us that the model becomes increasingly unstable when predicting cancer for higher ages. This in itself is valuable information. This means that errors in these instances can have a strong effect on the model.



A decision tree that models the relationship between the influence of the instances and their features. The maximum depth of the tree is set to 2.

This first influence analysis revealed the *overall* most influential instance. Now we select one of the instances, namely the 7-th instance, for which we want to explain the prediction by finding the most influential training data instances. It is like a counterfactual question: How would the prediction for instance 7 change if we omit instance i from the training process? We repeat this removal for all instances. Then we select the training instances that result in the biggest change in the prediction of instance 7 when they are omitted from the training and use them to explain the prediction of the model for that instance. I chose to explain the prediction for instance 7 because it is the instance with the highest predicted probability of cancer (7.35%), which I thought was an interesting case to analyze more deeply. We could return the, say, top 10 most influential instances for predicting the 7-th instance printed as a table. Not very useful, because we could not see much. Again, it makes more sense to find out what distinguishes the influential instances from the non-influential instances by analyzing their features. We use a decision tree trained to predict the influence given the features, but in reality we misuse it only to find a structure and not to actually predict something. The following decision tree shows which kind of training instances where most influential for predicting the 7-th instance.



Decision tree that explains which instances were most influential for predicting the 7-th instance. Data from women who smoked for 18.5 years or longer had a large influence on the prediction of the 7-th instance, with an average change in absolute prediction by 11.7 percentage points of cancer probability.

Data instances of women who smoked or have been smoking for 18.5 years or longer have a high influence on the prediction of the 7-th instance. The woman behind the 7-th instance smoked for 34 years. In the data, 12 women (1.40%) smoked 18.5 years or longer. Any mistake made in collecting the number of years of smoking of one of these women will have a huge impact on the predicted outcome for the 7-th instance.

The most extreme change in the prediction happens when we remove instance number 663. The patient allegedly smoked for 22 years, aligned with the results from the decision tree. The predicted probability for the 7-th instance changes from 7.35% to 66.60% if we remove the instance 663!

If we take a closer look at the features of the most influential instance, we can see another possible problem. The data say that the woman is 28 years old and has been smoking for 22 years. Either it is a really extreme case and she really started smoking at 6, or this is a data error. I tend to believe the latter. This is certainly a situation in which we must question the accuracy of the data.

These examples showed how useful it is to identify influential instances to debug models. One problem with the proposed approach is that the model needs to be retrained for each training instance. The whole retraining can be quite slow, because if you have thousands of training instances,

you will have to retrain your model thousands of times. Assuming the model takes one day to train and you have 1000 training instances, then the computation of influential instances – without parallelization – will take almost 3 years. Nobody has time for this. In the rest of this chapter, I will show you a method that does not require retraining the model.

Influence Functions

You: I want to know the influence a training instance has on a particular prediction.
Research: You can delete the training instance, retrain the model, and measure the difference in the prediction.
You: Great! But do you have a method for me that works without retraining? It takes so much time.
Research: Do you have a model with a loss function that is twice differentiable with respect to its parameters?
You: I trained a neural network with the logistic loss. So yes.
Research: Then you can approximate the influence of the instance on the model parameters and on the prediction with **influence functions**. The influence function is a measure of how strongly the model parameters or predictions depend on a training instance. Instead of deleting the instance, the method upweights the instance in the loss by a very small step. This method involves approximating the loss around the current model parameters using the gradient and Hessian matrix. Loss upweighting is similar to deleting the instance.
You: Great, that's what I'm looking for!

Koh and Liang (2017)[112] suggested using influence functions, a method of robust statistics, to measure how an instance influences model parameters or predictions. As with deletion diagnostics, the influence functions trace the model parameters and predictions back to the responsible training instance. However, instead of deleting training instances, the method approximates how much the model changes when the instance is upweighted in the empirical risk (sum of the loss over the training data).

The method of influence functions requires access to the loss gradient with respect to the model parameters, which only works for a subset of machine learning models. Logistic regression, neural networks and support vector machines qualify, tree-based methods like random forests do not. Influence functions help to understand the model behavior, debug the model and detect errors in the dataset.

The following section explains the intuition and math behind influence functions.

Math behind influence functions

The key idea behind influence functions is to upweight the loss of a training instance by an infinitesimally small step ϵ, which results in new model parameters:

$$\hat{\theta}_{\epsilon,z} = \arg\min_{\theta\in\Theta}(1-\epsilon)\frac{1}{n}\sum_{i=1}^{n}L(z_i,\theta) + \epsilon L(z,\theta)$$

[112]Koh, Pang Wei, and Percy Liang. "Understanding black-box predictions via influence functions." arXiv preprint arXiv:1703.04730 (2017).

where θ is the model parameter vector and $\hat{\theta}_{\epsilon,z}$ is the parameter vector after upweighting z by a very small number ϵ. L is the loss function with which the model was trained, z_i is the training data and z is the training instance which we want to upweight to simulate its removal. The intuition behind this formula is: How much will the loss change if we upweight a particular instance z_i from the training data by a little (ϵ) and downweight the other data instances accordingly? What would the parameter vector look like to optimize this new combined loss? The influence function of the parameters, i.e. the influence of upweighting training instance z on the parameters, can be calculated as follows.

$$I_{\text{up,params}}(z) = \left.\frac{d\hat{\theta}_{\epsilon,z}}{d\epsilon}\right|_{\epsilon=0} = -H_{\hat{\theta}}^{-1}\nabla_{\theta}L(z,\hat{\theta})$$

The last expression $\nabla_{\theta}L(z,\hat{\theta})$ is the loss gradient with respect to the parameters for the upweighted training instance. The gradient is the rate of change of the loss of the training instance. It tells us how much the loss changes when we change the model parameters $\hat{\theta}$ by a bit. A positive entry in the gradient vector means that a small increase in the corresponding model parameter increases the loss, a negative entry means that the increase of the parameter reduces the loss. The first part $H_{\hat{\theta}}^{-1}$ is the inverse Hessian matrix (second derivative of the loss with respect to the model parameters). The Hessian matrix is the rate of change of the gradient, or expressed as loss, it is the rate of change of the rate of change of the loss. It can be estimated using:

$$H_{\theta} = \frac{1}{n}\sum_{i=1}^{n}\nabla_{\hat{\theta}}^{2}L(z_i,\hat{\theta})$$

More informally: The Hessian matrix records how curved the loss is at a certain point. The Hessian is a matrix and not just a vector, because it describes the curvature of the loss and the curvature depends on the direction in which we look. The actual calculation of the Hessian matrix is time-consuming if you have many parameters. Koh and Liang suggested some tricks to calculate it efficiently, which goes beyond the scope of this chapter. Updating the model parameters, as described by the above formula, is equivalent to taking a single Newton step after forming a quadratic expansion around the estimated model parameters.

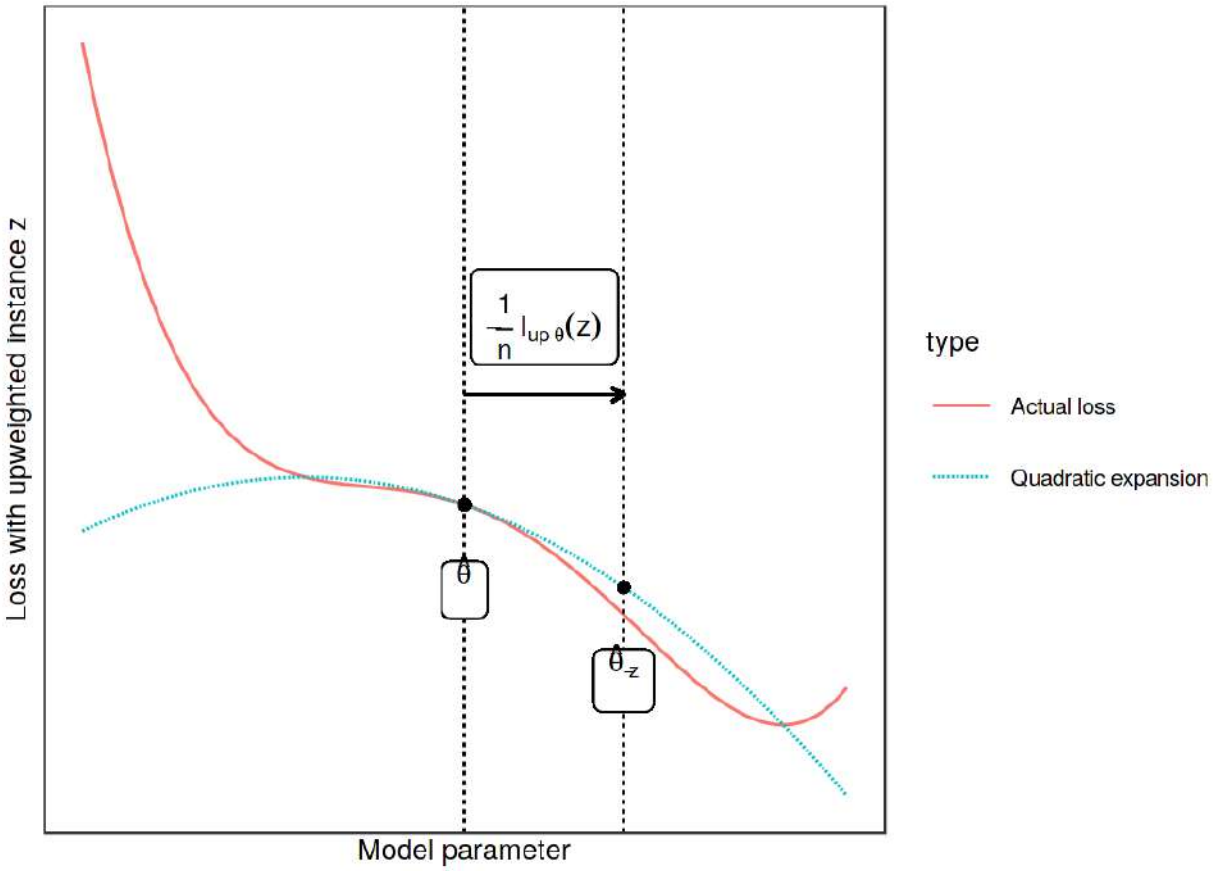
What intuition is behind this influence function formula? The formula comes from forming a quadratic expansion around the parameters $\hat{\theta}$. That means we do not actually know, or it is too complex to calculate how exactly the loss of instance z will change when it is removed/upweighted. We approximate the function locally by using information about the steepness (= gradient) and the curvature (= Hessian matrix) at the current model parameter setting. With this loss approximation, we can calculate what the new parameters would approximately look like if we upweighted instance z:

$$\hat{\theta}_{-z} \approx \hat{\theta} - \frac{1}{n}I_{\text{up,params}}(z)$$

The approximate parameter vector is basically the original parameter minus the gradient of the loss of z (because we want to decrease the loss) scaled by the curvature (= multiplied by the inverse

Hessian matrix) and scaled by 1 over n, because that is the weight of a single training instance.

The following figure shows how the upweighting works. The x-axis shows the value of the θ parameter and the y-axis the corresponding value of the loss with upweighted instance z. The model parameter here is 1-dimensional for demonstration purposes, but in reality it is usually high-dimensional. We move only 1 over n into the direction of improvement of the loss for instance z. We do not know how the loss would really change when we delete z, but with the first and second derivative of the loss, we create this quadratic approximation around our current model parameter and pretend that this is how the real loss would behave.



Updating the model parameter (x-axis) by forming a quadratic expansion of the loss around the current model parameter, and moving 1/n into the direction in which the loss with upweighted instance z (y-axis) improves most. This upweighting of instance z in the loss approximates the parameter changes if we delete z and train the model on the reduced data.

We do not actually need to calculate the new parameters, but we can use the influence function as a measure of the influence of z on the parameters.

How do the *predictions* change when we upweight training instance z? We can either calculate the new parameters and then make predictions using the newly parameterized model, or we can also calculate the influence of instance z on the predictions directly, since we can calculate the influence by using the chain rule:

$$
\begin{aligned}
I_{up,loss}(z, z_{test}) &= \left.\frac{dL(z_{test}, \hat{\theta}_{\epsilon,z})}{d\epsilon}\right|_{\epsilon=0} \\
&= \nabla_\theta L(z_{test}, \hat{\theta})^T \left.\frac{d\hat{\theta}_{\epsilon,z}}{d\epsilon}\right|_{\epsilon=0} \\
&= -\nabla_\theta L(z_{test}, \hat{\theta})^T H_\theta^{-1} \nabla_\theta L(z, \hat{\theta})
\end{aligned}
$$

The first line of this equation means that we measure the influence of a training instance on a certain prediction z_{test} as a change in loss of the test instance when we upweight the instance z and get new parameters $\hat{\theta}_{\epsilon,z}$. For the second line of the equation, we have applied the chain rule of derivatives and get the derivative of the loss of the test instance with respect to the parameters times the influence of z on the parameters. In the third line, we replace the expression with the influence function for the parameters. The first term in the third line $\nabla_\theta L(z_{test}, \hat{\theta})^T$ is the gradient of the test instance with respect to the model parameters.

Having a formula is great and the scientific and accurate way of showing things. But I think it is very important to get some intuition about what the formula means. The formula for $I_{\text{up,loss}}$ states that the influence function of the training instance z on the prediction of an instance z_{test} is "how strongly the instance reacts to a change of the model parameters" multiplied by "how much the parameters change when we upweight the instance z". Another way to read the formula: The influence is proportional to how large the gradients for the training and test loss are. The higher the gradient of the training loss, the higher its influence on the parameters and the higher the influence on the test prediction. The higher the gradient of the test prediction, the more influenceable the test instance. The entire construct can also be seen as a measure of the similarity (as learned by the model) between the training and the test instance.

That is it with theory and intuition. The next section explains how influence functions can be applied.

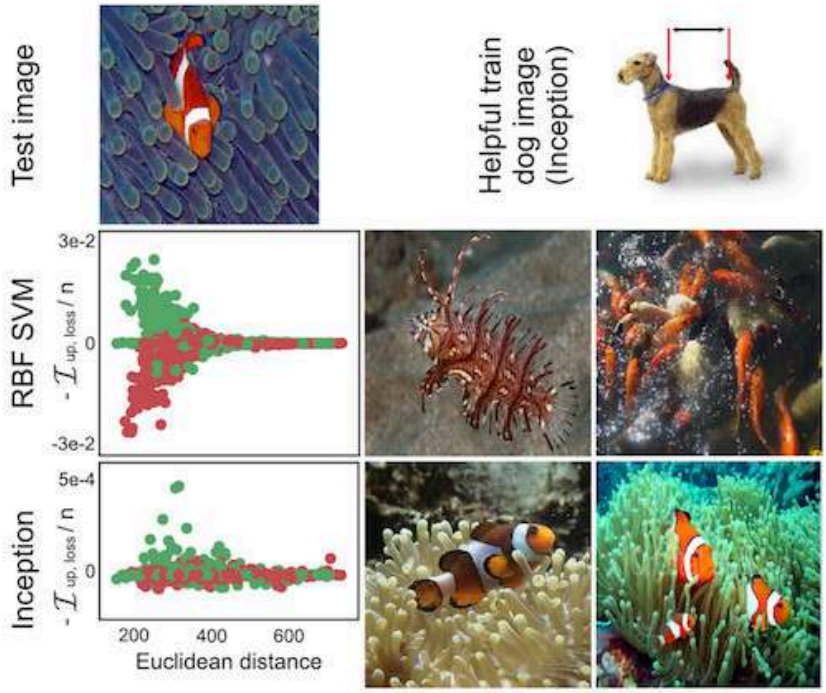
Application of Influence Functions

Influence functions have many applications, some of which have already been presented in this chapter.

Understanding model behavior

Different machine learning models have different ways of making predictions. Even if two models have the same performance, the way they make predictions from the features can be very different and therefore fail in different scenarios. Understanding the particular weaknesses of a model by identifying influential instances helps to form a "mental model" of the machine learning model behavior in your mind. The following figure shows an example where a support vector machine (SVM) and a neural network were trained to distinguish images of dogs and fish. The most influential instances of an exemplary image of a fish were very different for both models. For the SVM, instances were influential if they were similar in color. For the neural network, instances were influential if they were conceptually similar. For the neural network, even one image of a dog was among the most influential images, showing that it learned the concepts and not the Euclidean distance in color

space.



Dog or fish? For the SVM prediction (middle row) images that had similar colors as the test image were the most influential. For the neural network prediction (bottom row) fish in different setting were most influential, but also a dog image (top right). Work by Koh and Liang (2017).

Handling domain mismatches / Debugging model errors

Handling domain mismatch is closely related to better understand the model behavior. Domain mismatch means that the distribution of training and test data is different, which can cause the model to perform poorly on the test data. Influence functions can identify training instances that caused the error. Suppose you have trained a prediction model for the outcome of patients who have undergone surgery. All these patients come from the same hospital. Now you use the model in another hospital and see that it does not work well for many patients. Of course, you assume that the two hospitals have different patients, and if you look at their data, you can see that they differ in many features. But what are the features or instances that have "broken" the model? Here too, influential instances are a good way to answer this question. You take one of the new patients, for whom the model has made a false prediction, find and analyze the most influential instances. For example, this could show that the second hospital has older patients on average and the most influential instances from the training data are the few older patients from the first hospital and the model simply lacked the data to learn to predict this subgroup well. The conclusion would be that the model needs to be trained on more patients who are older in order to work well in the second hospital.

Fixing training data

If you have a limit on how many training instances you can check for correctness, how do you make

an efficient selection? The best way is to select the most influential instances, because – by definition – they have the most influence on the model. Even if you would have an instance with obviously incorrect values, if the instance is not influential and you only need the data for the prediction model, it is a better choice to check the influential instances. For example, you train a model for predicting whether a patient should remain in hospital or be discharged early. You really want to make sure that the model is robust and makes correct predictions, because a wrong release of a patient can have bad consequences. Patient records can be very messy, so you do not have perfect confidence in the quality of the data. But checking patient information and correcting it can be very time-consuming, because once you have reported which patients you need to check, the hospital actually needs to send someone to look at the records of the selected patients more closely, which might be handwritten and lying in some archive. Checking data for a patient could take an hour or more. In view of theses costs, it makes sense to check only a few important data instances. The best way is to select patients who have had a high influence on the prediction model. Koh and Liang (2017) showed that this type of selection works much better than random selection or the selection of those with the highest loss or wrong classification.

Advantages of Identifying Influential Instances

The approaches of deletion diagnostics and influence functions are very different from the mostly feature-perturbation based approaches presented in the Model-Agnostic chapter. A look at influential instances emphasizes the role of training data in the learning process. This makes influence functions and deletion diagnostics **one of the best debugging tools for machine learning models**. Of the techniques presented in this book, they are the only ones that directly help to identify the instances which should be checked for errors.

Deletion diagnostics are model-agnostic, meaning the approach can be applied to any model. Also influence functions based on the derivatives can be applied to a broad class of models.

We can use these methods to **compare different machine learning models** and better understand their different behaviors, going beyond comparing only the predictive performance.

We have not talked about this topic in this chapter, but **influence functions via derivatives can also be used to create adversarial training data**. These are instances that are manipulated in such a way that the model cannot predict certain test instances correctly when the model is trained on those manipulated instances. The difference to the methods in the Adversarial Examples chapter is that the attack takes place during training time, also known as poisoning attacks. If you are interested, read the paper by Koh and Liang (2017).

For deletion diagnostics and influence functions, we considered the difference in the prediction and for the influence function the increase of the loss. But, really, **the approach is generalizable** to any question of the form: "What happens to … when we delete or upweight instance z?", where you can fill "…" with any function of your model of your desire. You can analyze how much a training instance influences the overall loss of the model. You can analyze how much a training instance influences the feature importance. You can analyze how much a training instance influences which feature is selected for the first split when training a decision tree.

Disadvantages of Identifying Influential Instances

Deletion diagnostics are very **expensive to calculate** because they require retraining. But history has shown that computer resources are constantly increasing. A calculation that 20 years ago was unthinkable in terms of resources can easily be performed with your smartphone. You can train models with thousands of training instances and hundreds of parameters on a laptop in seconds/minutes. It is therefore not a big leap to assume that deletion diagnostics will work without problems even with large neural networks in 10 years.

Influence functions are a good alternative to deletion diagnostics, but only for models with differentiable parameters, such as neural networks. They do not work for tree-based methods like random forests, boosted trees or decision trees. Even if you have models with parameters and a loss function, the loss may not be differentiable. But for the last problem, there is a trick: Use a differentiable loss as substitute for calculating the influence when, for example, the underlying model uses the Hinge loss instead of some differentiable loss. The loss is replaced by a smoothed version of the problematic loss for the influence functions, but the model can still be trained with the non-smooth loss.

Influence functions are only approximate, because the approach forms a quadratic expansion around the parameters. The approximation can be wrong and the influence of an instance is actually higher or lower when removed. Koh and Liang (2017) showed for some examples that the influence calculated by the influence function was close to the influence measure obtained when the model was actually retrained after the instance was deleted. But there is no guarantee that the approximation will always be so close.

There is **no clear cutoff of the influence measure at which we call an instance influential or non-influential**. It is useful to sort the instances by influence, but it would be great to have the means not only to sort the instances, but actually to distinguish between influential and non-influential. For example, if you identify the top 10 most influential training instances for a test instance, some of them may not be influential because, for example, only the top 3 were really influential.

The influence measures **only take into account the deletion of individual instances** and not the deletion of several instances at once. Larger groups of data instances may have some interactions that strongly influence model training and prediction. But the problem lies in combinatorics: There are n possibilities to delete an individual instance from the data. There are n times (n-1) possibilities to delete two instances from the training data. There are n times (n-1) times (n-2) possibilities to delete three … I guess you can see where this is going, there are just too many combinations.

Software and Alternatives

Deletion diagnostics are very simple to implement. Take a look at the code[113] I wrote for the examples in this chapter.

[113] https://github.com/christophM/interpretable-ml-book/blob/master/manuscript/06.5-example-based-influence-fct.Rmd

For linear models and generalized linear models many influence measures like Cook's distance are implemented in R in the `stats` package.

Koh and Liang published the Python code for influence functions from their paper in a repository[114]. That is great! Unfortunately it is "only" the code of the paper and not a maintained and documented Python module. The code is focused on the Tensorflow library, so you cannot use it directly for black box models using other frameworks, like sci-kit learn.

Keita Kurita wrote a great blog post for influence functions[115] that helped me understand Koh and Liang's paper better. The blog post goes a little deeper into the mathematics behind influence functions for black box models and also talks about some of the mathematical 'tricks' with which the method is efficiently implemented.

[114] https://github.com/kohpangwei/influence-release
[115] http://mlexplained.com/2018/06/01/paper-dissected-understanding-black-box-predictions-via-influence-functions/#more-641

A Look into the Crystal Ball

What is the future of interpretable machine learning? This chapter is a speculative mental exercise and a subjective guess how interpretable machine learning will develop. I opened the book with rather pessimistic short stories and would like to conclude with a more optimistic outlook.

I have based my "predictions" on three premises:

1. **Digitization: Any (interesting) information will be digitized**. Think of electronic cash and online transactions. Think of e-books, music and videos. Think of all the sensory data about our environment, our human behavior, industrial production processes and so on. The drivers of the digitization of everything are: Cheap computers/sensors/storage, scaling effects (winner takes it all), new business models, modular value chains, cost pressure and much more.
2. **Automation: When a task can be automated and the cost of automation is lower than the cost of performing the task over time, the task will be automated**. Even before the introduction of the computer we had a certain degree of automation. For example, the weaving machine automated weaving or the steam machine automated horsepower. But computers and digitization take automation to the next level. Simply the fact that you can program for-loops, write Excel macros, automate e-mail responses, and so on, show how much an individual can automate. Ticket machines automate the purchase of train tickets (no cashier needed any longer), washing machines automate laundry, standing orders automate payment transactions and so on. Automating tasks frees up time and money, so there is a huge economic and personal incentive to automate things. We are currently observing the automation of language translation, driving and, to a small degree, even scientific discovery.

3. **Misspecification: We are not able to perfectly specify a goal with all its constraints**. Think of the genie in a bottle that always takes your wishes literally: "I want to be the richest person in the world!" -> You become the richest person, but as a side effect the currency you hold crashes due to inflation.
   "I want to be happy for the rest of my life!" -> The next 5 minutes you feel very happy, then the genie kills you.
   "I wish for world peace!" -> The genie kills all humans.
   We specify goals incorrectly, either because we do not know all the constraints or because we cannot measure them. Let's look at corporations as an example of imperfect goal specification. A corporation has the simple goal of earning money for its shareholders. But this specification does not capture the true goal with all its constraints that we really strive for: For example, we do not appreciate a company killing people to make money, poisoning rivers, or simply printing its own money. We have invented laws, regulations, sanctions, compliance procedures, labor unions and more to patch up the imperfect goal specification. Another example that you can experience for yourself is Paperclips[116], a game in which you play a machine with the goal of producing as many paperclips as possible. WARNING: It is addictive. I do not want to spoil it too much, but let's say things get out of control really fast. In machine learning, the imperfections in the goal specification come from imperfect data abstractions (biased populations, measurement errors, ...), unconstrained loss functions, lack of knowledge of the constraints, shifting of the distribution between training and application data and much more.

Digitization is driving automation. Imperfect goal specification conflicts with automation. I claim that this conflict is mediated partially by interpretation methods.

The stage for our predictions is set, the crystal ball is ready, now we look at where the field could go!

The Future of Machine Learning

Without machine learning there can be no interpretable machine learning. Therefore we have to guess where machine learning is heading before we can talk about interpretability.

Machine learning (or "AI") is associated with a lot of promises and expectations. But let's start with a less optimistic observation: While science develops a lot of fancy machine learning tools, in my experience it is quite difficult to integrate them into existing processes and products. Not because it is not possible, but simply because it takes time for companies and institutions to catch up. In the gold rush of the current AI hype, companies open up "AI labs", "Machine Learning Units" and hire "Data Scientists", "Machine Learning Experts", "AI engineers", and so on, but the reality is, in my experience, rather frustrating. Often companies do not even have data in the required form and the data scientists wait idle for months. Sometimes companies have such high expectation of AI and Data Science due to the media that data scientists could never fulfill them. And often nobody knows

[116] http://www.decisionproblem.com/paperclips/index2.html

how to integrate data scientists into existing structures and many other problems. This leads to my first prediction.

Machine learning will grow up slowly but steadily.

Digitalization is advancing and the temptation to automate is constantly pulling. Even if the path of machine learning adoption is slow and stony, machine learning is constantly moving from science to business processes, products and real world applications.

I believe we need to better explain to non-experts what types of problems can be formulated as machine learning problems. I know many highly paid data scientists who perform Excel calculations or classic business intelligence with reporting and SQL queries instead of applying machine learning. But a few companies are already successfully using machine learning, with the big Internet companies at the forefront. We need to find better ways to integrate machine learning into processes and products, train people and develop machine learning tools that are easy to use. I believe that machine learning will become much easier to use: We can already see that machine learning is becoming more accessible, for example through cloud services ("Machine Learning as a service" – just to throw a few buzzwords around). Once machine learning has matured – and this toddler has already taken its first steps – my next prediction is:

Machine learning will fuel a lot of things.

Based on the principle "Whatever can be automated will be automated", I conclude that whenever possible, tasks will be formulated as prediction problems and solved with machine learning. Machine learning is a form of automation or can at least be part of it. Many tasks currently performed by humans are replaced by machine learning. Here are some examples of tasks where machine learning is used to automate parts of it:

- Sorting / decision-making / completion of documents (e.g. in insurance companies, the legal sector or consulting firms)
- Data-driven decisions such as credit applications
- Drug discovery
- Quality controls in assembly lines
- Self-driving cars
- Diagnosis of diseases
- Translation. For this book, I used a translation service called (DeepL[117]) powered by deep neural networks to improve my sentences by translating them from English into German and back into English.
- …

The breakthrough for machine learning is not only achieved through better computers / more data / better software, but also:

Interpretability tools catalyze the adoption of machine learning.

[117] https://deepl.com

Based on the premise that the goal of a machine learning model can never be perfectly specified, it follows that interpretable machine learning is necessary to close the gap between the misspecified and the actual goal. In many areas and sectors, interpretability will be the catalyst for the adoption of machine learning. Some anecdotal evidence: Many people I have spoken to do not use machine learning because they cannot explain the models to others. I believe that interpretability will address this issue and make machine learning attractive to organisations and people who demand some transparency. In addition to the misspecification of the problem, many industries require interpretability, be it for legal reasons, due to risk aversion or to gain insight into the underlying task. Machine learning automates the modeling process and moves the human a bit further away from the data and the underlying task: This increases the risk of problems with experimental design, choice of training distribution, sampling, data encoding, feature engineering, and so on. Interpretation tools make it easier to identify these problems.

The Future of Interpretability

Let us take a look at the possible future of machine learning interpretability.

The focus will be on model-agnostic interpretability tools.

It is much easier to automate interpretability when it is decoupled from the underlying machine learning model. The advantage of model-agnostic interpretability lies in its modularity. We can easily replace the underlying machine learning model. We can just as easily replace the interpretation method. For these reasons, model-agnostic methods will scale much better. That is why I believe that model-agnostic methods will become more dominant in the long term. But intrinsically interpretable methods will also have a place.

Machine learning will be automated and, with it, interpretability.

An already visible trend is the automation of model training. That includes automated engineering and selection of features, automated hyperparameter optimization, comparison of different models, and ensembling or stacking of the models. The result is the best possible prediction model. When we use model-agnostic interpretation methods, we can automatically apply them to any model that emerges from the automated machine learning process. In a way, we can automate this second step as well: Automatically compute the feature importance, plot the partial dependence, train a surrogate model, and so on. Nobody stops you from automatically computing all these model interpretations. The actual interpretation still requires people. Imagine: You upload a dataset, specify the prediction goal and at the push of a button the best prediction model is trained and the program spits out all interpretations of the model. There are already first products and I argue that for many applications it will be sufficient to use these automated machine learning services. Today anyone can build websites without knowing HTML, CSS and Javascript, but there are still many web developers around. Similarly, I believe that everyone will be able to train machine learning models without knowing how to program, and there will still be a need for machine learning experts.

We do not analyze data, we analyze models.

The raw data itself is always useless. (I exaggerate on purpose. The reality is that you need a deep understanding of the data to conduct a meaningful analysis.) I don't care about the data; I care about the knowledge contained in the data. Interpretable machine learning is a great way to distill knowledge from data. You can probe the model extensively, the model automatically recognizes if and how features are relevant for the prediction (many models have built-in feature selection), the model can automatically detect how relationships are represented, and – if trained correctly – the final model is a very good approximation of reality.

Many analytical tools are already based on data models (because they are based on distribution assumptions):

- Simple hypothesis tests like Student's t-test.
- Hypothesis tests with adjustments for confounders (usually GLMs)
- Analysis of Variance (ANOVA)
- The correlation coefficient (the standardized linear regression coefficient is related to Pearson's correlation coefficient)
- …

What I am telling you here is actually nothing new. So why switch from analyzing assumption-based, transparent models to analyzing assumption-free black box models? Because making all these assumptions is problematic: They are usually wrong (unless you believe that most of the world follows a Gaussian distribution), difficult to check, very inflexible and hard to automate. In many domains, assumption-based models typically have a worse predictive performance on untouched test data than black box machine learning models. This is only true for big datasets, since interpretable models with good assumptions often perform better with small datasets than black box models. The black box machine learning approach requires a lot of data to work well. With the digitization of everything, we will have ever bigger datasets and therefore the approach of machine learning becomes more attractive. We do not make assumptions, we approximate reality as close as possible (while avoiding overfitting of the training data). I argue that we should develop all the tools that we have in statistics to answer questions (hypothesis tests, correlation measures, interaction measures, visualization tools, confidence intervals, p-values, prediction intervals, probability distributions) and rewrite them for black box models. In a way, this is already happening:

- Let us take a classical linear model: The standardized regression coefficient is already a feature importance measure. With the permutation feature importance measure, we have a tool that works with any model.
- In a linear model, the coefficients measures the effect of a single feature on the predicted outcome. The generalized version of this is the partial dependence plot.
- Test whether A or B is better: For this we can also use partial dependence functions. What we do not have yet (to the best of my best knowledge) are statistical tests for arbitrary black box models.

The data scientists will automate themselves.

I believe that data scientists will eventually automate themselves out of the job for many analysis and prediction tasks. For this to happen, the tasks must be well-defined and there must to be some processes and routines around them. Today, these routines and processes are missing, but data scientists and colleagues are working on them. As machine learning becomes an integral part of many industries and institutions, many of the tasks will be automated.

Robots and programs will explain themselves.

We need more intuitive interfaces to machines and programs that make heavy use of machine learning. Some examples: A self-driving car that reports why it stopped abruptly ("70% probability that a kid will cross the road"); A credit default program that explains to a bank employee why a credit application was rejected ("Applicant has too many credit cards and is employed in an unstable job."); A robot arm that explains why it moved the item from the conveyor belt into the trash bin ("The item has a craze at the bottom.").

Interpretability could boost machine intelligence research.

I can imagine that by doing more research on how programs and machines can explain themselves, we can improve improve our understanding of intelligence and become better at creating intelligent machines.

In the end, all these predictions are speculations and we have to see what the future really brings. Form your own opinion and continue learning!

Contribute to the Book

Thank you for reading my book about Interpretable Machine Learning. The book is under continuous development. It will be improved over time and more chapters will be added. Very similar to how software is developed.

All text and code for the book is open source and available at github.com[118]. On the Github page you can suggest fixes and open issues[119] if you find a mistake or if something is missing.

If you want to help out even more, the issues page is also the best place to find problems to fix and a good way to contribute to the book. If you are interested in a larger contribution, please send me a message with your concrete idea: christoph.molnar.ai@gmail.com[120].

[118] https://github.com/christophM/interpretable-ml-book
[119] https://github.com/christophM/interpretable-ml-book/issues
[120] mailto:christoph.molnar.ai@gmail.com

Citing this Book

If you found this book useful for your blog post, research article or product, I would be grateful if you would cite this book. You can cite the book like this:

```
Molnar, Christoph. "Interpretable machine learning. A Guide for Making Black Box Mod\
els Explainable", 2019. https://christophm.github.io/interpretable-ml-book/.
```

Or use the following bibtex entry:

```
@book{molnar,
  title      = {Interpretable Machine Learning},
  author     = {Christoph Molnar},
  publisher  = {https://christophm.github.io/interpretable-ml-book/},
  note       = {\url{https://christophm.github.io/interpretable-ml-book/}},
  year       = {2019},
  subtitle   = {A Guide for Making Black Box Models Explainable}
}
```

I am always curious about where and how interpretation methods are used in industry and research. If you use the book as a reference, it would be great if you wrote me a line and told me what for. This is of course optional and only serves to satisfy my own curiosity and to stimulate interesting exchanges. My mail is christoph.molnar.ai@gmail.com .

Acknowledgements

Writing this book was (and still is) a lot of fun. But it is also a lot of work and I am very happy about the support I received.

My biggest thank-you goes to Katrin who had the hardest job in terms of hours and effort: she proof-read the book from beginning to end and discovered many spelling mistakes and inconsistencies that I would never have found. I am very grateful for her support.

A big thanks goes to Verena Haunschmid for writing the section about LIME explanations for images. She works in data science and I recommend following her on Twitter: @ExpectAPatronum[121].
I also want to thank all the early readers who contributed corrections[122] on Github!

Furthermore, I want to thank everyone who created illustrations: The cover was designed by my friend @YvonneDoinel[123]. The graphics in the Shapley Value chapter were created by Abi Aryan[124], using icons made by Freepik[125] from Flaticon[126]. The awesome frog with the crystal ball[127] in the chapter about the Future of Interpretability was designed by @TopeconHeroes[128]. Verena Haunschmid created the graphic in the RuleFit chapter. I would like to thank all researchers who allowed me to use images from their research articles.

In at least three aspects, the way I published this book is unconventional. First, it is available both as website and as ebook/pdf. The software I used to create this book is called bookdown, written by Yihui Xie[129], who created many R packages that make it easy to combine R code and text. Thanks a lot! Secondly, I self-publish the book on the platform Leanpub[130], instead of working with a traditional publisher. And third, I published the book as in-progress book, which has helped me enormously to get feedback and to monetize it along the way. Many thanks to leanpub for making this possible and handling the royalties fairly.
I would also like to thank you, dear reader, for reading this book without a big publisher name behind it.

I am grateful for the funding of my research on interpretable machine learning by the Bavarian State Ministry of Science and the Arts in the framework of the Centre Digitisation.Bavaria (ZD.B).

[121] https://twitter.com/ExpectAPatronum
[122] https://github.com/christophM/interpretable-ml-book/graphs/contributors
[123] https://twitter.com/YvonneDoinel
[124] https://twitter.com/GoAbiAryan
[125] http://www.freepik.com/
[126] https://www.flaticon.com/
[127] http://www.chojugiga.com/
[128] https://twitter.com/topeconheroes
[129] https://yihui.name/
[130] https://leanpub.com/

References

"Definition of Algorithm." https://www.merriam-webster.com/dictionary/algorithm. (2017).

Aamodt, Agnar, and Enric Plaza. "Case-based reasoning: Foundational issues, methodological variations, and system approaches." AI communications 7.1 (1994): 39-59.

Alberto, Túlio C, Johannes V Lochter, and Tiago A Almeida. "Tubespam: comment spam filtering on YouTube." In Machine Learning and Applications (Icmla), Ieee 14th International Conference on, 138–43. IEEE. (2015).

Alvarez-Melis, David, and Tommi S. Jaakkola. "On the robustness of interpretability methods." arXiv preprint arXiv:1806.08049 (2018).

Apley, Daniel W. "Visualizing the effects of predictor variables in black box supervised learning models." arXiv preprint arXiv:1612.08468 (2016).

Athalye, Anish, and Ilya Sutskever. "Synthesizing robust adversarial examples." arXiv preprint arXiv:1707.07397 (2017).

Biggio, Battista, and Fabio Roli. "Wild Patterns: Ten years after the rise of adversarial machine learning." Pattern Recognition 84 (2018): 317-331.

Breiman, Leo."Random Forests." Machine Learning 45 (1). Springer: 5-32 (2001).

Brown, Tom B., et al. "Adversarial patch." arXiv preprint arXiv:1712.09665 (2017).

Cohen, William W. "Fast effective rule induction." Machine Learning Proceedings (1995). 115-123.

Cook, R. Dennis. "Detection of influential observation in linear regression." Technometrics 19.1 (1977): 15-18.

Doshi-Velez, Finale, and Been Kim. "Towards a rigorous science of interpretable machine learning," no. Ml: 1–13. http://arxiv.org/abs/1702.08608 (2017).

Fanaee-T, Hadi, and Joao Gama. "Event labeling combining ensemble detectors and background knowledge." Progress in Artificial Intelligence. Springer Berlin Heidelberg, 1–15. doi:10.1007/s13748-013-0040-3. (2013).

Fernandes, Kelwin, Jaime S Cardoso, and Jessica Fernandes. "Transfer learning with partial observability applied to cervical cancer screening." In Iberian Conference on Pattern Recognition and Image Analysis, 243–50. Springer. (2017).

Fisher, Aaron, Cynthia Rudin, and Francesca Dominici. "Model Class Reliance: Variable importance measures for any machine learning model class, from the 'Rashomon' perspective." http://arxiv.org/abs/1801.01489 (2018).

Fokkema, Marjolein, and Benjamin Christoffersen. “Pre: Prediction rule ensembles”. https://CRAN.R-project.org/package=pre (2017).

Friedman, Jerome H, and Bogdan E Popescu. “Predictive learning via rule ensembles.” The Annals of Applied Statistics. JSTOR, 916–54. (2008).

Friedman, Jerome H. “Greedy function approximation: A gradient boosting machine.” Annals of statistics (2001): 1189-1232.

Friedman, Jerome, Trevor Hastie, and Robert Tibshirani. “The elements of statistical learning”. www.web.stanford.edu/~hastie/ElemStatLearn/ (2009).

Fürnkranz, Johannes, Dragan Gamberger, and Nada Lavrač. “Foundations of rule learning.” Springer Science & Business Media, (2012).

Goldstein, Alex, et al. “Package ‘ICEbox’.” (2017).

Goodfellow, Ian J., Jonathon Shlens, and Christian Szegedy. “Explaining and harnessing adversarial examples.” arXiv preprint arXiv:1412.6572 (2014).

Greenwell, Brandon M., Bradley C. Boehmke, and Andrew J. McCarthy. “A simple and effective model-based variable importance measure.” arXiv preprint arXiv:1805.04755 (2018).

Heider, Fritz, and Marianne Simmel. “An experimental study of apparent behavior.” The American Journal of Psychology 57 (2). JSTOR: 243–59. (1944).

Holte, Robert C. “Very simple classification rules perform well on most commonly used datasets.” Machine learning 11.1 (1993): 63-90.

Hooker, Giles. “Discovering additive structure in black box functions.” Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining. (2004).

Kahneman, Daniel, and Amos Tversky. “The Simulation Heuristic.” Stanford Univ CA Dept of Psychology. (1981).

Kaufman, Leonard, and Peter Rousseeuw. “Clustering by means of medoids”. North-Holland (1987).

Kim, Been, Rajiv Khanna, and Oluwasanmi O. Koyejo. “Examples are not enough, learn to criticize! Criticism for interpretability.” Advances in Neural Information Processing Systems (2016).

Koh, Pang Wei, and Percy Liang. “Understanding black-box predictions via influence functions.” arXiv preprint arXiv:1703.04730 (2017).

Laugel, Thibault, et al. “Inverse classification for comparison-based interpretability in machine learning.” arXiv preprint arXiv:1712.08443 (2017).

Letham, Benjamin, et al. “Interpretable classifiers using rules and Bayesian analysis: Building a better stroke prediction model.” The Annals of Applied Statistics 9.3 (2015): 1350-1371.

Lipton, Peter. “Contrastive explanation.” Royal Institute of Philosophy Supplements 27 (1990): 247-266.

Lipton, Zachary C. “The mythos of model interpretability.” arXiv preprint arXiv:1606.03490, (2016).

Lundberg, Scott, and Su-In Lee. “An unexpected unity among methods for interpreting model predictions.” arXiv preprint arXiv:1611.07478 (2016).

Martens, David, and Foster Provost. “Explaining data-driven document classifications.” (2014).

Miller, Tim. “Explanation in artificial intelligence: Insights from the social sciences.” arXiv Preprint arXiv:1706.07269. (2017).

Nickerson, Raymond S. “Confirmation Bias: A ubiquitous phenomenon in many guises.” Review of General Psychology 2 (2). Educational Publishing Foundation: 175. (1998).

Papernot, Nicolas, et al. “Practical black-box attacks against machine learning.” Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security. ACM (2017).

Ribeiro, Marco Tulio, Sameer Singh, and Carlos Guestrin. “Anchors: High-precision model-agnostic explanations.” AAAI Conference on Artificial Intelligence (2018).

Ribeiro, Marco Tulio, Sameer Singh, and Carlos Guestrin. “Model-agnostic interpretability of machine learning.” ICML Workshop on Human Interpretability in Machine Learning. (2016).

Ribeiro, Marco Tulio, Sameer Singh, and Carlos Guestrin. “Why should I trust you?: Explaining the predictions of any classifier.” Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining. ACM (2016).

Shapley, Lloyd S. “A value for n-person games.” Contributions to the Theory of Games 2.28 (1953): 307-317.

Staniak, Mateusz, and Przemyslaw Biecek. “Explanations of model predictions with live and breakDown packages.” arXiv preprint arXiv:1804.01955 (2018).

Su, Jiawei, Danilo Vasconcellos Vargas, and Kouichi Sakurai. “One pixel attack for fooling deep neural networks.” IEEE Transactions on Evolutionary Computation (2019).

Szegedy, Christian, et al. “Intriguing properties of neural networks.” arXiv preprint arXiv:1312.6199 (2013).

Wachter, Sandra, Brent Mittelstadt, and Chris Russell. “Counterfactual explanations without opening the black box: Automated decisions and the GDPR.” (2017).

Yang, Hongyu, Cynthia Rudin, and Margo Seltzer. “Scalable Bayesian rule lists.” Proceedings of the 34th International Conference on Machine Learning-Volume 70. JMLR. org, 2017.

Zhao, Qingyuan, and Trevor Hastie. “Causal interpretations of black-box models.” Journal of Business & Economic Statistics, to appear. (2017).

Štrumbelj, Erik, and Igor Kononenko. “A general method for visualizing and explaining black-box regression models.” In International Conference on Adaptive and Natural Computing Algorithms, 21–30. Springer. (2011).

Štrumbelj, Erik, and Igor Kononenko. “Explaining prediction models and individual predictions with feature contributions.” Knowledge and information systems 41.3 (2014): 647-665.

R Packages Used for Examples

base. R Core Team (2018). R: A language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria. URL https://www.R-project.org/.

data.table. Matt Dowle and Arun Srinivasan (2019). data.table: Extension of `data.frame`. R package version 1.12.1. http://r-datatable.com

dplyr. Hadley Wickham, Romain François, Lionel Henry and Kirill Müller (2018). dplyr: A Grammar of Data Manipulation. R package version 0.7.8. https://CRAN.R-project.org/package=dplyr

ggplot2. Hadley Wickham, Winston Chang, Lionel Henry, Thomas Lin Pedersen, Kohske Takahashi, Claus Wilke and Kara Woo (2018). ggplot2: Create Elegant Data Visualisations Using the Grammar of Graphics. R package version 3.1.0. https://CRAN.R-project.org/package=ggplot2

iml. Christoph Molnar (2019). iml: Interpretable Machine Learning. R package version 0.9.0. https://github.com/christophM/iml

knitr. Yihui Xie (2018). knitr: A General-Purpose Package for Dynamic Report Generation in R. R package version 1.20. https://CRAN.R-project.org/package=knitr

libcoin. Torsten Hothorn (2019). libcoin: Linear Test Statistics for Permutation Inference. R package version 1.0-3. https://CRAN.R-project.org/package=libcoin

memoise. Hadley Wickham, Jim Hester, Kirill Müller and Daniel Cook (2017). memoise: Memoisation of Functions. R package version 1.1.0. https://CRAN.R-project.org/package=memoise

mlr. Bernd Bischl, Michel Lang, Lars Kotthoff, Julia Schiffner, Jakob Richter, Zachary Jones, Giuseppe Casalicchio, Mason Gallo and Patrick Schratz (2018). mlr: Machine Learning in R. R package version 2.13. https://CRAN.R-project.org/package=mlr

mvtnorm. Alan Genz, Frank Bretz, Tetsuhisa Miwa, Xuefei Mi and Torsten Hothorn (2018). mvtnorm: Multivariate Normal and t Distributions. R package version 1.0-8. https://CRAN.R-project.org/package=mvtnorm

NLP. Kurt Hornik (2018). NLP: Natural Language Processing Infrastructure. R package version 0.2-0. https://CRAN.R-project.org/package=NLP

ParamHelpers. Bernd Bischl, Michel Lang, Jakob Richter, Jakob Bossek, Daniel Horn and Pascal Kerschke (2018). ParamHelpers: Helpers for Parameters in Black-Box Optimization, Tuning and Machine Learning. R package version 1.11. https://CRAN.R-project.org/package=ParamHelpers

partykit. Torsten Hothorn and Achim Zeileis (2019). partykit: A Toolkit for Recursive Partytioning. R package version 1.2-3. https://CRAN.R-project.org/package=partykit

pre. Marjolein Fokkema and Benjamin Christoffersen (2018). pre: Prediction Rule Ensembles. R package version 0.6.0. https://CRAN.R-project.org/package=pre

readr. Hadley Wickham, Jim Hester and Romain Francois (2017). readr: Read Rectangular Text Data. R package version 1.1.1. https://CRAN.R-project.org/package=readr

rpart. Terry Therneau and Beth Atkinson (2018). rpart: Recursive Partitioning and Regression Trees. R package version 4.1-13. https://CRAN.R-project.org/package=rpart

tidyr. Hadley Wickham and Lionel Henry (2018). tidyr: Easily Tidy Data with 'spread()' and 'gather()' Functions. R package version 0.8.2. https://CRAN.R-project.org/package=tidyr

tm. Ingo Feinerer and Kurt Hornik (2018). tm: Text Mining Package. R package version 0.7-5. https://CRAN.R-project.org/package=tm