Interpretable Machine Learning

@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 2018-08-14

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License

Contents

Introduction

This book will teach 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 mathematics. This book is not for people who try to learn machine learning from scratch. If you are new to machine learning, there are loads of books and other resources for learning the basics. I recommend the book Elements of Statistical Learning[1] from Hastie, Tibshirani, and Friedman (2009) [2] and Andrew Ng's "Machine Learning" online course on coursera[3] to get started with machine learning. Both the book and the course are available free of charge!

New methods for machine learning interpretability are published at breakneck speed. Keeping update with all of them would be madness and simply impossible. That's why you won't find the most novel and shiny methods in this book, but rather the basic concepts of machine learning interpretability. These basics will prepare you well to make machine learning models interpretable. Internalizing the basic concepts also empowers you to better understand and evaluate any new paper on interpretability that has been published on arxiv.org[4] in the last 5 minutes since you began to read this book (I may be exaggerating).

This book starts with some (dystopian) short stories, which are not needed to understand the book, but hopefully are entertaining! Then the book explores the concepts of machine learning interpretability: It shows when interpretability is important and discusses different types of explanations. Definitions used throughout the book can be looked up here. Most of the models and methods explained are presented using real data examples described here. One way to make machine learning interpretable is to use interpretable models, like linear models or decision trees. The other option is the use model-agnostic interpretation tools, that can be applied to any supervised machine learning model. The model-agnostic chapter covers methods like 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 output. Finish the book 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!

[1] https://web.stanford.edu/~hastie/ElemStatLearn/

[2] Hastie, T, R Tibshirani, and J Friedman. 2009. The elements of statistical learning. http://link.springer.com/content/pdf/10.1007/978-0-387-84858-7.pdf.

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

[4] arxiv.org

Storytime

Get started with a few 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[5]. If you like these kind of stories or are interested in AI, I recommend signing 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 was removing the pump's computer 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 he was unscrewing the pump's back plate. After he had removed 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 an ironic undertone.

He booted the pump. Lena plugged the other end of the cable into her tablet. "Alright, diagnostics are running." she announced. "I am really curious to see what went wrong." "It certainly shot our John Doe to Nirvana. This high concentration of this morphine stuff. Man. I mean … that's a first, right? Normally a broken pump gives too little of the sweet stuff or nothing. But never, you know, like this crazy shot." Tom explained. "I know. You don't have to convince me … Hey, look at that." Lena held up the tablet. "Do you see this peek here? That's the painkiller potency. Look: this line shows the reference level. The poor guy had a painkiller mix in his blood system that could kill him 17 times over. All 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 sensory functions seem to be okay. 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. It also picked up the slowing heart rate of the patient and the extremely low cortisol levels caused by the morphine derivate and the other pain blocking agents." She continued swiping through the diagnostics. Tom stared in a gaze at the screen. It was his first real device failure to investigate.

"Ok, here is our first piece of the puzzle. The system failed to send a warning to the hospital communication channel. The warning was triggered, but rejected on the protocol level. Might be our fault, but could be also the hospitals fault. Please send the logs over to the IT team." Lena told Tom. Tom nodded, eyes still locked on the screen. Lena continued. "It's weird. The warning should also have triggered the shutdown of the pump. But it obviously failed to do so. That must be a bug. Something the quality team missed. Something really bad. Maybe connected to the protocol issue." "So, the pump's emergency systems somehow broke, but why did the pump go full bananas

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

and inject so much painkiller?" Tom wondered. "Good question. You are right. Protocol emergency failure aside, the pump shouldn't have administered this amount of medication in the first place. The algorithm should have stopped much earlier on its own, given the low cortisol level 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 would have read the documentation, which I sent to you, you would know that the pump was first trained on animal trials, later on humans to learn on its own how to find the perfect amount of pain killer, given the sensory input. The pump's algorithm might be opaque and complex, but it is not random. That means the pump would show the same behaviour in the exact same situation. Our patient would die again. Some combination or unwanted interaction of the sensory inputs must have triggered the erratical behaviour of the pump. That's why we have to dig deeper and find out what happened here." Lena explained.

"I see …" Tom responded, lost in thoughts. "Wasn't the patient going to die soon anyway? Because of cancer or something?" Lena nodded while reading the analysis report. Tom got up, and walked to the window. He looked outside, eyes fixating on some point in the distance. "Maybe the machine did him a favour, you know, like 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. For a reason. If I were the pump, I would have done the same." She finally lifted her head and looked at him. He continued looking at something outside. Nobody said anything for a minute or two. Lena lowered her head again and continued the analysis. "No Tom. It's a bug… Just some goddam bug".

Trust Fall

2050, A subway station in Singapore

She was rushing to Bishan subway station. With her thoughts she was already at work. The tests for the new neural architecture should have finished by now. She lead the re-design of the government's "tax affinity prediction system for individual entities", which predicts if an individual will hide money from the tax office. Her team came up with an elegant piece of engineering. If successful, the system would not only serve the tax office, but also feed into other systems, like the anti-terrorist defence and the trade registry. One day, the government might even integrate it into the civic trust score. The trust system estimates how trustworthy an individual is. The estimate affects every part of your daily life, like getting a loan or how long you have to wait when getting a new passport. Descending the escalator, she imagined how an integration into the current trust score system could look like.

Routinely, she wiped her hand over the RFID reader without reducing her walking speed. Her mind was occupied, yet 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, bottom first, onto the floor. The door was supposed to open, … but it didn't. Baffled, she stood up and looked at the gate's screen. "Please try again some other time." it suggested in friendly colours. A person walked by, and, ignoring her, wiped his hand over the reader. The doors opened and he walked through. The doors closed again.

She wiped her nose. It hurt, but at least it wasn't bleeding. She tried to open the door, but got rejected again. It was odd. Maybe her public transport account did not have sufficient tokens. She raised her watch to check the account balance.

"Login denied. Please contact your Citizens Advice Bureau!" the watch informed her.

A feeling of nausea hit her like a fist to the stomach. With trembling hands she started the mobile game "Sniper Guild", an ego-shooter. After a few seconds, the loading screen shut down. She felt dizzy and sat down on the floor again.

There was only one possible explanation: Her trust score had dropped. Substantially. A small drop meant minor inconveniences, like not getting first class flights. A low trust score was rare and meant that you were classified as a threat to society. One measure for dealing with those people was to keep them from public places - for example the subway. The government restricted financial transactions of low-trust subjects. They also started actively monitoring of your behaviour on social media, even going as far as to restrict certain contents, like violent games. It became exponentially more difficult to increase your trust score, the lower it was. People with a very low trust score usually never recovered.

She could think of no reason why her score should have dropped. The score was based on machine learning. It worked like a well oiled engine, stabilising society. The performance of the trust score system was always closely monitored. Machine learning had become a lot better since the beginning of the century. It had become so efficient that decisions made by the trust score system could not be disputed. An infallible system.

She laughed hysterically. Infallible system. If only. The system failed rarely. But it did fail. She was an edge case; an error of the system; from now on, an outcast. Nobody dared to question the system. It had become too integrated into the government, into society itself to be questioned. In democratic countries it was forbidden to form anti-democratic movements, not because they where inherently vicious, but because they would de-stabilise the current system. The same logic applied to algorithmic critique: Critique in the algorithms was forbidden, because of its danger to the status quo.

The algorithmic trust was the very fabric the societal order was made of. For the greater good, rare incorrect trust scorings were accepted silently. Hundreds of other prediction systems and databases were feeding into the score, so it was impossible to know what triggered the drop in her score. Wild emotions twisted her, most of all, terror. She vomited on the floor.

Her tax affinity system was eventually integrated into the trust system, but she never got to know.

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 with her left hand one of the projector drones, that were buzzing in the room. "History is important!" the teacher said with an upset voice, looking at the girls. Xola blushed. She hadn't expected her teacher to overhear her.

“Xola, what did you just learn?” the teacher asked her. “That the ancient people used all resources from earther planet and then they died?” she asked carefully. “No. They made the climate go hot and it wasn’t the people, it was the computers and machines. And it’s planet earth, not earther planet.” another girl named Lin added. Xola nodded in agreement. With a hint of a pride smile, the teacher nodded. “You are both right. Do you know why it happened?” “Because the humans were short-sighted and greedy?” Xola wondered. “The humans couldn’t stop their machines!” Lin blurted out.

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

The teacher started the poem. A dozen of the little drones repositioned themselves in front of the kids’ eyes and started projecting the video. It showed a person in a suit, standing in a forest but with only tree stumps left. He started reciting:

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 coins are mined,

and nature has fallen behind;

There will be trouble,

after all, exponential growth is a bubble.

The tragedy of the commons unfolding,

exploding,

before our eyes.

Cold computing 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 civilisation.

And so we march on, relentlessly.

As we are part of the machine.

Embracing entropy.

"A grim reminder." the teacher broke the silence in the room, "It's 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 motors. Xola liked how it warmed her hands.

What Is Machine Learning?

Machine learning is a method for teaching computers to make and improve predictions or behaviours based on data.

Predicting the value of a house by learning from historical house sales can be done with machine learning. The book focuses on supervised machine learning, which includes all problems where we know the label or the outcome of interest (e.g. the past sale prices of houses) and want to learn to predict. Excluded from supervised learning are, for example, clustering tasks (=unsupervised learning), where we have no label, but want to find clusters of data points. Also excluded are things like reinforcement learning, where an agent learns to optimise some reward by acting in an environment (e.g. a computer playing Tetris). The goal in supervised learning is to learn a predictive model that maps features (e.g. house size, location, type of floor, ...) to an output (e.g. value of the house). If the output is categorical, the task is called classification and if it is numerical, then regression. Machine learning is a set of algorithms that can learn these mappings from training data, which are pairs of input features and a target. The machine learning algorithm learns a model by changing parameters (like linear weights) or learning structures (like trees). The algorithm is guided by a score or loss function that is minimised. In the house value example, the machine minimises some form of difference between the estimated house sales price and the predicted sales price. A fully trained machine learning model can then be used to make predictions for new instances and be integrated into a product or process.

Estimating house values, recommending products, identifying street signs, counting people on the street, assessing a person's credit worthiness and detecting fraud: All these examples have in common that they can and increasingly are realised with machine learning. The tasks are different, but the approach is the same: Step 1 is to collect data. The more, the better. The data needs to have the information you want to predict and additional information from which the prediction should be made. For a street sign detector ("Is there a street sign in the image?") you would collect street images and label them accordingly with street sign yes vs. no. For a loan default predictor you need historical data from actual loans, the information if the customers defaulted on their loans and data that helps you predict, like the customers income, age and so on. For a house value estimator, you would want to collect data from historical house sales and information about the real estate like size, location and so on. Step 2: Feed this information into a machine learning algorithm, which produces a sign detector model, a credit worthiness model or a house value estimator. This model can then be used in Step 3: Integrate the model into the product or process, like a self-driving car, a loan application process or a real estate marketplace website.

Machines exceed humans in a lot of tasks, like playing chess (or, since recently, Go) or predicting the weather. Even if the machine is as good as a human at a task, or slightly worse, there remain big advantages in speed, reproducibility and scale. A machine learning model that has been implemented once, can do a task much faster than humans, will reliably produce the same results from the same input and can be copied endlessly. Replicating a machine learning model on another machine is fast and cheap. Training a second human to do a task can take decades (especially when they are young) and is very costly. A big disadvantage of using machine learning is that insights about the data and

the task the machine is solving are hidden within 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, like the RandomForest, consist of hundreds of decision trees that "vote" to make predictions. Again, to fully understand how the decision was made, you would need to look into the votes and structures of each of the hundreds of trees. That just does not work out, no matter how clever you are or how good your working memory is. The best performing models are blends of multiple models (also called ensembles), which in itself cannot be interpreted, even if each single model would be interpretable. If you only focus on performance, you automatically will get more and more opaque models. Just have a look at interviews with winners on the kaggle.com machine learning competition platform[6]: The winning models were mostly ensembles of models or very complex models like boosted trees or deep neural networks.

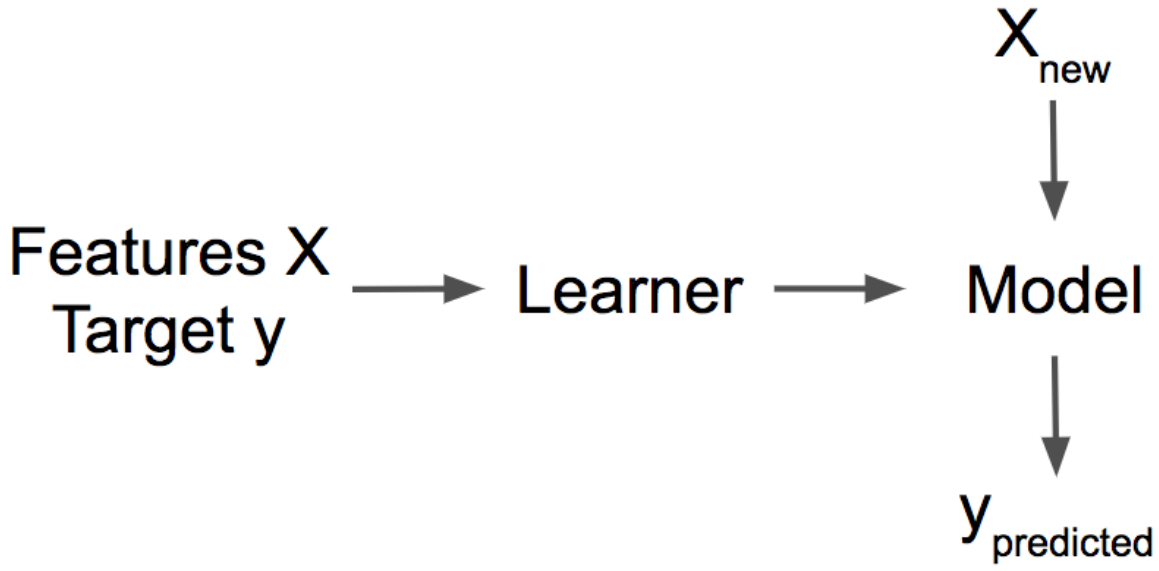
Definitions

To avoid confusion through 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 [7]. An algorithm can be seen as a recipe the defines the inputs, the output and all the steps required to get from the inputs to the output. Cooking recipes are algorithms, where the ingredients are the inputs, the cooked meal is the output and the preparation and cooking steps are the algorithm instructions.
- A **Machine learning algorithm** is a set of rules that a machine follows to learn how to a achieve a particular goal. The output of a machine learning algorithm is a machine learning model. Machine learning algorithms are also called "learner" or "inducer" (e.g. "tree inducer").
- A **(Machine learning) Model** is the outcome of a machine learning algorithm. This can be a set of weights for a linear model or for a neural network plus the information about the architecture. Other names for the rather unspecific word "model" are "predictor" or - depending on the task it solves - "classifier" or "regression model".
- **Dataset**: A table containing the data from which the machine learns. The dataset contains the features and the target. When used for inducing a model, the dataset is called training data.
- **Features**: The features/information used for prediction/classification/clustering. A feature is one column in the dataset. Throughout the book, the features are assumed to be interpretable, meaning it's easy to understand what they stand for. An exception are images where each input feature is a pixel and interpretability works often by graying out bigger parts of the images.
- **Target**: The thing the machine learns to predict.
- **(machine learning) Task**: The combination of a dataset with features and a target. Depending on the type of the target, the task can be classification, regression, survival analysis, clustering, or outlier detection.
- **Prediction**: The machine learning model "guesses" what the target value should be based on given features.

[6] http://blog.kaggle.com/
[7] "Definition of Algorithm." 2017. https://www.merriam-webster.com/dictionary/algorithm.

- **Instance**: One row in the dataset. Other names for 'instance' are: (data) point, example, observation.



plot of chunk learner-definition, "A learner learns a model from labeled training data. The model is used to make predictions."

Interpretability

Throughout the book, I will use this rather simple, yet elegant definition of interpretability from Miller (2017) [8] : **Interpretability is the degree to which a human can understand the cause of a decision**. Another take is: **Interpretability is the degree to which a human can consistently predict the model's result**. The higher the interpretability of a model, the easier it is for someone to comprehend why certain decisions (read: predictions) were made. A model has better interpretability than another model, if its decisions are easier to comprehend for a human than decisions from the second model. I will be using both the terms interpretable and explainable equally. Like Miller (2017), I believe it makes sense to distinguish between the terms interpretability/explainability and explanation. Making a machine learning interpretable can, but does not necessarily have to, imply providing a (human-style) explanation of a prediction. See the section about explanations to learn what we humans see as a good explanation.

The Importance of Interpretability

If a machine learning model performs well, **why not 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 [9])

Let's dive deeper into the reasons why interpretability is so important. In predictive modelling, you have to make a trade-off: Do you simply want to know **what** is predicted? For example the probability that a client will churn or how effective some medication will be for a patient. Or do you want to know **why** the prediction was made and possibly paying for the interpretability with a drop in accuracy? In some cases you don't care why a decision was made, only the assurance that the predictive performance was good on a test dataset is enough. But in other cases, knowing the 'why' can help you understand more about the problem, the data and why a model might fail. Some models might not need explanations, because they are used in a low risk environment, meaning a mistake has no severe consequences, (e.g. a movie recommender system) or the method has already been extensively studied and evaluated (e.g. optical character recognition). The necessity for interpretability comes from an incompleteness in the problem formalisation (Doshi-Velez and Kim 2017), meaning that for certain problems or tasks it is not enough to get the answer (the **what**). The model also has to give an explanation how it came to the answer (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) **Human curiosity and learning**: Humans have a mental model of their environment, which gets updated when something

[8] Miller, Tim. 2017. "Explanation in Artificial Intelligence: Insights from the Social Sciences." arXiv Preprint arXiv:1706.07269.

[9] Doshi-Velez, Finale, and Been Kim. 2017. "Towards A Rigorous Science of Interpretable Machine Learning," no. Ml: 1–13. http://arxiv.org/abs/1702.08608.

unexpected happens. This update is done by finding an explanation for the unexpected event. For example, a human feels unexpectedly sick and asks himself: "Why do I feel so sick?". He learns that he becomes sick every time he eats those red berries. He updates his mental model and decides that the berries caused the sickness and therefore should be avoided. Curiosity and learning is important for any machine learning model used in the research context, where scientific findings stay completely hidden, when the machine learning model only gives predictions without explanations. To facilitate learning and satisfy curiosity about why certain predictions or behaviours are created by machines, interpretability and explanations are crucial. Of course, humans don't need an explanation for everything that happens. Most people are okay with not understanding how a computer works. The emphasis of this point is more on unexpected events, that makes us curious. Like: Why is my computer shutting down unexpectedly?

Closely related to learning is the human desire to **find meaning in the world**. We want to reconcile 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 himself. There is a contraction between the knowledge about the dog's past behaviour and the newly made, unpleasant experience of the bite. The explanation of the vet reconciles the dog holders contradiction: "The dog was under stress and did bite, dogs are animals and this can happen." The more a machine's decision affects a human's life, the more important it will be for the machine to explain its behaviour. When a machine learning model rejects a loan application, this could be quite unexpected for the applicant. He can only reconcile this inconsistency between expectation and reality by having some form of explanation. The explanations don't actually have to fully explain the situation, but should address a main cause. Another example is algorithmic product recommendation. Personally, I always reflect on why certain products or movies have been recommended to me algorithmically. Often it is quite clear: The advertising is following me on the Internet because I have bought a washing machine recently, and I know that I will be followed by washing machine advertisements the next days. Yes, it makes sense to suggest gloves, when I already have a winter hat in my shopping basket. The algorithm recommended this movie, because users that liked other movies that I also liked, enjoyed the recommended movie. Increasingly, Internet companies are adding explanations to their recommendations. A good example is the Amazon product recommendation based on frequently bought product combinations:

There is a shift in many scientific disciplines 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 can only be solved with big datasets and black box machine learning models. The model itself becomes a source of knowledge, instead of the data. Interpretability allows to tap into this additional knowledge captured by the model.

Machine learning models are taking over real world tasks, that demand **safety measures** and testing. Imagine a self-driving car automatically detects cyclists, which is as desired. You want to be 100% sure that the abstraction the system learned will be fail-safe, because running over cyclists is quite bad. An explanation might reveal that the most important feature learned is to recognise the two wheels of a bike and this explanation helps you to think about edge cases like bikes with side bags, that partially cover the wheels.

By default most machine learning models pick up biases from the training data. This can turn your machine learning models into racists which discriminate against protected groups. Interpretability is a useful debugging tool to **detect bias** in machine learning models. It might happen that the machine learning model you trained for automatically approving or rejecting loan applications discriminates against some minority. Your main goal is to give out loans to people that will pay them back eventually. In this case, the incompleteness in the problem formulation lies in the fact that you not only want to minimise loan defaults, but you are also required to not discriminate based on certain demographics. This is an additional constraint, which is part of your problem formulation (Handing out loans in a low-risk and compliant way), which is not captured by the loss function, which the machine learning model optimises.

The process of integrating machines and algorithms into our daily lives demands interpretability to increase **social acceptance**. People attribute beliefs, desires, intentions and so on to objects. In a famous experiment, Heider and Simmel (1944) [10] showed the participants videos of shapes, where a 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, attributing intentions

[10]Heider, Fritz, and Marianne Simmel. 1944. "An Experimental Study of Apparent Behavior." The American Journal of Psychology 57 (2). JSTOR: 243–59.

and even emotions and personality traits to the shapes. Robots are a good example, like my vacuum cleaner, which I named 'Doge'. When Doge gets stuck, I think: "Doge wants to continue cleaning, but asks me for help because it got stuck." Later, when Doge finished cleaning and searches the home base to recharge I think: "Doge has the desire to recharge and intents to find the home base". Also I attribute personality traits: "Doge is a bit dumb, but in a cute way". These are my thoughts, especially when I find out that Doge threw over some plant while cleaning the house dutifully. A machine or algorithm explaining its prediction will receive more acceptance. See also the chapter about explanations, which argues that explanations are a social process.

Explanations are used to **manage social interactions**. Through the creation of a shared meaning of something, the explainer influences the actions, emotions and beliefs of the receiver of the explanation. In order to allow a machine to interact with us, it might need to shape our emotions and beliefs. Machines have to "persuade" us, so that we believe that they can achieve their intended goal. I would not completely accept my robot vacuum cleaner if it would not explain its behaviour 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 can be a misalignment between the goal of the explaining machine, which is generating trust, and the goal of the recipient, which is to understand the prediction or behaviour. Maybe the correct explanation why Doge got stuck could be that the battery was very low, additionally one of the wheels is not working properly and on top of that there is a bug that causes the robot to re-try to go to the same spot over and over again, even though there was some obstacle in the way. These reasons (and some more) caused the robot to get stuck, but it only explained that there was something in the way, and this was enough for me to trust its behaviour, and to get a shared meaning of that accident, which I can share with my girlfriend. ("By the way, Doge got stuck again in the bathroom, we have to remove the carpets before we let it clean"). The example of the robot getting stuck on the carpet might not even require an explanation, because I can explain it to myself by observing that Doge can't move on this carpet mess. But there are other situations, which are less obvious, like a full dirt bag.

Doge, my vacuum cleaner got stuck. As an explanation for the accident, Doge told me that it needs to be on a flat surface.

Only with interpretability can machine learning algorithms be **debugged and audited**. So even in low risk environments, like movie recommendation, interpretability in the research and development stage as well as after deployment is valuable. Because later, when some model is used in a product, things can go wrong. Having an interpretation for a faulty prediction helps to understand the cause of the fault. It delivers a direction for how to fix the system. Consider an example of a husky versus wolf classifier, that misclassifies some huskies as wolfs. Using interpretable machine learning methods, you would find out that the misclassification happened due to the snow on the image. The classifier learned to use snow as a feature for classifying images as wolfs, which might make sense in terms of separating features in the training dataset, but not in the real world use.

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

- Fairness: Making sure the predictions are unbiased and not discriminating against protected groups (implicit or explicit). An interpretable model can tell you why it decided that a certain person is not worthy of a credit and for a human it becomes easier to judge if the decision was based on a learned demographic (e.g. racial) bias.
- Privacy: Ensuring that sensitive information in the data is protected.
- Reliability or Robustness: Test that small changes in the input don't lead to big changes in the prediction.
- Causality: Check if only causal relationships are picked up. Meaning a predicted change in a decision due to arbitrary changes in the input values are also happening in reality.

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

When we don't need interpretability.

The following scenarios illustrate when we don't need or even don't want interpretability for 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 to for their next vacation based on Facebook data. Mike just likes it to surprise his friends with educated guesses where they're going on vacation. There is no real problem if the model is wrong (just a little embarrassment for Mike), it is also not problematic if Mike can't explain the output of his model. It's perfectly fine not to have any interpretability. The situation would change when Mike starts building a company around these vacation destination predictions. If the model is wrong, the company will lose money, or the model could refuse services to people based on some learned racial bias. As soon as the model has a significant impact, either financially or socially, the 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 of envelopes and extracts the addresses. There are years of experience in using these systems and it is clear that they work. Also we are not really interested in gaining additional insights about the task at hand.

Interpretability might enable **gaming the system**. Problems with users fooling a system result from a mismatch in the objectives of the creator and the user of a model. Credit scoring is one such system because the banks want to ensure to give loans to applicants who are likely to return it and the applicants have the goal to get the loan even if the bank has a different opinion. This mismatch between objectives introduces incentives for the applicants to game the system to increase their chances of getting a loan. If an applicant knows that having more than two credit cards affects the score negatively, he simply returns his third credit card to improve his score, and simply gets a new card after the loan has been approved. While his score improved, the true probability of repaying the loan remained the same. The system can only be gamed if the inputs are proxies for another feature, but are not the cause of the outcome. Whenever possible, proxy features should be avoided, as they are often the reason for defective models. For example, Google developed a system called Google Flu Trends for predicting flu outbreaks that correlates Google searches with flu outbreaks - and it performed rather poorly so far. The distribution of searches changed and Google Flu Trends missed many flu outbreaks. Google searches are not known to cause the flu and people searching for symptoms like "fever" merely imply a correlation. Ideally, models would only use causal features, because then they are not gameable and a lot of issues with biases would not occur.

Criteria for Interpretability Methods

Methods for machine learning interpretability can be classified according to different criteria:

- **Intrinsic or post hoc?** Intrinsic interpretability means selecting and training a machine learning model that is considered to be intrinsically interpretable (for example short decision trees). Post hoc interpretability means selecting and training a black box model (for example a neural network) and applying interpretability methods after the training (for example measuring the feature importance). The "intrinsic or post hoc"-criterion determined the layout of the chapters in the book: The two main chapters are the intrinsically interpretable models chapter and the post hoc (and model-agnostic) interpretability methods chapter.
- **Outcome of the interpretability method**: The different interpretability methods can be roughly differentiated according to their outcomes:
  - **Feature summary statistic**: Many interpretability methods provide a kind of summary statistic of how each feature affects the model predictions. These can be feature importance measures or statistics about the interaction strength between features.
  - **Feature summary visualization**: Most feature summary statistics can be visualized. However, some feature summaries can only be visualized and not meaningfully be placed in a table. The partial dependence of a feature is such a case. For non-linear relationships, partial dependence plots are arbitrary curves showing a feature and the average predicted outcome.
  - **Model internals (e.g. learned weights)**: The interpretation of intrinsically interpretable models falls under this category. Examples are the weights in linear models or the learned tree structure (which features and feature values are 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 at the same time summary statistics for the features. Another method that outputs model internals is the visualization of feature detectors that are learned in convolutional neural networks. Interpretability methods that output model internals are model-specific by definition (see next point).
  - **Data point**: This category includes all methods that return data points (can be existing or newly created) to make a model interpretable. Counterfactuals, for example: To explain the prediction of a data point, find a similar data point by changing some of the features for which the predicted outcome changes in a relevant way (like a flip in the predicted class). Another example is the identification of prototypes of predicted classes. Interpretability methods that output new data points require that the data points themselves can be interpreted. This works well for images and text, but is less useful for tabular data with hundreds of features.
  - **Intrinsically interpretable model**: This is a little circular, but one solution to interpreting black box models is to approximate them (either globally or locally) with an interpretable model. The interpretable model themselves are interpreted by internal model parameter 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. Any tool that only works for e.g. interpreting neural networks is model-specific. Model-agnostic tools can be used on any machine learning model and are usually post hoc. These agnostic methods usually operate by analysing feature input and output pairs. By definition, these methods can't have access to any model internals like weights or structural information.
- **Local or global?**: Does the interpretation method explain a single 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, which 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 is capable of picking up. If you are using convolutional neural networks for classifying 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 of the specific model that is learned in the end and not about how single predictions are made. For this level of transparency, only knowledge about the algorithm and not about the data or concrete learned models are required. This book focuses on model interpretability and not algorithm transparency. Algorithms like the least squares method for linear models are well studied and understood. They score high in transparency. It is not clear how they exactly work, so they are less transparent.

Global, Holistic Model Interpretability

How does the trained model make predictions?

You could call a model interpretable if you can comprehend the whole model at once (Lipton 2016[11]). To explain the global model output, you need the trained model, knowledge about the algorithm and the data. This level of interpretability is about understanding how the model makes the decisions, based on a holistic view of its features and each of the learned components like weights, parameters, and structures. Which features are the important ones and what kind of interactions are happening?

[11]Lipton, Zachary C. 2016. "The Mythos of Model Interpretability." ICML Workshop on Human Interpretability in Machine Learning, no. Whi.

Global model interpretability helps to understand the distribution of your target variable based on the features. Arguably, global model interpretability is very hard to achieve in practice. Any model that exceeds a handful of parameters or weights, probably won't fit in an average human's short term memory. I'd argue that you cannot really imagine a linear model with 5 features and draw in your head the hyperplane that was estimated in the 5-dimensional feature space. Each feature space with more than 3 dimensions is just not imaginable for humans. Usually when people try to comprehend a model, they look at parts of it, like the weights in linear models.

Global Model Interpretability on a Modular Level

How do parts of the model influence predictions?

You might not be able to comprehend a Naive Bayes model with many hundred features, because there is no way you could hold all the feature weights in your brain's working memory. But you can understand a single weight easily. Not many models are interpretable on a strict parameter level. While global model interpretability is usually out of reach, there is a better chance to understand at least some models on a modular level. In the case of linear models, the interpretable parts are the weights and the distribution of the features, for trees it would be splits (used feature plus the cut-off point) and leaf node predictions. Linear models for example look like they would be perfectly interpretable on a modular level, but the interpretation of a single weight is interlocked with all of the other weights. The interpretation of a single weight always comes with the footnote that the other input features stay at the same value, which is not the case in many real world applications. A linear model predicting the value of a house, which takes into account both the size of the house and the number of rooms might have a negative weight for the rooms feature, which is counter intuitive. But it can happen, because there is already the highly correlated flat size feature and in a market where people prefer bigger rooms, a flat with less rooms might be worth more than a flat with more rooms when both have the same size. The weights only make sense in the context of the other features used in the model. But arguably the weights in a linear model still have better interpretability than the weights of a deep neural network.

Local Interpretability for a Single Prediction

Why did the model make a specific decision for an instance?

You can zoom in on a single instance and examine what kind of prediction the model makes for this input, and why it made this decision. When you look at one example, the local distribution of the target variable might behave more nicely. Locally it might depend only linearly or monotonic on some features rather than having a complex dependence on the features. For example the value of an apartment might not depend linearly on the size. But if you only look at a specific apartment of 100 square meters and check how the price changes by going up and down by 10 square meters, there is a chance that this subregion in your data space is linear. Local explanations can be more accurate compared to global explanations because of this. This book presents methods that can make single predictions more interpretable in the section about model-agnostic methods.

Local Interpretability for a Group of Prediction

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

The model predictions for multiple instances can be explained by either using methods for global model interpretability (on a modular level) or single instance explanations. The global methods can be applied by taking the group of instances, pretending it's the complete dataset, and using the global methods on this subset. The single explanation methods can be used on each instance and listed or aggregated afterwards for the whole group.

Evaluating Interpretability

There is no real consensus on what interpretability in machine learning is. Also it is not clear how to measure it. But there is some first research on it and the attempt to formulate some approaches for the evaluation, as described in the following section.

Approaches for Evaluating the Interpretability Quality

Doshi-Velez and Kim (2017) propose three major levels when evaluating interpretability:

- **Application level evaluation (real task)**: Put the explanation into the product and let the end user test it. For example, on an application level, radiologists would test fracture detection software (which includes a machine learning component to suggest where fractures might be in an x-ray image) directly in order to evaluate the model. This requires a good experimental setup and an idea of how to assess the 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 conducted with the domain experts, but with lay humans. This makes experiments less expensive (especially when the domain experts are radiologists) and it is easier to find more humans. An example would be to show a user different explanations and the human would choose the best.
- **Function level evaluation (proxy task)** does not require any humans. This works best when the class of models used was already 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 might be the depth of the tree. Shorter trees would get a better explainability rating. It would make sense to add the constraint that the predictive performance of the tree remains good and does not drop too much compared to a larger tree.

More on Function Level Evaluation

Model size is an easy way to measure explanation quality, but it is too simplistic. For example, a sparse model with features that are themselves not interpretable is still not a good explanation.

There are more dimensions to interpretability:

- Model sparsity: How many features are being used by the explanation?
- Monotonicity: Is there a monotonicity constraint? Monotonicity means that a feature has a monotonic relationship with the target. If the feature increases, the target either always increases or always decreases, but never switches between increasing and decreasing.
- Uncertainty: Is a measurement of uncertainty part of the explanation?
- Interactions: Is the explanation able to include interactions of features?
- Cognitive processing time: How long does it take to understand the explanation?
- Feature complexity: What features were used for the explanation? PCA components are harder to understand than word occurrences, for example.
- Description length of explanation.

Human-friendly Explanations

Let's dig deeper and discover what we humans accept as 'good' explanations and what the implications for interpretable machine learning are.

Research from the humanities can help us to figure that out. Miller (2017) did a huge survey of publications about 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 (just 1 or 2 causes), which contrast the current situation with a situation where the event would not have happened. Especially abnormal causes make good explanations. Explanations are social interactions between the explainer and the explainee (receiver of the explanation) and therefore the social context has a huge influence on the actual content of the explanation.

If you build the explanation system to get ALL the factors for a certain prediction or behaviour, you do not want a human-style explanation, but rather a complete causal attribution. You probably want a causal attribution when you are legally required to state all influencing features or if you are debugging the machine learning model. In this case, ignore the following points. In all other setting, where mostly lay persons or people with little time are the recipients of the explanation, follow the advice here.

What is an explanation?

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

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

The first two kind of questions can be answered with an "everyday"-explanation, while the third one is from the category "More general scientific phenomena and philosophical questions". We focus on the "everyday"-type explanation, because this is relevant for interpretable machine learning. Questions starting with "how" can usually be turned into "why" questions: "How was my loan rejected?" can be turned into "Why was my loan rejected".

The term "explanation" means the social and cognitive process of explaining, but it's also the product of these processes. The explainer can be a human or a machine

What is a "good" explanation?

Now that we know what an explanation is, the question arises, what a good explanation is.

"Many artificial intelligence and machine learning publications and methods claim to be about 'interpretable' or 'explainable' methods, yet often this claim is only based on the authors intuition instead of hard facts and research." - Miller (2017)

Miller (2017) summarises what a 'good' explanation is, which this Chapter replicates in condensed form and with concrete suggestions for machine learning applications.

Explanations are contrastive (Lipton 2016): Humans usually don't ask why a certain prediction was made, but rather why this prediction was made instead of another prediction. We tend to think in counterfactual cases, i.e. "How would the prediction have looked like, if input X were different?". For a house value prediction, a person might be interested in why the predicted price was high compared to the lower price she expected. When my loan application is rejected, I am not interested what in general constitutes a rejection or an approval. I am interested in the factors of my application that would need to change so that it got accepted. I want to know the contrast between my application and the would-be-accepted version of my application. The realisation that contrastive explanations matter, is an important finding for explainable machine learning. As we will see, most interpretable models allow to extract some form of explanation that implicitly contrast it to an artificial data instance or an average of instances. A doctor who wonders: "Why did the treatment not work on the patient?", might ask for an explanation contrastive to a patient, where the treatment worked and who is similar to the non-responsive patient. Contrastive explanations are easier to understand than complete explanations. A complete explanation to the doctor's why question (why does the treatment not work) might include: The patient has the disease already since 10 years, 11 genes are over-expressed making the disease more severe, the patients body is very fast in breaking down the medication into ineffective chemicals , etc.. The contrastive explanation, which answers the question compared to the other patient, for whom the drug worked, might be much simpler: The non-responsive patient has a certain combination of genes, that make the medication much less effective, compared to the other patient. 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 don't want a complete explanation for a prediction but rather compare what the difference were to another instance's prediction (could also be an artificial one). Making explanations contrastive is application dependent, because it requires a point of reference for comparison. And this might depend on the data point to be explained, but also on the user receiving an explanation.

A user of a house price prediction website might want to have an explanation of a house price prediction contrastive to her own house or maybe to some other house on the website or maybe to an average house in the neighbourhood. The solution for creating contrastive explanations in an automated fashion might include finding prototypes or archetypes in the data to contrast to.

Explanations are selected: People don't expect explanations to cover the actual and complete list of causes of an event. We are used to selecting one or two causes from a huge number of possible causes as THE explanation. For proof, switch on the television and watch some news: "The drop in share prices is blamed on a growing backlash against the product due to problems consumers are reporting with the latest software update.", "Tsubasa and his team lost the match because of a weak defence: they left their opponents to much free space to play out their strategy.", "The increased distrust in established institutions and our government are the main factors that reduced voter turnout." The fact that an event can be explained by different causes is called the Rashomon Effect. Rashomon is a Japanese movie in which alternative, contradictory stories (explanations) of a samurai's death are told. For machine learning models it is beneficial, when a good prediction can be made from different features. Ensemble methods can combine multiple models with different features (different explanations) and thrive because averaging over those "stories" makes the predictions more robust and accurate. But it also means that there is no good selective explanation why they made the prediction. **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 type of explanations. If I wanted to explain why digital cryptocurrencies are worth so much, to a technical person I would say things like: "The decentralised, distributed blockchain-based ledger that cannot be controlled by a central entity resonates with people's desire to secure their wealth, which explains the high demand and price.". But to my grandma I might 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**: Be mindful of the social setting of your machine learning application and of the target audience. Getting the social part of the machine learning model right depends completely on your specific application. Find experts from the humanities (e.g. psychologists and sociologists) to help you out.

Explanations focus on the abnormal. People focus more on abnormal causes to explain events (Kahnemann 1981[12]). These are causes, that had a small likelihood but happened anyways (counterfactual explanation). And removing these abnormal causes would have changed the outcome a lot. Humans consider these kinds of "abnormal" causes to be good explanations. An example (Štrumbelj and Kononenko (2011)[13]): Assume that we have a dataset of test situations between teachers and students. The teachers have the option to directly let students pass a course after they have given a presentation or they can ask additional questions to test the student's knowledge, which determines if the student passes. This means we have one feature 'teacher'-feature, which is either 0 (teacher

[12]Kahneman, Daniel, and Amos Tversky. 1981. "The Simulation Heuristic." STANFORD UNIV CA DEPT OF PSYCHOLOGY.

[13]Štrumbelj, Erik, and Igor Kononenko. 2011. "A General Method for Visualizing and Explaining Black-Box Regression Models." In International Conference on Adaptive and Natural Computing Algorithms, 21–30. Springer.

does not test) or 1 (teacher does test). The students can have different levels of preparation (student feature), which translate to different probabilities of correctly answering the teacher's question (in case she decides to test the student). We want to predict if a student will pass the course and explain our prediction. The chance to pass is 100% if the teacher does not ask additional questions, else the probability to pass is according to the student's level of preparation and the resulting probability to correctly answer the questions. Scenario 1: The teacher asks the students additional questions most of the time (e.g. 95 out of 100 times). A student who did not study (10% chance to pass the questions part) was not among the lucky ones and gets additional questions, which he fails to correctly answer. Why did the student fail the course? We would say it was the student's fault to not study. Scenario 2: The teacher rarely asks additional questions (e.g. 3 out of 100 times). For a student who did not learn for possible questions, we would still predict a high probability to pass the course, since questions are unlikely. Of course, one of the students did not prepare for the questions (resulting in a 10% chance to pass the questions). He is unlucky and the teacher asks additional questions, which the student cannot answer and he fails the course. What is the reason for failing? I'd argue that now, the better explanation is that the teacher did test the student, because it was unlikely that the teacher would test. The teacher feature had an abnormal value. **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 predictor example might be that a rather expensive house has three balconies. Even if some attribution method finds out that the three balconies contribute the same price difference as the above average house size, the good neighbourhood and the recent renovation, the abnormal feature "three balconies" might be the best explanation 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 is more important than truthfulness. An explanation that selects only one or two possible causes can never cover the complete list of causes. Selectivity omits part of the truth. It's not true that only one or two factors caused a stock market crash for example, but the truth is that there are millions of causes that influence millions of people to act in a way that caused a crash in the end. **What it means for interpretable machine learning**: The explanation should predict the event as truthfully as possible, which is sometimes called **fidelity** in the context of machine learning. So when we say that three balconies increase the price of a house, it should hold true for other houses as well (or at least for similar houses). To humans, fidelity is not as important for a good explanations as selectivity, contrast and the social aspect.

Good explanations are coherent with prior beliefs of the explainee. Humans tend to ignore information that is not coherent with their prior beliefs. This effect is known as confirmation bias (Nickerson 1998[14]). Explanations are not spared from this type of bias: People will tend to devalue or ignore explanations that do not cohere with their beliefs. This of course differs individually, but there are also group-based prior beliefs like political opinions. **What it means for interpretable machine**

[14]Nickerson, Raymond S. 1998. "Confirmation Bias: A Ubiquitous Phenomenon in Many Guises." Review of General Psychology 2 (2). Educational Publishing Foundation: 175.

learning: Good explanations are consistent with prior beliefs. This one is hard to infuse into machine learning and would probably drastically compromise predictive accuracy. An example would be negative effects of house size for the predicted price of the house for a few of the houses, which, let's assume, improves accuracy (because of some complex interactions), but strongly contradicts prior beliefs. One thing you can do is to enforce monotonicity constraints (a feature can affect the outcome only into one direction) or use something like a linear model that has this property.

Good explanations are general and probable. A cause that can explain a lot of events is very general and could be considered as a good explanation. Note that this contradicts the fact that people explain things with abnormal causes. As I see it, abnormal causes beat general causes. Abnormal causes are, by definition, rare. So in the absence of some abnormal event, a general explanation is judged to be good by humans. Also keep in mind that people tend to judge probabilities of joint events incorrectly. (Joe is a librarian. Is it more likely that he is shy or that he is a shy person that loves reading books?). A good example is 'The bigger a house the more expensive it is', which is a very general, good explanation why houses are expensive or cheap. **What it means for interpretable machine learning**: Generality is easily measured by a feature's support, which is the number of instances for which the explanation applies over the total number of instances.

Datasets

Throughout the book all the models and techniques will be applied on real datasets, which are freely available online. We will be using different datasets for different tasks: classification, regression and text classification.

Bike Sharing Counts (Regression)

This dataset contains daily counts of bike rentals from bike sharing company Capital-Bikeshare[15] in Washington D.C., along with weather and seasonal information. The data was kindly open sourced by Capital-Bikeshare and the folks from Fanaee-T and Gama (2013)[16] have added the weather data and the seasonal information. The goal is to predict how many rental bikes will be out on the street given weather and day. The data can be downloaded from the UCI Machine Learning Repository[17].

For the examples, new features were introduced and not all original features were used. Here is the list of features that were used:

- season : spring (1), summer (2), autumn (3), winter (4).
- holiday : Binary feature indicating if the day was a holiday (1) or not (0).
- yr: The year (2011 or 2012).
- days_since_2011: Number of days since the 01.01.2011 (the first day in the dataset). This feature was introduced to account for the trend, in this case that the bike rental service became more popular over time.
- workingday : Binary feature indicating if the day was a workingday (1) or weekend / holiday (0).
- weathersit : The weather situation on that day
  - Clear, Few clouds, Partly cloudy, Cloudy
  - Mist + Cloudy, 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 + Fog
- temp : Temperature in degrees Celsius.
- hum: Relative humidity in percent (0 to 100).
- windspeed: Wind speed in km per hour.
- cnt: Count of total rental bikes including both casual and registered. The count was used as the target in the regression tasks.

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

[16] Fanaee-T, Hadi, and Joao Gama. 2013. "Event Labeling Combining Ensemble Detectors and Background Knowledge." Progress in Artificial Intelligence. Springer Berlin Heidelberg, 1–15. doi:10.1007/s13748-013-0040-3.

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

YouTube Spam Comments (Text Classification)

As an example for text classification we will be using 1956 comments from 5 different YouTube videos. Thankfully the authors that used this dataset in an article about spam classification made the data freely available[18] (Alberto, Lochter, and Almeida 2015[19]).

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

You can flip through some of the comments. The comments had been hand labeled as spam or legitimate. Spam has been 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 over to YouTube and have a look at the comment section. But please don't get trapped in the YouTube hell, ending up watching videos about monkeys stealing and drinking cocktails from tourists on the beach. Also the Google Spam detector probably has changed a lot since 2015.

Watch the view-record breaking video "Gangnam Style" here[20]

Risk Factors for Cervical Cancer (Classification)

The cervical cancer dataset contains indicators and risk factors for predicting if a woman will get cervical cancer. The features contain demographics (e.g. age), habits, and medical history. The data can be downloaded from the UCI Machine Learning repository[21] and is described by K. Fernandes,

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

[19] Alberto, Túlio C, Johannes V Lochter, and Tiago A Almeida. 2015. "Tubespam: Comment Spam Filtering on Youtube." In Machine Learning and Applications (Icmla), 2015 Ieee 14th International Conference on, 138–43. IEEE.

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

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

Cardoso, and Fernandes (2017) [22].

The subset of features, which are used in the examples are:

- Age in years
- Number of sexual partners
- First sexual intercourse (age in years)
- Number of pregnancies
- Smokes yes (1) or no (1)
- Smokes (years)
- Hormonal Contraceptives yes (1) or no (0)
- Hormonal Contraceptives (years)
- IUD: Intrauterine device yes (1) or no (1)
- IUD (years): Number of years with an intrauterine device
- STDs: Ever had a sexually transmitted disease? Yes (1) or no (0)
- STDs (number): Number of sexually transmitted diseases.
- STDs: Number of diagnosis
- STDs: Time since first diagnosis
- STDs: Time since last diagnosis
- Biopsy: Biopsy results "Healthy" or "Cancer". Target outcome.

As the biopsy serves as the gold standard for diagnosing cervical cancer, the classification task in this book used the biopsy outcome as the target. Missing values for each column were imputed by the mode (most frequent value), which is probably a bad solution, because the value of the answer might be correlated with the probability for a value being 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 suffice!

[22] Fernandes, Kelwin, Jaime S Cardoso, and Jessica Fernandes. 2017. "Transfer Learning with Partial Observability Applied to Cervical Cancer Screening." In Iberian Conference on Pattern Recognition and Image Analysis, 243–50. Springer.

Interpretable Models

The most straightforward way to get to interpretable machine learning is to use only a subset of algorithms that create interpretable models.

Very common model types of this group of interpretable models are:

- Linear regression model.
- Logistic regression.
- Decision trees.

In the following chapters we will talk about these models. Not in detail, only the basics, because there are already a ton of books, videos, tutorials, papers and more material. We will focus on how to interpret the models. This chapter covers linear models, logistic regression, and decision trees in more detail. It also lists some more.

All of the interpretable model types 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 over the interpretable model types and their properties. A model is linear if the association between features and target is modeled linearly. A monotonic model ensures that the relationship between a feature and the target outcome is always in the same direction over the whole range of the feature: an increase in the features value will consistently lead to either an increase or a decrease of the target outcome, but never both for this feature. 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 the features for predicting the outcome. You can always include interactions into any kind of model by manually creating interaction features. This can be important for correctly predicting the outcome, but too many or too complex interactions can hurt interpretability. Some models only handle regression, some only classification, and some can manage to do both.

You can use this table to choose a suitable interpretable model for your task (either regression (regr.) or classification (class.)):

Algorithm	Linear	Monotone	Interaction	Task
Linear models	Yes	Yes	No	Regr.
Logistic regression	No	Yes	No	Class.
Decision trees	No	No	Yes	Class. + Regr.
RuleFit	Yes	No	Yes	Class. + Regr.
Naive Bayes	Yes	Yes	No	Class.n
k-nearest neighbours	No	No	No	Class. + Regr.

Terminology

- Y is the target outcome.
- X are the features (also called variables, covariables, covariates, or inputs).
- β are regression weights (also called coefficients).

Linear Model

Linear models have been used for a long time by statisticians, computer scientists, and other people tackling quantitative problems. Linear models learn linear (and therefore monotonic) relationships between the features and the target. The linearity of the learned relationship makes the interpretation easy.

Linear models can be used to model the dependency of a regression target y on p features x. The learned relationships are linear and, for a singular instance i, can be written as:

$$y_i = \beta_0 + \beta_1 x_{i1} + \ldots + \beta_p x_{ip} + \epsilon_i$$

The i-th instance's outcome is a weighted sum of its p features. The β_j represent the learned feature weights or coefficients. The ϵ_i is the error we are still making, i.e. the difference between the predicted outcome and the actual outcome.

Different methods can be used to estimate the optimal weight vector $\hat{\boldsymbol{\beta}}$. The ordinary least squares method is commonly used to find the weights that minimise the squared difference between the actual and the estimated outcome:

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

We won't go into detail about how the optimal weights can be found, but if you are interested you can read Chapter 3.2 of the book "Elements of Statistical Learning" (Hastie, Tibshirani, and Friedman 2009)[23] or one of the other zillions of sources about linear regression models.

The biggest advantage of linear regression models is their linearity: It makes the estimation procedure straightforward and, most importantly, these linear equations have an easy to understand interpretation on a modular level (i.e. the weights). That is one of the main reasons why the linear model and all similar models are so widespread in academic fields like medicine, sociology, psychology, and many more quantitative research fields. In these areas it is important to not only predict, e.g., the clinical outcome of a patient, but also to quantify the influence of the medication while at the same time accounting for things like gender, age, and other features in an interpretable manner.

Linear regression models also come with some assumptions that make them easy to use and interpret but which are often not satisfied in reality. The assumptions are: Linearity, normality, homoscedasticity, independence, fixed features, and absence of multicollinearity.

- **Linearity**: Linear regression models force the estimated response to be a linear combination of the features, which is both their greatest strength and biggest limitation. Linearity leads to

[23] Hastie, T, R Tibshirani, and J Friedman. 2009. The elements of statistical learning. http://link.springer.com/content/pdf/10.1007/978-0-387-84858-7.pdf.

interpretable models: linear effects are simple to quantify and describe (see also next chapter) and are additive, so it is easy to separate the effects. If you suspect interactions of features or a non-linear association of a feature with the target value, then you can add interaction terms and use techniques like regression splines to estimate non-linear effects.

- **Normality**: The target outcome given the features are assumed to follow a normal distribution. If this assumption is violated, then the estimated confidence intervals of the feature weights are not valid. Consequently, any interpretation of the features p-values is not valid.
- **Homoscedasticity** (constant variance): The variance of the error terms ϵ_i is assumed to be constant over the whole feature space. Let's say you want to predict the value of a house given the living area in square meters. You estimate a linear model, which assumes that no matter how big the house, the error terms around the predicted response have 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 bigger houses, since also the prices are higher and there is more room for prices to vary.
- **Independence**: Each instance is assumed to be independent from the next one. If you have repeated measurements, like multiple records per patient, the data points are not independent from each other and there are special linear model classes to deal with these cases, like mixed effect models or GEEs.
- **Fixed features**: The input features are seen as 'fixed', carrying no errors or variation, which, of course, is very unrealistic and only makes sense in controlled experimental settings. But not assuming fixed features would mean that you have to fit very complex measurement error models that account for the measurement errors of your input features. And usually you don't want to do that.
- **Absence of multicollinearity**: Basically you don't want features to be highly correlated, because this messes up the estimation of the weights. In a situation where two features are highly correlated (something like correlation > 0.9) it will become problematic to estimate the weights, since 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 model depends on the type of the corresponding feature:

- Numerical feature: For an increase of the numerical feature x_j by one unit, the estimated outcome changes by β_j. An example of a numerical feature is the size of a house.
- Binary feature: A feature, that for each instance takes on one of two possible values. An example is the feature "House comes with a garden". One of the values counts as the reference level (in some programming languages coded with 0), like "No garden". A change of the feature x_j from the reference level to the other level changes the estimated outcome by β_j.
- Categorical feature with multiple levels: A feature with a fixed amount of possible values. An example is the feature "Flooring type", with possible levels "carpet", "laminate" and "parquet". One solution to deal with many levels is to one-hot-encode them, meaning each level gets its

own binary column. From a categorical feature with l levels, you only need $l - 1$ columns, otherwise the coding is overparameterised. The interpretation for each level is then according to the binary features. Some languages, like R, allow you to code categorical features in different ways, described here.

- 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 feature for estimating the intercept. The interpretation is: Given all numerical features are zero and the categorical features are at the reference level, the estimated outcome of y_i is β_0. The interpretation of β_0 is usually not relevant, because instances with all features at zero often don't make any sense, unless the features were standardised (mean of zero, standard deviation of one), where the intercept β_0 reflects the predicted outcome of an instance where all features are at their mean.

Another important measurement for interpreting linear models is the R^2 measurement. R^2 tells you how much of the total variance of your target outcome is explained by the model. The higher R^2 the better your model explains the data. The formula to calculate R^2 is: $R^2 = 1 - SSE/SST$, where SSE is the squared sum of the error terms:

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

and 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 looking at the squared differences between the predicted and actual target values. SST is the total variance of the target around the mean. So R^2 tells you how much of your variance can be explained by the linear model. R^2 ranges between 0 for models that explain nothing and 1 for models that explain all of the variance in your data.

There is a catch, because R^2 increases with the number of features in the model, even if they carry no information about the target value at all. So it is better to use the adjusted R-squared ($\bar{R}^2$), 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 isn't helpful to do interpretation on a model with very low R^2 or $\bar{R}^2$, because basically the model is not explaining much of the variance, so any interpretation of the weights are not meaningful.

Interpretation Example

In this example we use the linear model to predict the bike rentals on a day, given weather and calendrical information. For the interpretation we examine the estimated regression weights. The features are a mix of numerical and categorical features. The table shows for each feature the estimated weight and the standard error of the estimation:

	Weight estimate	Std. Error
(Intercept)	2399.4	238.3
seasonSUMMER	899.3	122.3
seasonFALL	138.2	161.7
seasonWINTER	425.6	110.8
holidayHOLIDAY	-686.1	203.3
workingdayWORKING DAY	124.9	73.3
weathersitMISTY	-379.4	87.6
weathersitRAIN/SNOW/STORM	-1901.5	223.6
temp	110.7	7.0
hum	-17.4	3.2
windspeed	-42.5	6.9
days_since_2011	4.9	0.2

Interpretation of a numerical feature ('Temperature'): An increase of the temperature by 1 degree Celsius increases the expected number of bikes by 110.7, given all other features stay the same.

Interpretation of a categorical feature ('weathersituation')): The estimated number of bikes is -1901.5 lower when it is rainy, snowing or stormy, compared to good weather, given that all other features stay the same. Also if the weather was misty, the expected number of bike rentals was -379.4 lower, compared to good weather, given all other features stay the same.

As you can see in the interpretation examples, the interpretations always come with the footnote that 'all other features stay the same'. That's because of the nature of linear models: The target is a linear combination of the weighted features. The estimated linear equation spans a hyperplane in the feature/target space (a simple line in the case of a single feature). The β (weight) values specify the slope (gradient) of the hyperplane in each direction. The good side is that it isolates the interpretation. If you think of the features as knobs that you can turn up or down, it is nice to see what happens when you would just turn the knob for one feature. On the bad side of things, the interpretation ignores the joint distribution with other features. Increasing one feature, but not changing others, might create unrealistic, or at least unlikely, data points.

Interpretation templates

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

Interpretation of a Numerical Feature

An increase of x_k by one unit increases the expectation for y by β_k units, given all other features stay the same.

Interpretation of a Categorical Feature

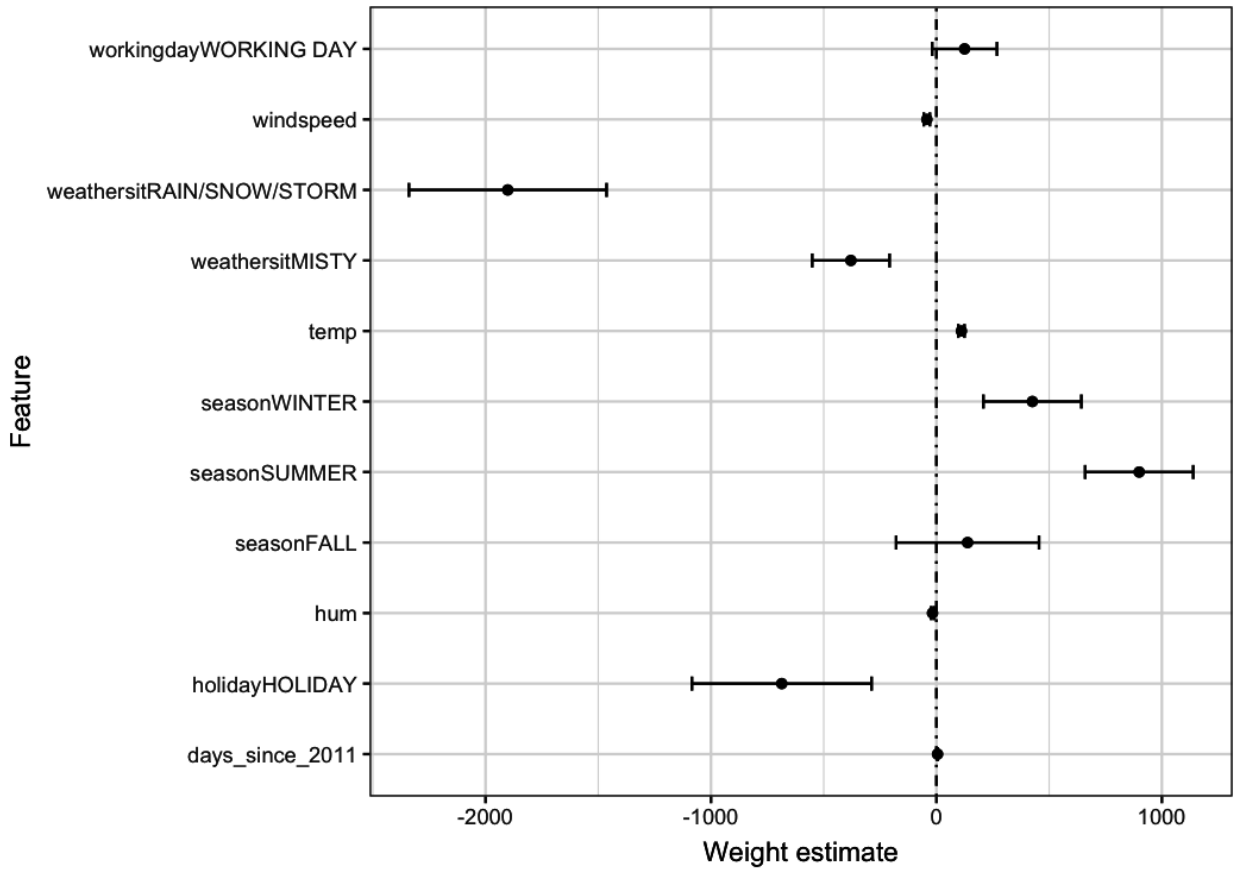
A change from x_k's reference level to the other category increases the expectation for y by β_k, given all other features stay the same.

Visual parameter interpretation

Different visualisations make the linear model outcomes easy and quick to grasp for humans.

Weight plot

The information of the weights table (weight estimates and variance) can be visualised in a weight plot (showing the results from the linear model fitted before):

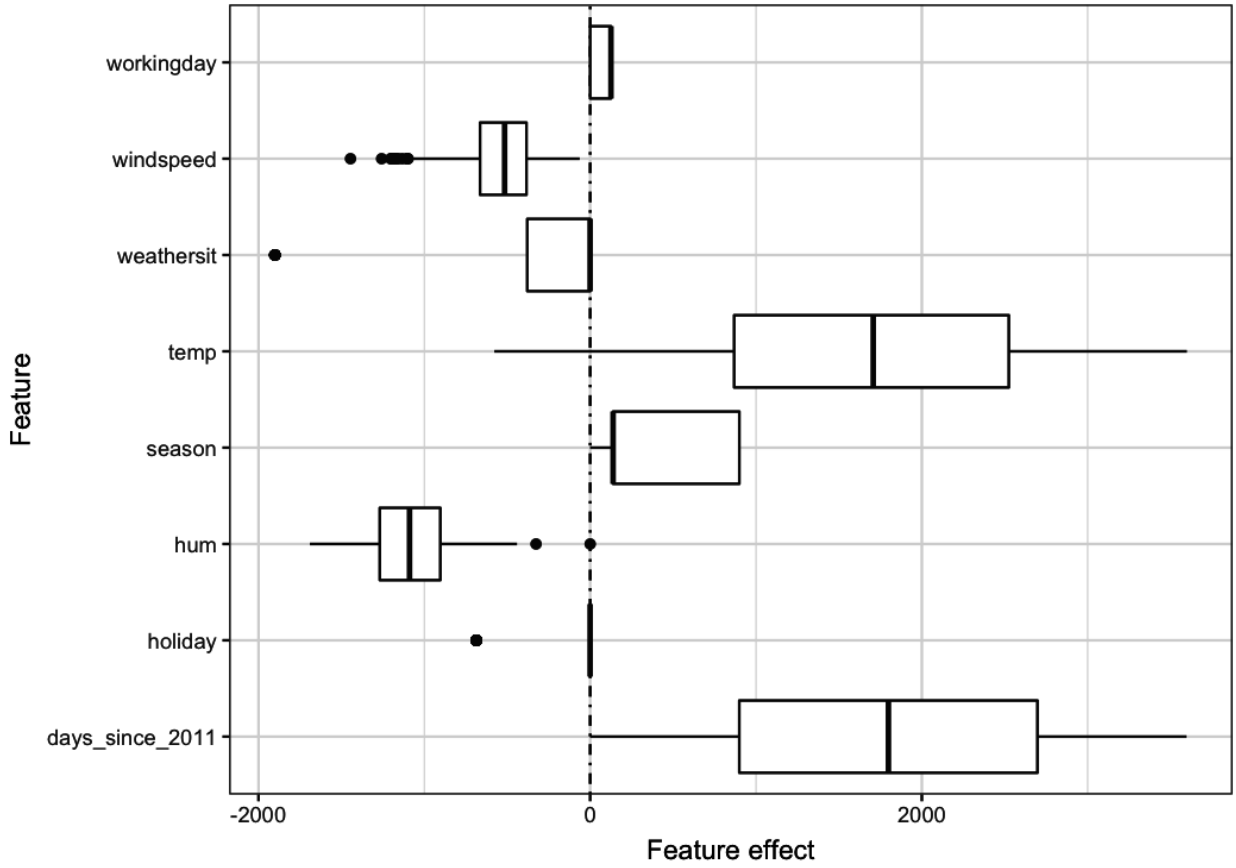


Each row in the plot represents one feature weight. The weights are displayed as points and the 0.95 confidence intervals with a line around the points. A 0.95 confidence interval means that if the linear model would be estimated 100 times on similar data, in 95 out of 100 times, the confidence interval would cover the true weight, under the linear model assumptions (linearity, normality, homoscedasticity, independence, fixed features, absence of multicolinearity).

The weight plot makes clear that rainy/snowy/stormy weather has a strong negative effect on the expected number of bikes. The working day feature's weight is close to zero and the zero is included in the 95% interval, meaning it is not influencing the prediction significantly. Some confidence intervals are very short and the estimates are close to zero, yet the features were important. Temperature is such a candidate. The problem about the weight plot is that the features are measured on different scales. While for weather situation feature the estimated β signifies the difference between good and rainy/storm/snowy weather, for temperature it signifies only an increase of 1 degree Celsius. You can improve the comparison by scaling the features to zero mean and unit standard deviation before fitting the linear model, to make the estimated weights comparable.

Effect Plot

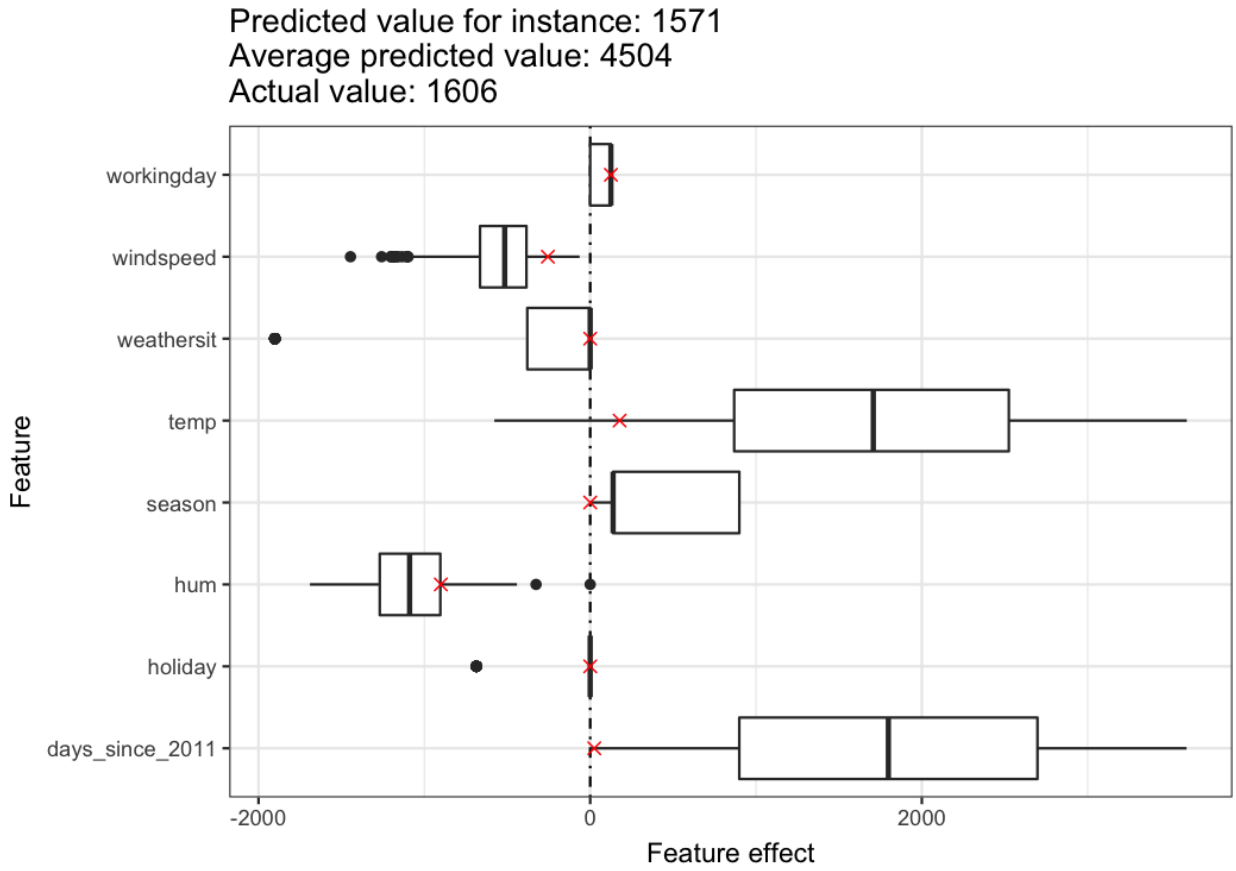
The weights of the linear model can be analysed more meaningfully when multiplied by the actual feature values. The weights depend on the scale of the features and will be different if you have a feature measuring some height and you switch from meters to centimetres. The weight will change, but the actual relationships 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 will get a similar contribution from this feature. The effect plot can help to understand how much the combination of a weight and a feature contributes to the predictions in your data. Start with the computation of the effects, which is the weight per feature times the feature of an instance: $\text{effect}_{i,j} = w_j x_{i,j}$. The resulting effects are visualised 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 than the median value. The horizontal lines extend to$\pm 1.58 \text{IQR}/\sqrt{n}$, with IQR being the inter quartile range ($q_{0.75} - q_{0.25}$). The points are outliers. The categorical feature effects can be aggregated into one boxplot, compared to the weight plot, where each weight gets a row.



The feature effect plot shows the distribution of the effects (= feature value times feature weight) over the dataset for each feature.

Explaining Single Predictions

How much did each feature of an instance contribute towards the prediction? This can, again, be answered by bringing together the weights and feature values of this instance and computing the effects. An interpretation of instance specific effects is only meaningful in comparison with the distribution of each feature's effects.



The effect for one instance shows the effect distribution while highlighting the effects of the instance of interest.

Let's have a look at the effect realisation for the rental bike count of one instance (i.e. one day). Some features contribute unusually little or much to the predicted bike count when compared to the overall dataset: Temperature (2 degrees) contributes less towards the predicted value compared to the average and the trend feature "days_since_2011" unusually much, because this instance is from late 2011 (5 days).

Coding Categorical Features

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

The standard in linear regression models is the treatment coding, which is sufficient in most cases. Using different codings boils down to creating different matrices (=design matrix) from your one column with the categorical feature. This section presents three different codings, but there are many more. The example used has six instances and one categorical feature with 3 levels. For the first two instances, the feature takes on category A, for instances three and four category B and for the last two instances category C.

Treatment coding:

The β per level is the estimated difference in y compared to the reference level. The intercept of the linear model is the mean of the reference group (given all other features stay the same). The first column of the design matrix is the intercept, which is always 1. Column two is an indicator whether instance i is in category B, column three is an indicator for category C. There is no need for a column for category A, because then the linear equation would be overspecified and no unique solution (= unique β's) can be found. Knowing that an instance is neither in category B or C is enough.

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 β per level is the estimated y-difference from the level to the overall mean (again, given all other features are zero or the reference level). The first column is again used to estimate the intercept. The weight β_0 which is 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 overall 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 level is the estimated mean of y for each level (given all feature are at value zero or reference level). Note that the intercept was dropped here, so that a unique solution for the linear model weights can be found.

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 bit deeper into different encodings of categorical features, checkout this webpage[24] and this blog post[25].

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

The disadvantages of linear models

Linear models can only represent linear relationships. Each **non-linearity 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 oversimplifies 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 area of the house. House area and number of rooms are highly correlated: the bigger a house is, the more rooms it has. If you now take both features into a linear model, it might happen, that the area of the house is the better predictor and get's a large positive weight. The number of rooms might end up getting a negative weight, because either, 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.

Do linear models create good explanations?

Judging by the attributes that constitute a good explanation as presented in this section, linear models don't create the best explanations. They are contrastive, but the reference instance is a data point for which all continuous features are zero and the categorical features at their reference levels. This is usually an artificial, meaningless instance, which is unlikely to occur in your dataset. There is an exception: When all continuous features are mean centered (feature minus mean of feature) and all categorical features are effect coded, the reference instance is the data point for which each feature is at its mean. This might also be a non-existent data point, but it might be at least more likely or meaningful. In this case, the β-values times the feature values (feature effects) explain the contribution to the predicted outcome contrastive to the reference mean-instance. Another aspect of a good explanation is selectivity, which can be achieved in linear models by using less features or by fitting sparse linear models. But by default, linear models don't create selective explanations. Linear models create truthful explanations, as long as the linear equation can model the relationship between features and outcome. The more non-linearities and interactions exist, the less accurate the linear model becomes and the less truthful explanations it will produce. The linearity makes the explanations more general and simple. The linear nature of the model, I believe, is the main factor why people like linear models for explaining relationships.

Extending Linear Models

Linear models have been extensively studied and extended to fix some of the shortcomings.

- Lasso is a method to "pressure" weights of irrelevant features to get an estimate of zero. Having unimportant features weighted by zero is useful, because having less terms to interpret makes the model more interpretable.
- Generalised linear models (GLM) allow the target outcome to have different distributions. The target outcome is not any longer required to be normally distributed given the features, but GLMs allow you to model for example Poisson distributed count variables. Logistic regression, is a GLM for categorical outcomes.
- Generalised additive models (GAMs) are GLMs with the additional ability to allow non-linear relationships with features, while maintaining the linear equation structure (sounds paradox, I know, but it works).

You can also apply all sorts of tricks to go around some of the problems:

- Adding interactions: You can define interactions between features and add them as new features before estimating the linear model. The RuleFit algorithm can add interactions automatically.
- Adding non-linear terms like polynomials to allow non-linear relationships with features.
- Stratifying data by feature and fitting linear models on subsets.

Sparse linear models

The examples for the linear models that I chose look all nice and tidy, right? But in reality you might not have just a handful of features, but hundreds or thousands. And your normal linear models? Interpretability goes downriver. You might even get into a situation with more features than instances and you can't fit a standard linear model at all. The good news is that there are ways to introduce sparsity (= only keeping a few features) into linear models.

Lasso

The most automatic and convenient way to introduce sparsity is to use the Lasso method. Lasso stands for "least absolute shrinkage and selection operator" and when added to a linear model, it performs feature selection and regularisation of the selected feature weights. Let's review the minimization problem, the βs optimise:

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

Lasso adds a term to this optimisation 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 penalisation of large $\boldsymbol{\beta}$-values. Since the L1-norm is used, many of the weights for the features will get an estimate of 0 and the others are shrunk. The parameter λ controls the strength of the regularising effect and is usually tuned by doing cross-validation. Especially when λ is large, many weights become 0.

Other Methods for Sparsity in Linear Models

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

Methods that include a pre-processing step:

- Hand selected features: You can always use expert knowledge to choose and discard some features. The big drawback is, that it can't be automated and you might not be an expert.
- Use some measures to pre-select features: An example is the correlation coefficient. You only take features into account that exceed some chosen threshold of correlation between the feature and the target. Disadvantage is that it only looks at the features one at a time. Some features might only show correlation after the linear model has accounted for some other features. Those you will miss with this approach.

Step-wise procedures:

- Forward selection: Fit the linear model with one feature. Do that with each feature. Choose the model that works best (for example decided by the highest R squared). Now again, for the remaining features, fit different versions of your model by adding each feature to your chosen model. Pick the one that performs best. Continue until some criterium is reached, like the maximum number of features in the model.
- Backward selection: Same as forward selection, but instead of adding features, start with the model that includes all features and try out which feature you have to remove to get the highest performance increase. Repeat until some stopping criterium is reached.

I recommend using Lasso, because it can be automated, looks at all features at the same time and can be controlled via λ. It also works for the logistic regression model for classification.

Logistic Regression

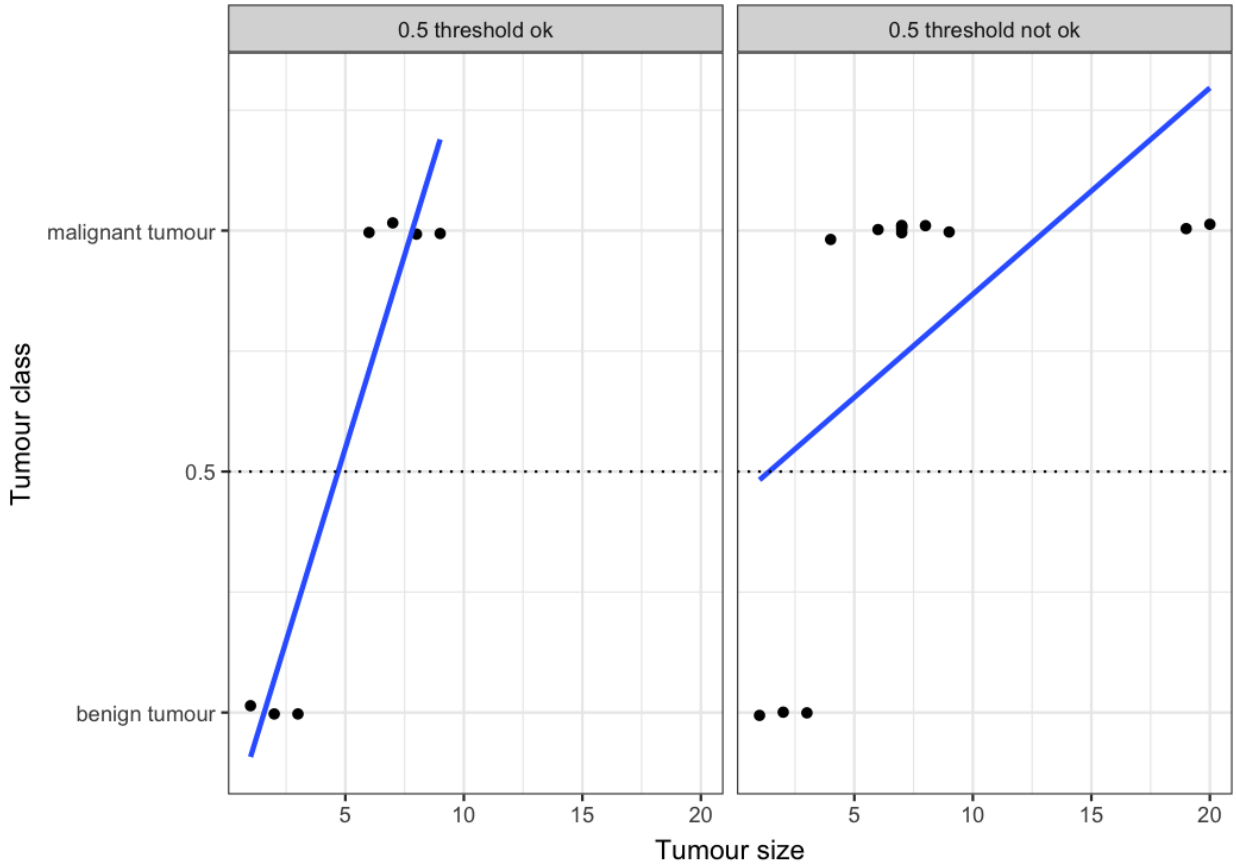
Logistic regression is the linear regression model made fit for classification problems.

What's Wrong with Linear Regression Models for Classification?

The linear regression model works well in regression setups, but fails in the classification case. Why is that? In case of two classes, you could label one of the classes with 0 and the other with 1 and use a linear model on it and it would estimate the weights for you. There are just a few problems with that approach:

- A linear model does not output probabilities, but it treats the classes as numbers (0 and 1) and fits the best hyperplane (if you have one feature, it's a line) that minimises the distances between the points and the hyperplane. So it simply interpolates between the points, but there is no meaning in it and you cannot interpret it as probabilities.
- Also a linear model will extrapolate the features and give you values below zero and above one, which are not meaningful and should tell you that there might be a more clever approach to classification.
- Since the predicted outcome is not a probability but some 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 was given on Stackoverflow[26]
- Linear models don't extend to classification problems with multiple classes. You would have to start labeling the next class with a 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. So for a feature with a positive weight, the higher the value of that feature the more it contributes to the prediction of a class with a higher number, even if classes that happened to get a similar number are not related at all.

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



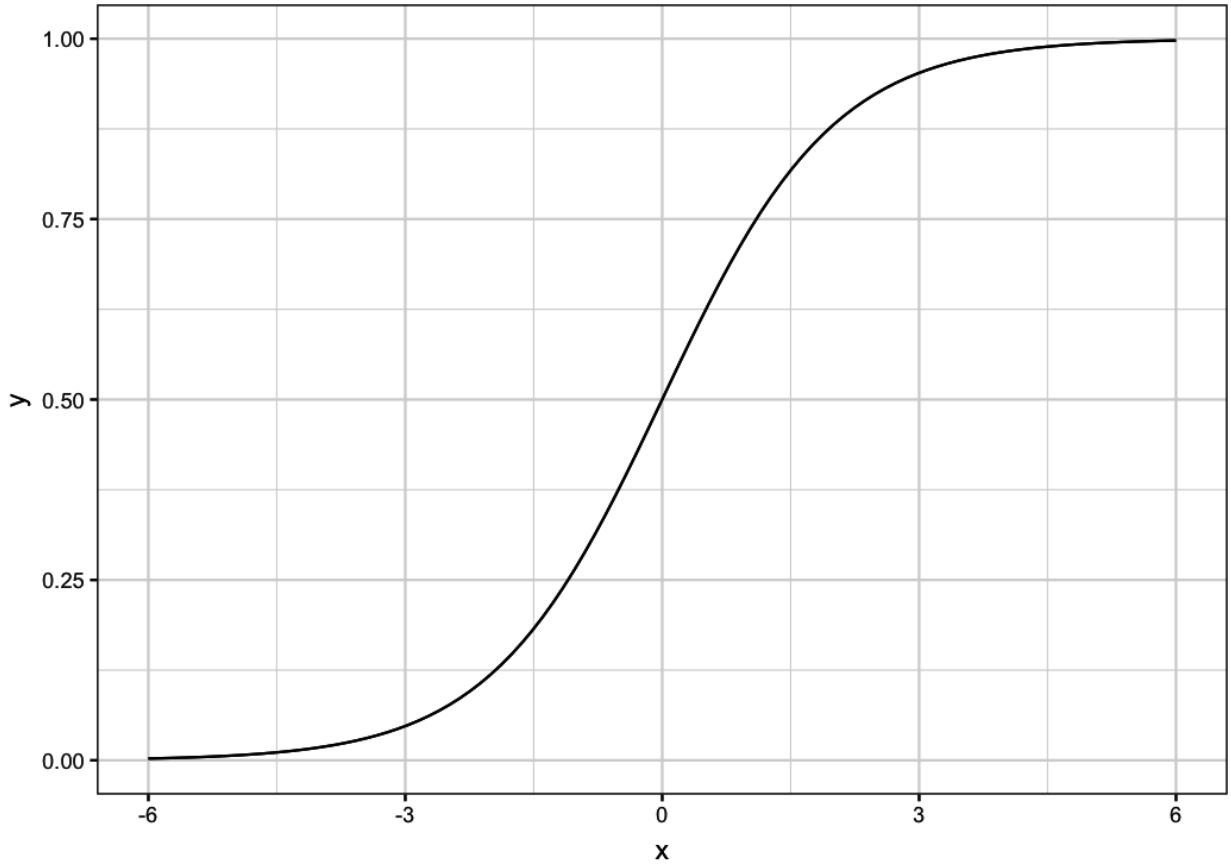
An illustration why linear regression does not work well in a binary classification setting. A linear model is fitted on artificial data for classifying a tumour as malignant (1) or benign (0), dependant on the size of the tumour. Each point is a tumour, the x-axis shows the size of the tumour, the y-axis the malignancy, points are slightly jittered to reduce over-plotting. The lines display the fitted curve from a linear model. In the data setting on the left, we can use 0.5 as a threshold for the predicted outcome of the linear model for separating benign from malignant tumours. After introducing a few more malignant tumour cases, especially with larger tumour sizes, the regression line shifts and a threshold of 0.5 does not separate the classes any longer.

Logistic Regression

A solution for classification is logistic regression. Instead of fitting a straight line or hyperplane, the logistic regression model uses a non-linear function, 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 only outputs numbers between 0 and 1. At input 0 it outputs 0.5.

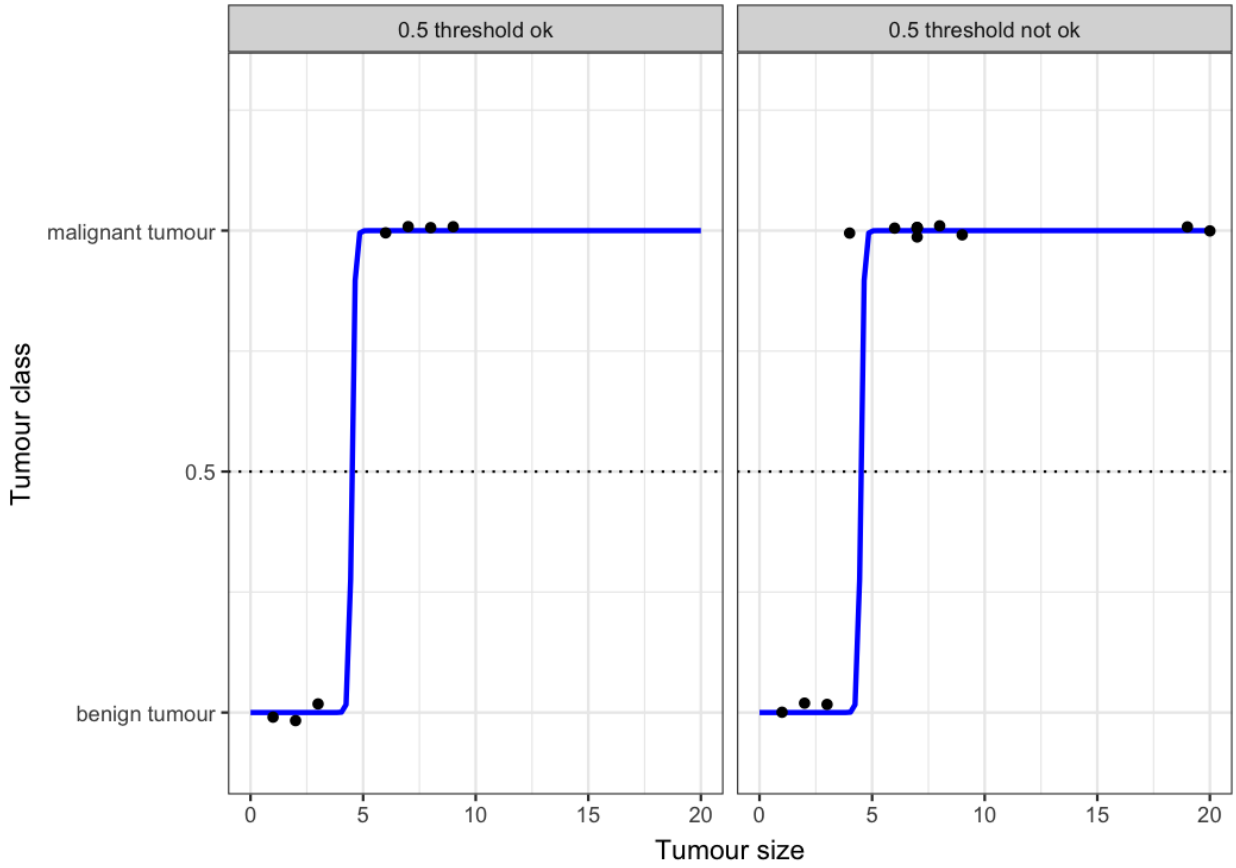
The step from linear regression models to logistic regression is kind of straightforward. In the linear regression model we modelled the relationship between the outcome and the features with a linear equation:

$$\hat{y}_i = \beta_0 + \beta_1 x_{i,1} + \ldots + \beta_p x_{i,p}$$

For the classification we prefer probabilities, which are between 0 and 1, so we wrap the right side of the equation into the logistic regression function and like that force the output to only take on values between 0 and 1.

$$P(y_i = 1) = \frac{1}{1 + exp(-(\beta_0 + \beta_1 x_{i,1} + \ldots + \beta_p x_{i,p}))}$$

Let's revisit the tumour size example again. But instead of the linear regression model, we use the logistic regression model:



The logistic regression model successfully finds the correct decision boundary to distinguish between malignant and benign tumours dependent on the size of the tumour in this example. The blue line is the logistic function shifted and squeezed so that it fits the data.

Interpretation

The interpretation of the logistic regression weights differs from the linear regression case, because in logistic regression the outcome is a probability between 0 and 1, and the weights don't affect the probability linearly, but are squeezed through the logistic function. That's why we need to reformulate the equation for the interpretation, so that there is only the linear term left on the right side of the formula.

$$log\left(\frac{P(y_i=1)}{1-P(y_i=1)}\right) = log\left(\frac{P(y_i=1)}{P(y_i=0)}\right) = \beta_0 + \beta_1 x_{i,1} + \ldots + \beta_p x_{i,p}$$

We call the inner term odds (probability of event divided by probability of no event): $\frac{P(y_i=1)}{1-P(y_i=1)}$ and wrapped in the logarithm it is called log odds:

$$log\left(\frac{P(y_i=1)}{1-P(y_i=1)}\right)$$

So with a logistic regression model we have a linear model for the log odds. Great! Doesn't sound helpful! Well, with a bit of shuffling again, you can find out how the prediction changes, when one of the features x_j is changed by 1 point. For this we can first apply the $exp()$ function on both sides of the equation:

$$\frac{P(y_i = 1)}{(1 - P(y_i = 1))} = odds_i = exp\left(\beta_0 + \beta_1 x_{i,1} + \ldots + \beta_p x_{i,p}\right)$$

Then we compare what happens when we increase one of the $x_{i,j}$ by 1. But instead of looking at the difference, we look at the ratio of the two predictions:

$$\frac{odds_{i,x_j+1}}{odds_i} = \frac{exp\left(\beta_0 + \beta_1 x_{i,1} + \ldots + \beta_j(x_{i,j} + 1) + \ldots + \beta_p x_{i,p}\right)}{exp\left(\beta_0 + \beta_1 x_{i,1} + \ldots + \beta_j x_{i,j} + \ldots + \beta_p x_{i,p}\right)}$$

Using the rule that:

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

and removing lots of terms gives us:

$$\frac{odds_{i,x_j+1}}{odds_i} = exp\left(\beta_j(x_{i,j} + 1) - \beta_j x_{i,j}\right) = exp\left(\beta_j\right)$$

And we end up with something simple like $\exp(\beta_j)$. So a change of x_j by one unit changes the odds ratio (multiplicatively) by a factor of $\exp(\beta_j)$. We could also interpret it this way: A change in x_j by one unit changes the log odds ratio by β_j units, but most people do the former because thinking about the $log()$ of something is known to be hard on the brain. Interpreting the odds ratio already needs a bit of getting used to. For example if you have odds of 2, it means that the probability for $y_i = 1$ is twice as big as $y_i = 0$. If you have a β_j (=odds ratio) of 0.7, then an increase in the respective x_j by one unit multiplies the odds by $\exp(0.7) \approx 2$ and the odds change to 4. But usually you don't deal with the odds and only interpret the β's as the odds ratios. Because for actually calculating the odds you would need to set a value for each feature x_j, which only makes sense if you want to look at one specific instance of your dataset.

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

- Numerical feature: For an increase of one unit of the feature x_j, the estimated odds change (multiplicatively) by a factor of $\exp(\beta_j)$
- Binary categorical feature: One of the two values of the feature is the reference level (in some languages the one that was coded in 0). A change of the feature x_j from the reference level to the other level changes the estimated odds (multiplicatively) by a factor of $\exp(\beta_j)$
- Categorical feature with many levels: One solution to deal with many possible feature values is to one-hot-encode them, meaning each level gets its own column. From a categorical feature

with L levels, you only need L-1 columns, otherwise it is over-parameterised. The interpretation for each level is then according to the binary features.

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

Example

We use the logistic regression model to predict cervical cancer given 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 of the number of diagnosed STDs (sexually transmitted diseases) changes (decreases) the odds for cancer vs. no cancer multiplicatively by 0.44, given all other features stay the same. Keep in mind that correlation does not imply causation. No recommendation here to get STDs.

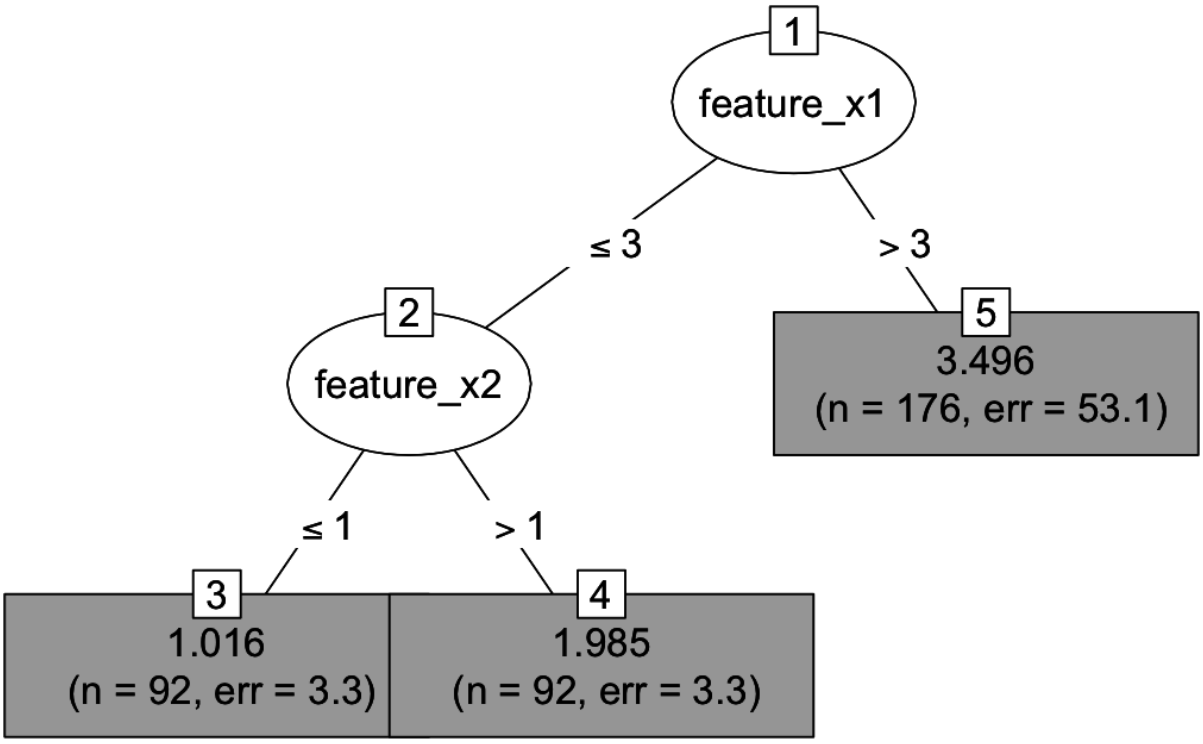
Interpretation of a categorical feature ('Hormonal contraceptives y/n'): For women with hormonal contraceptives, the odds for cancer vs. no cancer are by a factor of 1.12 higher, compared to women without hormonal contraceptives, given all other features stay the same.

Again as in the linear models, the interpretations are always coming with the clause that 'all other features stay the same'.

Decision Tree

Linear regression models and logistic regression fail in situations where the relationship between features and outcome is non-linear or where the features are interacting with each other. Time to shine for the decision trees! Tree-based models split the data according to certain cutoff values in the features multiple times. Splitting means that different subsets of the dataset are created, where each instance belongs to one subset. The final subsets are called terminal or leaf nodes and the intermediate subsets are called internal nodes or split nodes. For predicting the outcome in each leaf node, a simple model is fitted with the instances in this subset (for example the subsets average target outcome). Trees can be used for classification and regression.

There are a lot of tree algorithms with different approaches for how to grow a tree. They differ in the possible structure of the tree (e.g. number of splits per node), criteria for how to find the splits, when to stop splitting and how to estimate the simple models within the leaf nodes. Classification and regression trees (CART) is one of the more popular algorithms for tree induction. We will focus on CART, but the interpretation is similar for most of the tree types. I recommend the book 'The elements of statistical learning' (Hastie, Tibshirani, and Friedman 2009)[27] for a more detailed introduction.



[27]Hastie, T, R Tibshirani, and J Friedman. 2009. The elements of statistical learning. http://link.springer.com/content/pdf/10.1007/978-0-387-84858-7.pdf.

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

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

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

But where do the subsets come from? This is quite simple: The algorithm takes a feature and tries which cut-off point minimises the sum of squares of y for a regression tasks or the Gini index of the class distribution of y for classification tasks. The best cut-off point makes the two resulting subsets as different as possible in terms of the target outcome. For categorical features the algorithm tries to build subsets by trying different groupings of categories. After this was done for each feature, the algorithm looks for the feature with the best cut-off and chooses it to split the node into two new nodes. The algorithm continues doing this recursively in both of the new nodes until a stopping criterium 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.

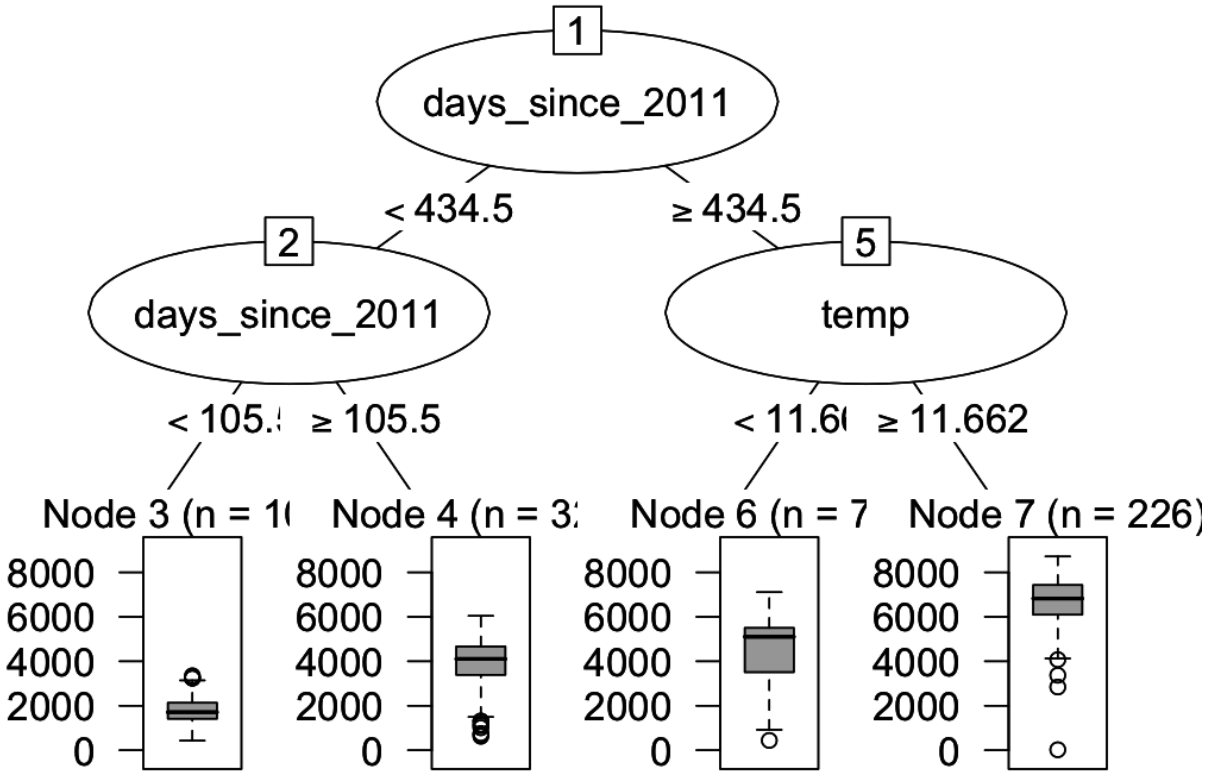
Interpretation

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 $\hat{y}_{\text{leafnode}}$.

Interpretation Example

Let's have a look again at the bike rental data. We want to predict the number of bike rentals on a given day. The learned tree visualized:



Regression tree fitted on the bike rental data. The maximally allowed depth for the tree was set to 2. The features picked for the tree splits were the trend feature (days since 2011) and the temperature (temp). The boxplots show the distribution of bike rentals in the terminal node.

Advantages

The tree structure is perfectly suited to **cover interactions** between features in the data. The data also ends up in **distinct groups**, which are often easier to grasp than points on a hyperplane like in linear regression. The interpretation is arguably pretty straightforward. The tree structure also has a **natural visualization**, with its nodes and edges. Trees **create good explanations** as defined here. The tree structure automatically invites to think about predicted values for single instances in a counterfactual way: "If feature x_j would have been bigger / smaller than the split point, the prediction would have been $\hat{y}_1$ instead of $\hat{y}_2$" The created explanations are contrastive, because you can always compare the prediction of an instance with relevant (as defined by the tree) "what-if"-scenarios, which 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 needs a maximum of three features and split points to create the explanation for the prediction of an 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 either falls into one or the other leave., 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 can handle a feature regardless of monotonic transformations.

Disadvantages

Handling of linear relationships, that's what trees suck at. Any linear relationship between an input feature and the outcome has to be approximated by hard splits, which produces 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 might not be desirable. Imagine a tree that predicts the value of a house and the tree splits in the square meters multiple times. One of the splits is at 100.5 square meters. Imagine a user of a house price estimator, that uses your decision tree model: She measures her house, concludes that the house has 99 square meters, types it into some nice web interface and get's a prediction of 200 000 Euro. The user notices that she forgot to measure a small storeroom with 2 square meters. The storeroom has a skewed wall, so she is not sure if she can count it fully towards the whole house area or only half of the space. So she decides to try both 100.0 and 101.0 square meters. The results: 200 000 Euro and 205 000 Euro, which is quite unintuitive, because there was no change from 99 square meters to 100, but from 100 to 101.

Trees are also quite **unstable**, so a few changes in the training dataset might create a completely different tree. That's because each split depends on the parent split. And if a different feature gets selected as the first split feature, the whole tree structure will change. It does not generate confidence in the model if the structure flips so easily.

Decision Rules (IF-THEN)

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's April (condition), THEN it will rain tomorrow (prediction). A single decision rule or a combination of several rules can be used to make predictions.

Keywords: decision rules, decision sets, decision lists, association rules, IF-THEN rules

Decision rules follow a general structure: IF the condition is true THEN make a particular 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 (number of $feature = value$ pairs combined with an AND) and there are not too many rules. In programming it's 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 $size > 100 \wedge garden = 1$ THEN $value = high$. Sometimes decision rules are also written like this: $(size > 100) \wedge (garden = 1) \Rightarrow (value = high)$.

Let's 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 that $value = high$.

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.

How do we assess whether a particular decision rule makes sense? 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 $(size = big) \wedge (location = good) \Rightarrow (value = high)$ 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's say of the 100 houses, where the rule $(size = big) \wedge (location = good) \Rightarrow (value = high)$

applies, 85 have $value = high$, 14 $value = medium$ and 1 $value = low$, then the accuracy of the rule is 85%.

Usually there is a trade-off between accuracy and support: By adding more features in 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:

- 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 dealing with 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 is true and so on. Decision lists are one-sided decision trees, where the first rule is the root node and with each rule, the tree grows in one direction. 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 is one of the simplest rule induction algorithm. 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's 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 x_j:
    - Create a cross table between the feature values and the (categorical) outcome.
    - For each feature values of x_j, 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 feature x_j.
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.

OneR can be considered as 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's 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:

	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 price for houses in good locations is $high$ and again we make two mistakes, because two houses have a $medium$ value. The error we make by of using the location feature is 4/10, for the size feature it's 3/10 and for the pet feature it's 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 issues, 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.

Let's try OneR with real data. We use the cervical cancer classification task to test the OneR algorithm. All continuous input features were divided into five intervals according to feature value frequency. 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 doesn't support regression tasks directly. 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 bike rentals with OneR by cutting the number of bike rentals 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. lower rental numbers 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.

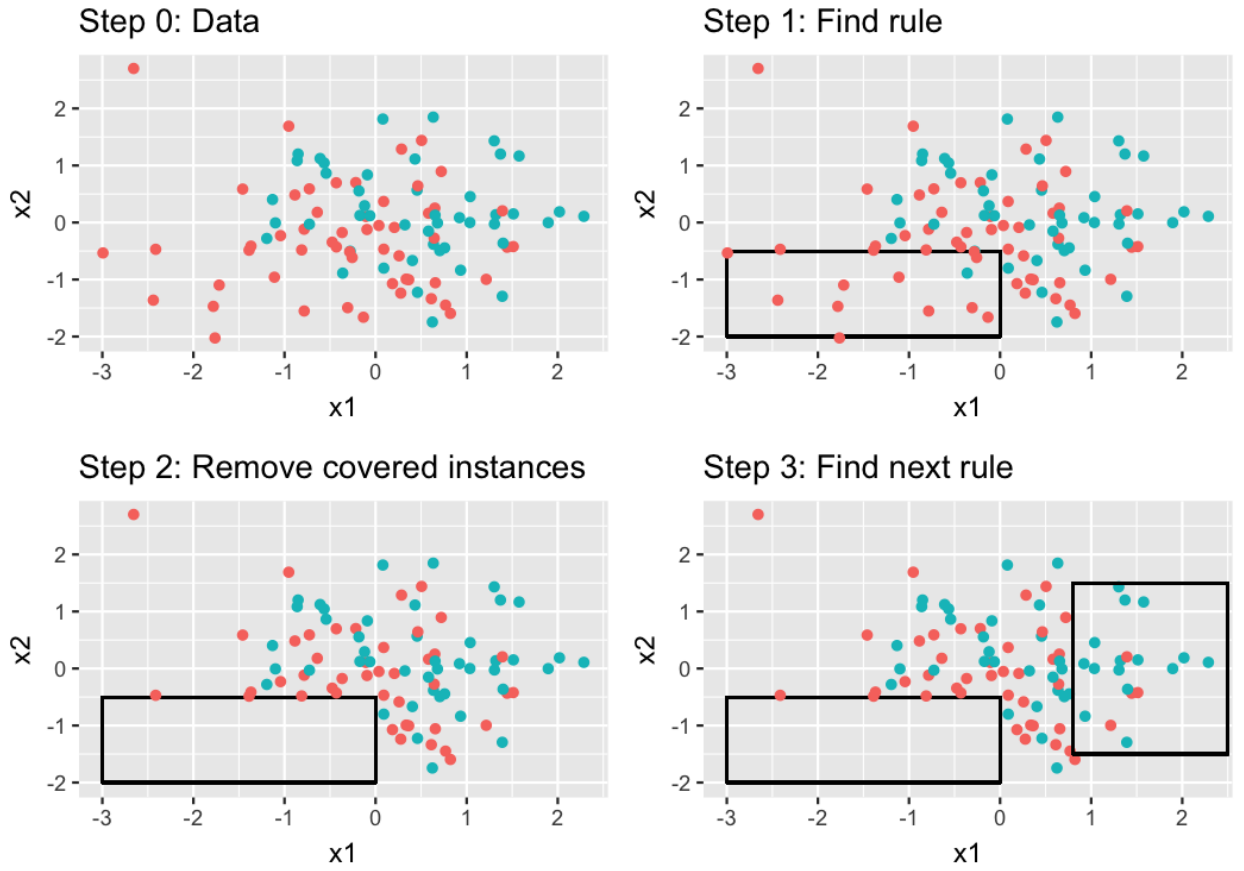
Sequential Covering

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, that is, each data point to which the condition applies, 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 quality threshold or some 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.



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).

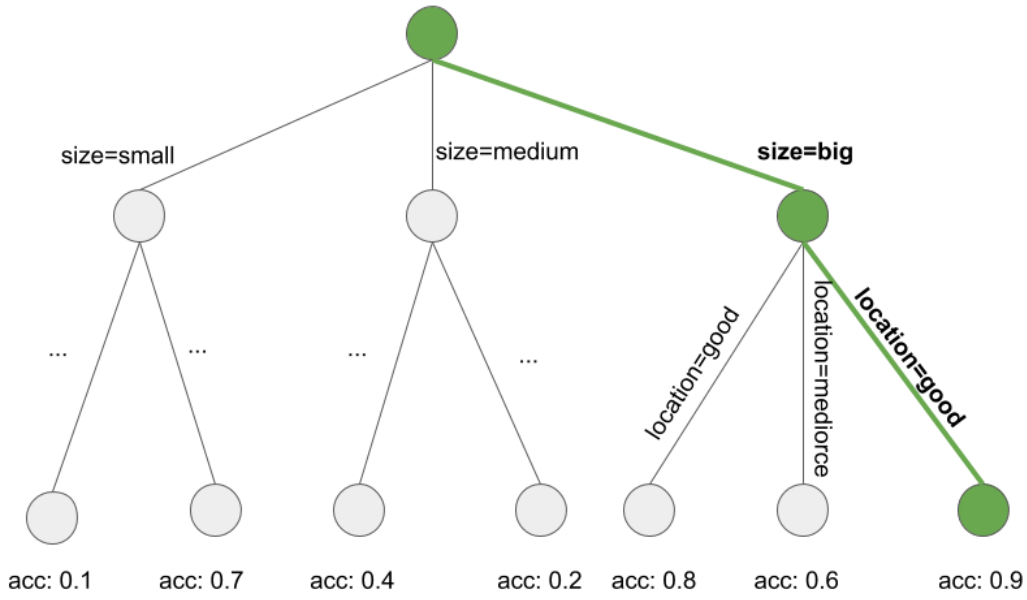
For multi-class settings, the approach must be modified. First, the classes are ordered by increasing prevalence. The sequential covering algorithm starts with the least common class, learns a rule for it, removes all covered instances, then moves on to the second least common class and so on. The current class is always treated as the positive class and all classes with a higher prevalence as 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 node in which we end 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. To extract a good rule, start at the root node, greedily and iteratively follow the path which locally produces the purest subset (e.g. with highest accuracy of target classes) and add all the split values to the rule condition. The prediction of the rule is the majority class of the node we land in. This example shows how we learn a rule for predicting the value of a house. We end up with: 'If $location = good$ and $size = big$, then $value = high$' (assuming high as the most common class in that terminal node).

Learning a single rule is equivalent to 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)[28] is a variant of the Sequential Covering algorithm. RIPPER is a bit more sophisticated and uses as 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.

We will use RIPPER to predict our data examples.

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

When we use RIPPER on the regression task to predict bike rentals some rules are found. Since RIPPER only works for classification, the bike counts must be turned into a categorical outcome. I

[28]Cohen, W. (1995). Fast effective rule induction. Icml. Retrieved from http://www.inf.ufrgs.br/~alvares/CMP259DCBD/Ripper.pdf

achieved this by cutting the bike counts into the quartiles. For example $(4548, 5956)$ is the interval covering predicted bike rentals between 4548 and 5956. The following table shows the 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 as the number of bike rentals . 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)[29] 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)[30]

But let's start slowly:

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 don't need the target outcome in this step) and extract frequently occurring patterns from them. A pattern can be a single feature value (e.g. size=medium) or a combination of feature values such as size=medium $\wedge$ location=bad.

[29]Letham, B., Rudin, C., McCormick, T. H., & Madigan, D. (2015). Interpretable classifiers using rules and bayesian analysis: Building a better stroke prediction model. Annals of Applied Statistics, 9(3), 1350–1371. https://doi.org/10.1214/15-AOAS848

[30]Borgelt, C. (2005). 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

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

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

where $x_j = A$ 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 $\wedge$ pets=allowed.

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

I'll 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 $size = big$ 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 $\wedge$ 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 (support), but not less. For example, if 20% of the houses are size=medium $\wedge$ 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 go into the details 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's 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's 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. (Well, that's Bayesian statistic.) If you are unfamiliar with Bayesian statistics, don't get too caught up in the following explanations. It's important to know that the Bayesian approach is a way to combine existing knowledge or requirements (so-called priori distributions) while also adapting 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 also 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 (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 can't directly calculate the correct answer, but we have to draw candidates, evaluate them and update our posteriori estimates (using the Markov chain Monte Carlo). 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 (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 length m for the list 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 $j = 1, \ldots, m$, do:
  - Sample the length of the rule l_j (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.

That's 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 (2016) [31]. 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[216]) 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),Smokes=0
Hormonal.Contraceptives=0,STDs=0
Num.of.pregnancies=[0,3.67),STDs..number.=[1.33,2.67)
Smokes=0,Hormonal.Contraceptives..years.=[0,10)
First.sexual.intercourse=[10,17.3)
Smokes..years.=[0,12.3),STDs..number.=[1.33,2.67)
Smokes=0,STDs..Number.of.diagnosis=[0,1)
Num.of.pregnancies=[3.67,7.33)
Num.of.pregnancies=[3.67,7.33),IUD..years.=[0,6.33)
Age=[13,36.7),STDs=0

Next, we apply the SBRL algorithm to the bike rental prediction task. This only works if the 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 bike rentals exceeds 4000 bikes on a day, else 0.

The following list was learned by SBRL:

[31] Yang, H., Rudin, C., & Seltzer, M. (2016). Scalable Bayesian Rule Lists, 31. Retrieved from http://arxiv.org/abs/1602.08610

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 {temp=[7.35,19.9)} (rule[489]) then positive probability = 0.44444444
else (default rule) then positive probability = 0.79746835

Let's predict the probability that the number of bike rentals will exceed 4000 for a day in 2012 with a temperature of 17 degrees Celsius. The first rule doesn't 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 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 tree often also suffer from replicated sub-trees, that is, when the splits following 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 binned, 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's the splitting criteria: Fixed interval lengths, quantiles or something else? Dealing with binning 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's 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 jumps 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[32], which was used for the examples in this book. OneR is also implemented in the Weka machine learning library[33] 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[34].
- SBRL is available as R package[35] (which I used for the examples), in Python[36] or as C implementation[37].

I won't 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' from Fuernkranz et. al (2012)[38]. It's 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[39], which implement RIPPER, M5Rules, OneR, PART and many more.

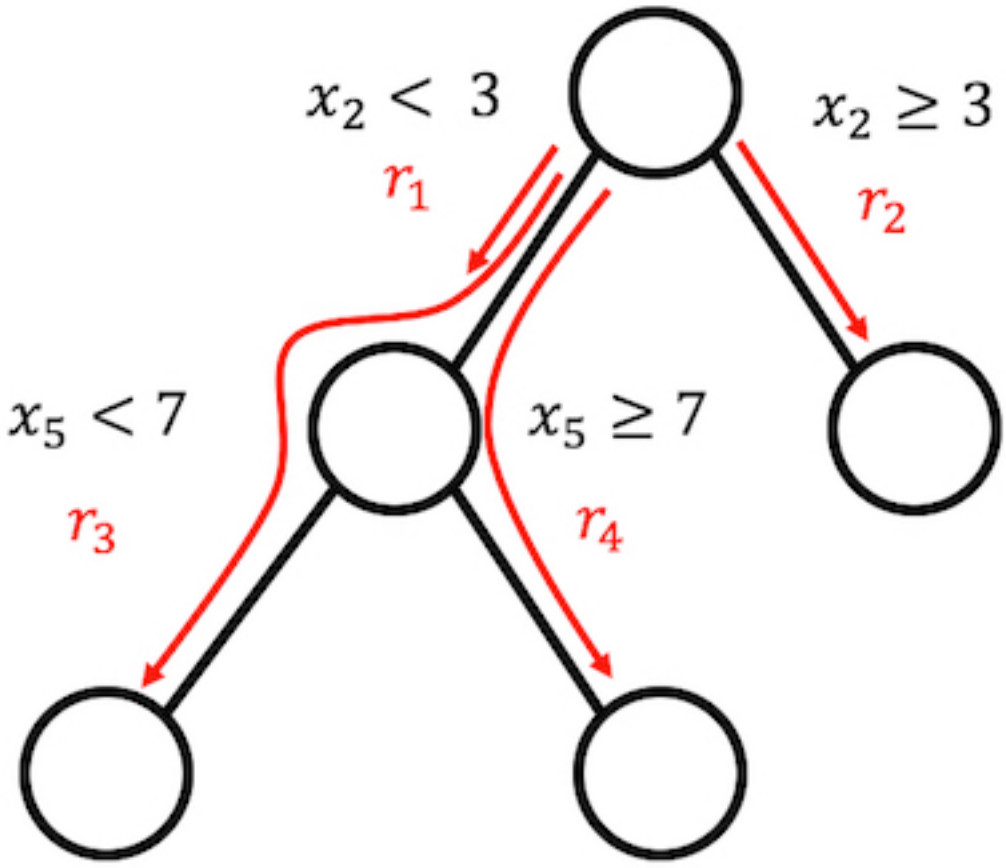
[32] https://cran.r-project.org/web/packages/OneR/
[33] (https://www.eecs.yorku.ca/tdb/_doc.php/userg/sw/weka/doc/weka/classifiers/rules/package-summary.html)
[34] https://cran.r-project.org/web/packages/RWeka/index.html
[35] https://cran.r-project.org/web/packages/sbrl/index.html
[36] https://github.com/datascienceinc/Skater
[37] https://github.com/Hongyuy/sbrlmod
[38] Fuernkranz, J., Gamberger, D., & Lavrač, N. (2012). Foundations of Rule Learning. Foundations of Rule Learning. https://doi.org/10.1007/978-3-540-75197-7_2
[39] https://www.eecs.yorku.ca/tdb/_doc.php/userg/sw/weka/doc/weka/classifiers/rules/package-summary.html

- IF-THEN rules can be used in linear models as described in this book in the chapter about the RuleFit algorithm.

RuleFit

The RuleFit algorithm (J. H. Friedman and Popescu 2008[40]) fits sparse linear models which include automatically detected interaction effects in the form of binary decision rules.

The standard linear model doesn't account for interactions between the features. Wouldn't it be convenient to have a model that is as simple and interpretable as linear models, but that also integrates feature interactions? RuleFit fills this gap. RuleFit fits a sparse linear model with the original features and also a set of new features which are decision rules. These new features capture interactions between the original features. RuleFit generates these features automatically from decision trees. Each path through a tree can be turned into a decision rule by combining the split decisions to a rule. The node predictions are thrown away and only the splits are used in the decision rules:



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

[40]Friedman, Jerome H, and Bogdan E Popescu. 2008. "Predictive Learning via Rule Ensembles." The Annals of Applied Statistics. JSTOR, 916–54.

Where do the decision trees come from? These are trees that are trained to predict the outcome of interest, so that the splits are meaningful for the task at hand and not arbitrary. Any algorithm that creates a lot of trees can be used for RuleFit, like a Random Forest for example. Each tree is disassembled into decision rules, which are used as additional features in a linear Lasso model.

The RuleFit paper uses the Boston housing data for illustration: The goal is to predict the median house value in the Boston neighbourhood. One of the rules (read: features) generated by RuleFit: "if (number of rooms > 6.64) and (concentration of nitric oxide < 0.67) then 1 else 0"

RuleFit also comes with a feature importance measurement, which helps to identify linear terms and rules that are important for the prediction. The feature importance is calculated from the weights of the regression model. The importance measure can be aggregated for the original features (which appear once untransformed and possibly in many decision rules).

RuleFit also introduces partial dependence plots to plot the average change of the prediction by changing one feature. The partial dependence plot is a model-agnostic method, which can be used with any model, and it has its own part in the book.

Interpretation and Example

Since RuleFit estimates a linear model in the end, the interpretation is equivalent to linear models. The only difference is that the model has new features that are coming 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 x_j increases by one unit, the predicted outcome changes by β_j.

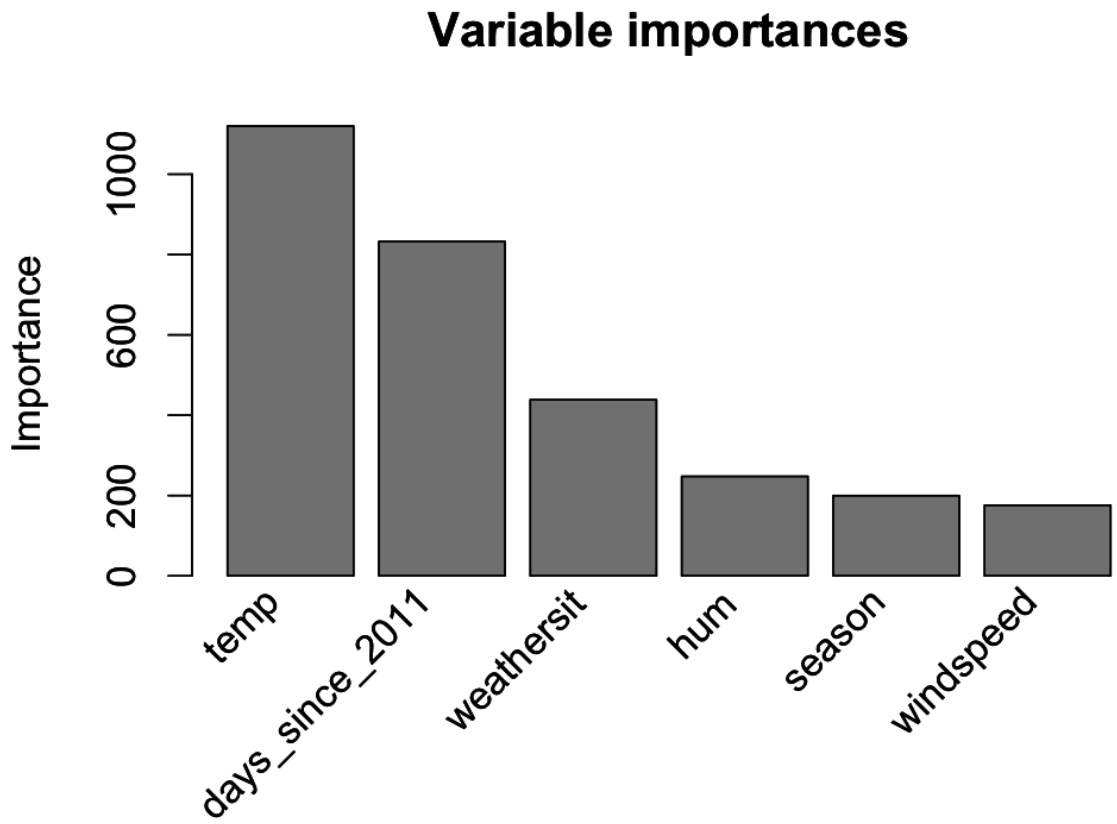
In this example, we use RuleFit to predict the number of bike rentals on a given day. Some of the genereated rules for the bike rental prediction task are: The table shows five of the generated rules with their Lasso weights and importances (measured as a function of the weight, see later in the chapter) after fitting 'RuleFit' on the bike dataset:

Description	Weight	Importance
days_since_2011 > 428 & temp > 5.199151	590	288
37.45835 <= hum <= 90.156225	-18	248
days_since_2011 > 111 & weathersit in ("GOOD", "MISTY")	598	228
temp > 8.058349 & weathersit in ("GOOD", "MISTY")	506	227
temp > 13.228349 & days_since_2011 > 554	522	186

The most important rule was: "days_since_2011 > 428 & temp > 5.199151" and the associated weight is 590.1. The interpretation is: If days_since_2011 > 428 & temp > 5.199151, then the predicted number of bike rentals increases by 590.1, given all other features values stay fixed. In total, 335 such rules were created from the original 8 features. Quite a lot! But thanks to Lasso, only 29 of the 335 got a weight different from zero.

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

important features:



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

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

Guidelines

In this section we will talk about the advantages and disadvantages of RuleFit and how to interpret it.

Interpretation template

The interpretation is analogue to linear models: The predicted outcome changes by β_j if feature x_j changes by one unit, given all other features stay the same. 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 for rule r_k in the linear model). And, respectively, for classification: If all conditions of decision rule r_k apply, the odds for event vs. no-event changes by a factor of α_k.

Advantages:

- RuleFit adds **feature interactions** automatically 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 non-linear relationships.
- RuleFit can handle both classification and regression tasks.
- The created rules are easy to interpret, because they are binary decision rules. Either the rule applies to an instance or not. Good interpretability is only guaranteed as long as the number of conditions within a rule is not to big. A rule with 1 to 3 conditions seems reasonable to me. This translates into 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 each instance, so for one instance only a handful of rules are important (non-zero weights). This improves local interpretability.
- The RuleFit proposes a bunch of useful diagnostic tools. These tools are model-agnostic, that's why you will find them in the model-agnostic section: feature importance, partial dependence plots and feature interactions.

Disadvantages:

- Sometimes RuleFit creates many rules which get a non-zero weight in the Lasso model. The interpretability degrades with higher number of features in the model. A promising solution is to force feature effects to be monotonic, meaning that an increase in a feature has to result in an increase of the predicted outcome.
- An anecdotal drawback: The papers claim good performance of RuleFit - often close to the predictive performance of Random Forests! - yet in the few cases where I personally tried it, the performance was disappointing.
- 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 (given all features are fixed, increasing feature x_j by one unit, increases the predicted outcome by β_j). It gets a bit more tricky if you have overlapping rules: For example one decision rule (feature) for the bike prediction could be: "temp > 10" and another rule could be "temp > 15 & weather='GOOD'". When the weather is good and the temperature is above 15 degrees, the temperature is automatically also always bigger then 10, which means in the cases where the second rule applies, the first one also always applies. The interpretation of the estimated weight for the second rule is: 'Given all other features are fixed, the predicted number of bikes increases by β_2'. 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.

The RuleFit algorithm is implemented in R by Fokkema and Christoffersen (2017)[41] and you can find a Python version on Github[42].

[41]Fokkema, Marjolein, and Benjamin Christoffersen. 2017. Pre: Prediction Rule Ensembles. https://CRAN.R-project.org/package=pre.
[42]https://github.com/christophM/rulefit

Theory

Let's dive deeper into the technicalities of the RuleFit algorithm. RuleFit consists of two components: The first component produces "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"). It enables automatic integration of interactions between features into a linear model, while having the interpretability of a sparse linear model.

Step 1: Rule generation

How does a rule look like? The rules that the algorithm generates have a simple form: For example: "if $x2 < 3$ and $x5 < 7$ then 1 else 0". The rules are constructed by disassembling decision trees: Each path to a node in a tree can be turned into a decision rule. The trees used for the rules are fitted to predict the target outcome. The splits and resulting rules are optimised to predict the outcome you are interested in. You simply chain the binary decisions that lead to a certain node with a logical "AND", and voilà, 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 turned into multiple rules. Not only boosted trees, but any type of ensemble of trees can be used to generate the trees for RuleFit:

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

where M is the number of trees and $f_m(x)$ represents the prediction function of the m-th tree. Bagged ensembles, Random forests, AdaBoost and MART yield ensemble of trees and can be used for RuleFit.

From all of the trees of the ensemble, we produce the rules. Each rule r_m takes on the form:

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

where T_m is the set of features used in m-th tree, $I(\cdot)$ is the indicator function, which is 1 if the feature x_j is in the specified subset of values s_{jm} for x_j (as specified by the tree splits) and otherwise 0. For numerical features, s_{jm} is one to multiple intervals in the value range of the feature x_j, depending on the number of splits in that feature. In case of a single split, the s_{jm} looks like one of the two cases: $x_{s_{jm},\text{lower}} < x_j$ or $x_j < x_{s_{jm},upper}$. Further splits in that feature create more complicated intervals. For categorical features the subset s_{jm} contains some specific categories of x_j.

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 will only be equal to 1 if all of the 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:

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

In total,

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

rules are created from the ensemble of M trees, with t_m terminal nodes each. A trick that is introduced by the RuleFit authors is to fit trees with random depth, so that a lot of diverse rules are generated with different lengths. Note that we throw away the predicted value in each node and only keep the conditions that lead us to the node and create a rule from it. The weighting of the decision rules will happen in step 2 of fitting RuleFit.

Another way to see the first step is, that it generates a new set of features out of your original features X. Those features are binary and can represent quite complex interactions of your original X. The rules are chosen to maximise the prediction task at hand. The rules are automatically generated from the covariates matrix X. You can see the rules simply as new features based on your original features.

Step 2: Sparse linear model

You will get A LOT of rules from the first step. Since the first step is only a feature transformation function on your original dataset you are still not done with fitting a model and also you want to reduce the number of rules. Next to the rules, also all your 'raw' features from your original dataset will be used in the Lasso linear model. Every rule and original feature becomes a feature in Lasso and gets a weight estimate. The original, raw features are added because trees suck at representing simple linear relationships between y and x. Before we put everything into Lasso to get a sparse linear model, we winsorise 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 x_j. A choice of 0.05 for δ means that every value of x_j that is in the 5% lowest or 5% highest values will be set to the values at 5% or 95% respectively. As a rule of thumb, you can choose $\delta = 0.025$. In addition, the linear terms have to be normalised so that they have the same prior influence 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 $s_k \sim U(0, 1)$.

We combine both types of features to generate a new feature matrix and estimate 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 $\hat{\alpha}$ are the estimated weights for the rule features and $\hat{\beta}$ 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:

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

The outcome is a linear model that has linear effects for all of the original features and for the rules. The interpretation is the same as with 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 standardised predictor:

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

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

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

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

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 for 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 x_j occurs as a linear term and possibly also within many decision rules. How do we measure the total importance of the feature x_j? The importance $J_j(x)$ of feature x_j can be measured at 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 that constitute rule r_k. Summing the feature importance over all instances gives us the global feature importance:

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

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

Other Interpretable Models

The list of interpretable models is ever-growing and of unknown size. It contains simple models like linear models, decision trees and naive Bayes, but also more complex ones that combine or modify non-interpretable machine learning models to make them interpretable. Especially publications about the latter type of models are currently created with a high frequency and it is hard to keep up with the developments. We only tease a few additional ones in this chapter, especially the simpler and more established candidates.

Naive Bayes classifier

The naive Bayes classifier makes use of the Bayes' theorem of conditional probabilities. For each feature it computes the probability for a class given the value of the feature. The naive Bayes classifier does so for each feature independently, which is the same as having 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 in the following way:

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

The term Z is a scaling parameter that ensures that the probabilities for all classes sum up to 1. The probability of a class, given the features 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 is interpretable on the modular level. For each classification it is very clear for each feature how much it contributes towards a certain class prediction.

K-Nearest Neighbours

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

This algorithm is different from the other interpretable models presented in this book, since it is an instance-based learning algorithm. How is k-nearest neighbours interpretable? For starters, it has no parameters to learn, so there is no interpretability on a modular level, like in linear models. Also, it lacks global model interpretability, since the model is inherently local and there are no global weights or structures that are learned explicitly by the k-nearest neighbour method. Maybe it is interpretable on a local level? To explain a prediction, you can always retrieve the k-neighbours

that were used for the prediction. If the method is interpretable solely depends on the question if you can ‘interpret’ a single instances in the dataset. If the dataset consists of hundreds or thousands of features, then it is not interpretable, I’d argue. But if you have few features or a way to reduce your instance to the most important features, presenting the k-nearest neighbours can give you good explanations.

And so many more …

Many algorithms can produce interpretable models and not all can be listed here. If you are a researcher or just a big fan and user of a certain interpretable method, that is not listed here, get in touch with me and add the method to this book!

Model-Agnostic Methods

Separating the explanations from the machine learning model (= model-agnostic interpretability methods) has some benefits (Ribeiro, Singh, and Guestrin 2016[43]). The big advantage of model-agnostic interpretability methods over model-specific ones is their flexibility. The machine learning developer is free to use any machine learning model she likes, when the interpretability methods can be applied to any model. Anything that is build on top of an interpretation of a machine learning model, like a graphic or some user interface, also becomes independent of the underlying machine learning model. Usually not one but many types of machine learning models are evaluated for solving a task and if you compare the models in terms of interpretability, it is easier to do with model-agnostic explanations, because the same method can be used for any type of model.

An alternative to model-agnostic interpretability methods is using only interpretable models, which often has the big disadvantage to usually loose accuracy compared to other machine learning models and locking you into one type of model and interpretability method. The other alternative is to use model-specific interpretability methods. The drawback here is also that it ties you to this one algorithm and it will be hard to switch to something else.

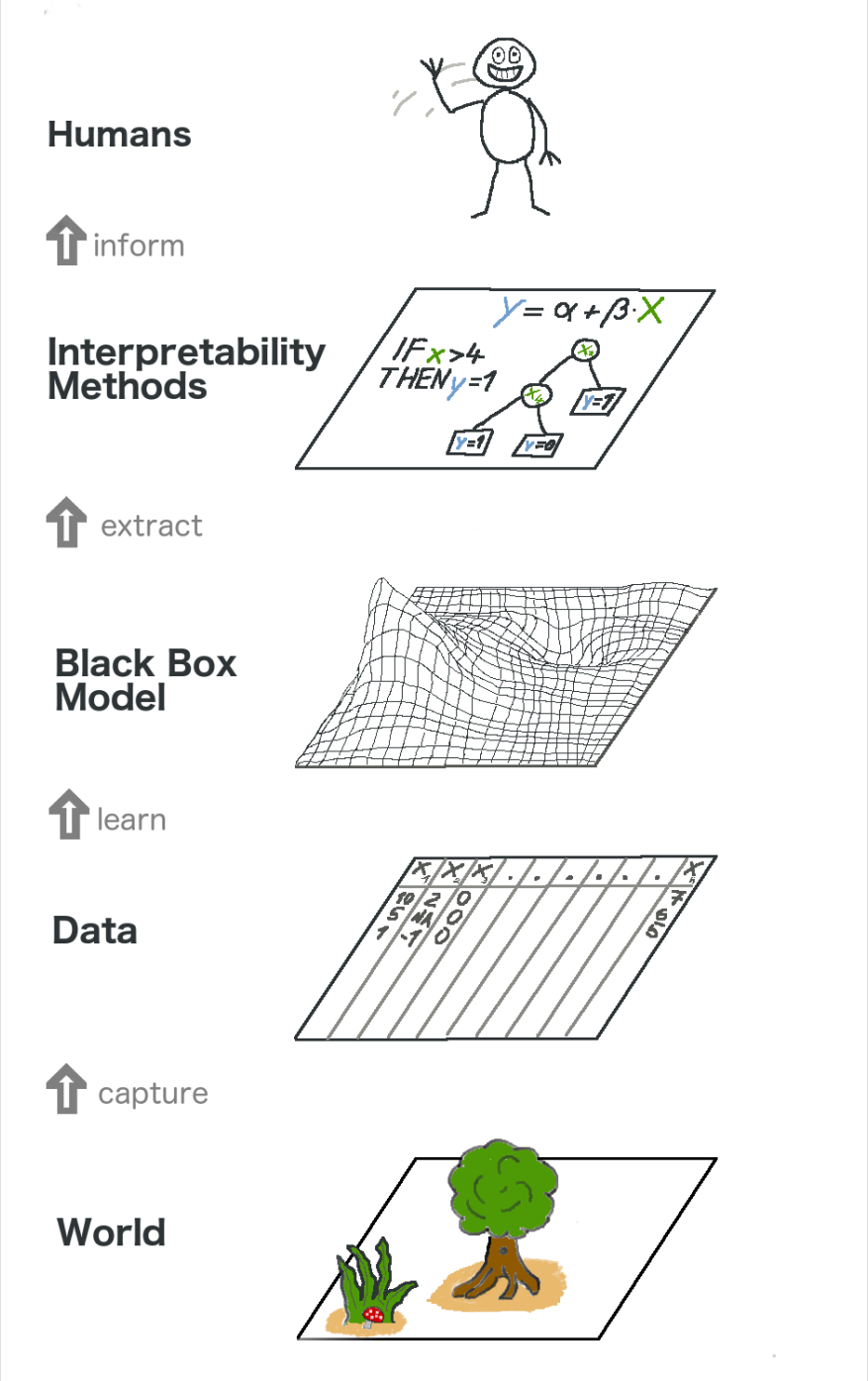
Desirable aspects of a model-agnostic explanation system are (Ribeiro, Singh, and Guestrin 2016):

- **Model flexibility**: Not being tied to an underlying particular machine learning model. The method should work for random forests as well as deep neural networks.
- **Explanation flexibility**: Not being tied to a certain form of explanation. In some cases it might be useful to have a linear formula in other cases a decision tree or a graphic with feature importances.
- **Representation flexibility**: The explanation system should not have to use the same feature representation as the model that is being explained. For a text classifier that uses abstract word embedding vectors it might be preferable to use the presence of single words for the explanation.

The bigger picture

Let's take a high level view on model-agnostic interpretability. We first abstract the world by capturing it by collecting data and abstract it further by learning the essence of the data (for the task) with a machine learning model. Interpretability is just another layer on top, that helps humans understand.

[43]Ribeiro, Marco Tulio, Sameer Singh, and Carlos Guestrin. 2016. "Model-Agnostic Interpretability of Machine Learning." ICML Workshop on Human Interpretability in Machine Learning, no. Whi.



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 bottom 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 digitalise 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 on top of 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.
- On top of the 'Black Box Model'-layer is the 'Interpretability Methods'-layer that helps us deal with the opaqueness of machine learning models. What were the 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 is waving at you because you are reading this book and you are helping to provide better explanations for black box models! Humans are the consumers of the explanations, ultimately.

This layered abstraction also helps in understanding what the differences in approaches between statisticians and machine learning practitioners are. Statistician are concerned with the 'Data' layer, like planning clinical trials or designing surveys. They skip the 'Black Box Model'-layer and go right to the 'Interpretability Methods'-layer. Machine learning specialists are also concerned with the 'Data'-layer, like collecting labeled samples of skin cancer images or crawling Wikipedia. Then comes the machine learning model. 'Interpretability Methods' is skipped and the human deals directly with the black box models prediction. It's a nice thing, that in interpretable machine learning, the work of a statistician and a machine learner fuses and becomes something better.

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 feed other machines and so on. But overall it is a useful abstraction for understanding how (model-agnostic) interpretability becomes this new layer on top of machine learning models.

Partial Dependence Plot (PDP)

The partial dependence plot shows the marginal effect of a feature on the predicted outcome of a previously fit model (J. H. Friedman 2001[44]). The prediction function is fixed at a few values of the chosen features and averaged over the other features.

Other names: marginal means, predictive margins, marginal effects.

A partial dependence plot can show if the relationship between the target and a feature is linear, monotonic or more complex. Applied to a linear regression model, partial dependence plots will always show a linear relationship, for example.

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 term x_S is the set of features for which the partial dependence function should be plotted and x_C are the other features that were used in the machine learning model $\hat{f}$. Usually, there are only one or two features in x_S. Concatenated, x_S and x_C make up x. Partial dependence works by marginalizing the machine learning model output $\hat{f}$ over the distribution of the features x_C, so that the remaining function shows the relationship between the x_S, in which we are interested, and the predicted outcome. By marginalizing over the other features, we get a function that only depends on features x_S, interactions between x_S and other features included.

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

$$\hat{f}_{x_S}(x_S) = \frac{1}{n}\sum_{i=1}^{n} \hat{f}(x_S, x_{Ci})$$

In this formula, x_{iC} 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. One assumption made for the PDP is that the features in x_C are uncorrelated with the features in x_S. If this assumption is violated, the averages, which are computed for the partial dependence plot, incorporate data points that are very unlikely or even impossible (see disadvantages).

For classification, where the machine model outputs probabilities, the partial dependence function displays the probability for a certain class given different values for features x_S. A straightforward way to handle multi-class problems is to plot one line or one plot per class.

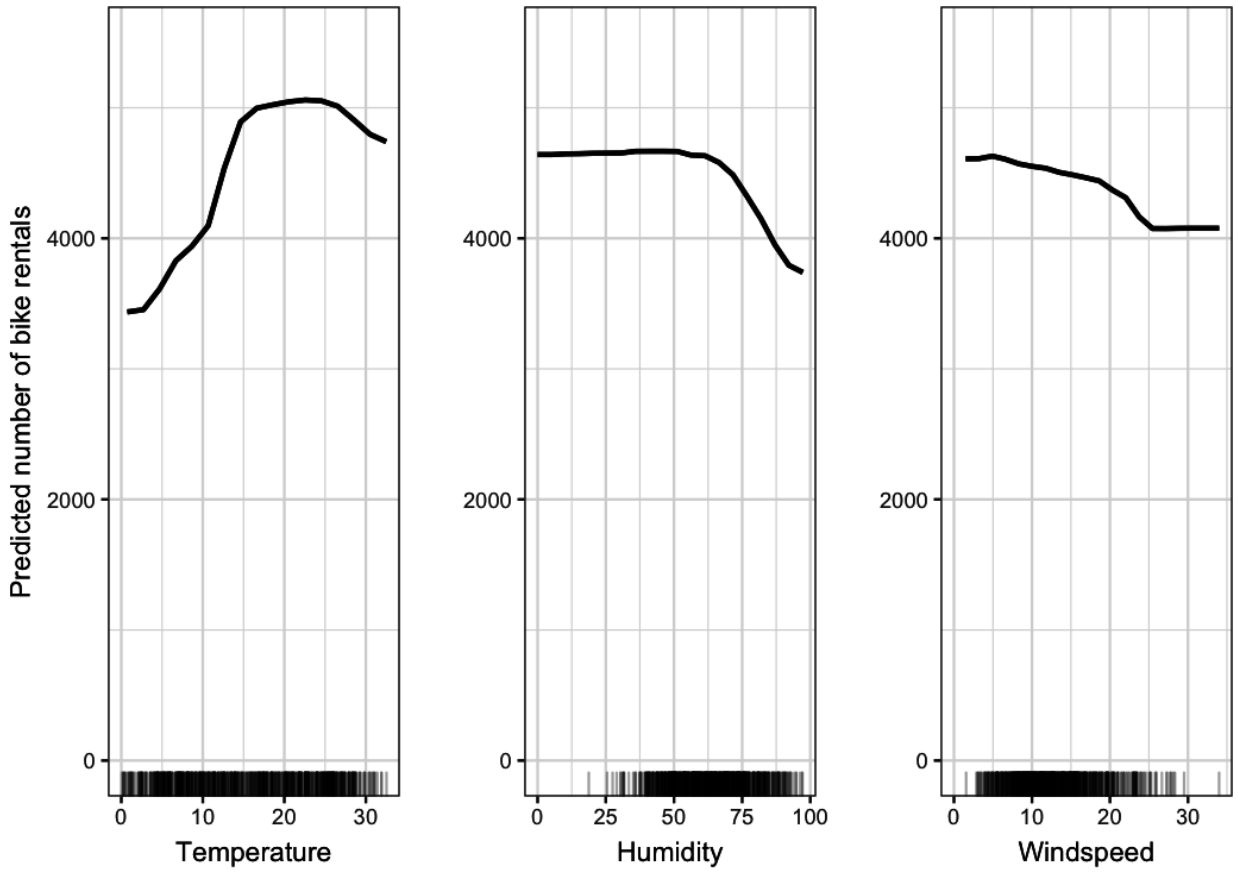
The partial dependence plot is a global method: The method takes into account all instances and makes a statement about the global relationship of a feature with the predicted outcome.

[44]Friedman, Jerome H. 2001. "Greedy Function Approximation: A Gradient Boosting Machine." Annals of Statistics. JSTOR, 1189â€“1232.

Examples

In practice, the set of features x_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's return to the regression example, in which we predict bike rentals. We first fit a machine learning model on the dataset, for which we want to analyse the partial dependencies. In this case, we fitted a RandomForest to predict the bike rentals and use the partial dependence plot to visualize the relationships the model learned. The influence of the weather features on the predicted bike counts:

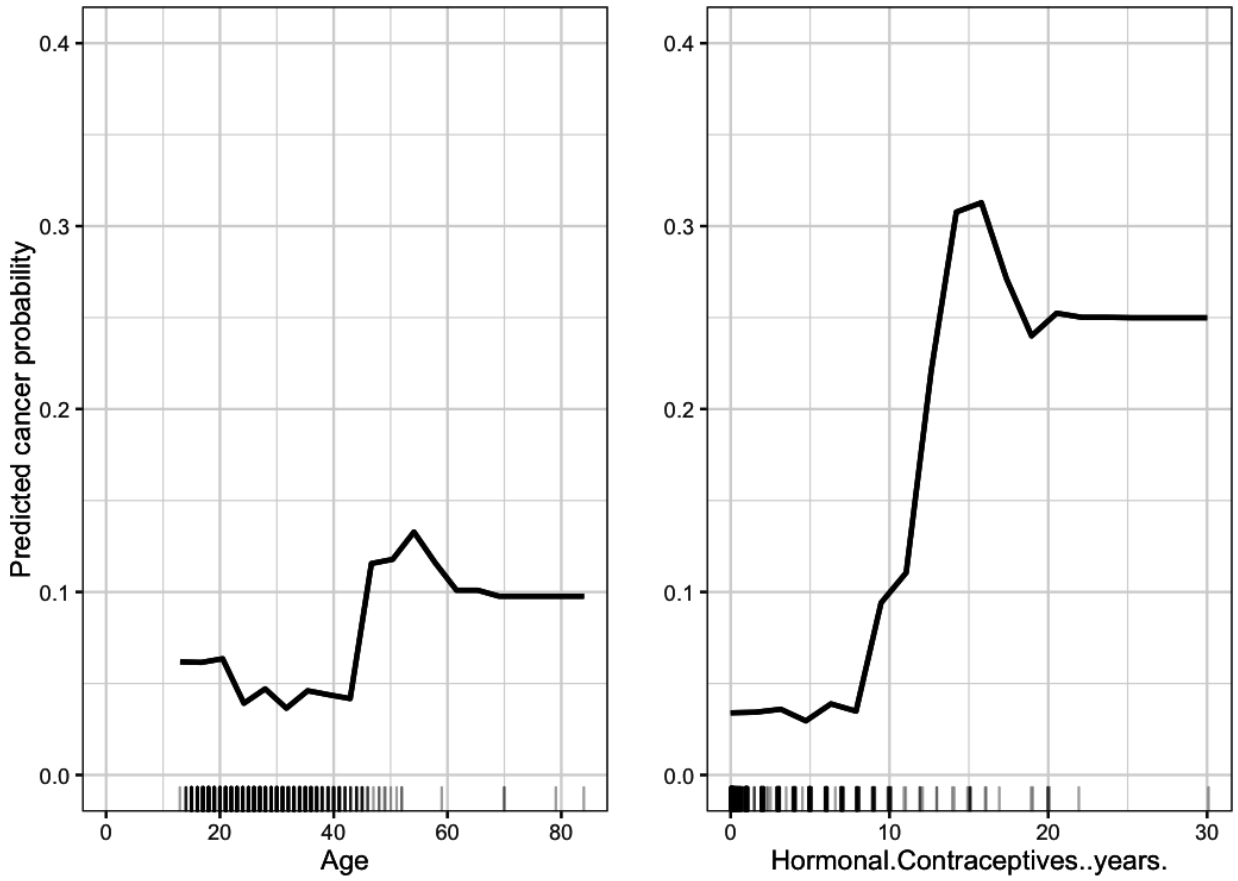


Partial dependence plots for the rental bike prediction model and different weather measurements (Temperature, Humidity, Windspeed). The biggest differences can be seen in the temperature: On average, the hotter the more bikes are rented, until 20C degrees, where it stays the same also for hotter temperatures and drops a bit again towards 30C degrees. The marks on the x-axis indicate the distribution of the feature in the data.

For warm (but not too hot) weather, the model predicts a high number of bike rentals on average. The potential bikers are increasingly inhibited in engaging in cycling when humidity reaches above 60%. Also, the more wind the less people like to bike, which makes sense. Interestingly, the predicted bike counts don't drop between 25 and 35 km/h windspeed, but there is just not so much training data, so we can't be confident about the effect. At least intuitively I would expect the bike rentals

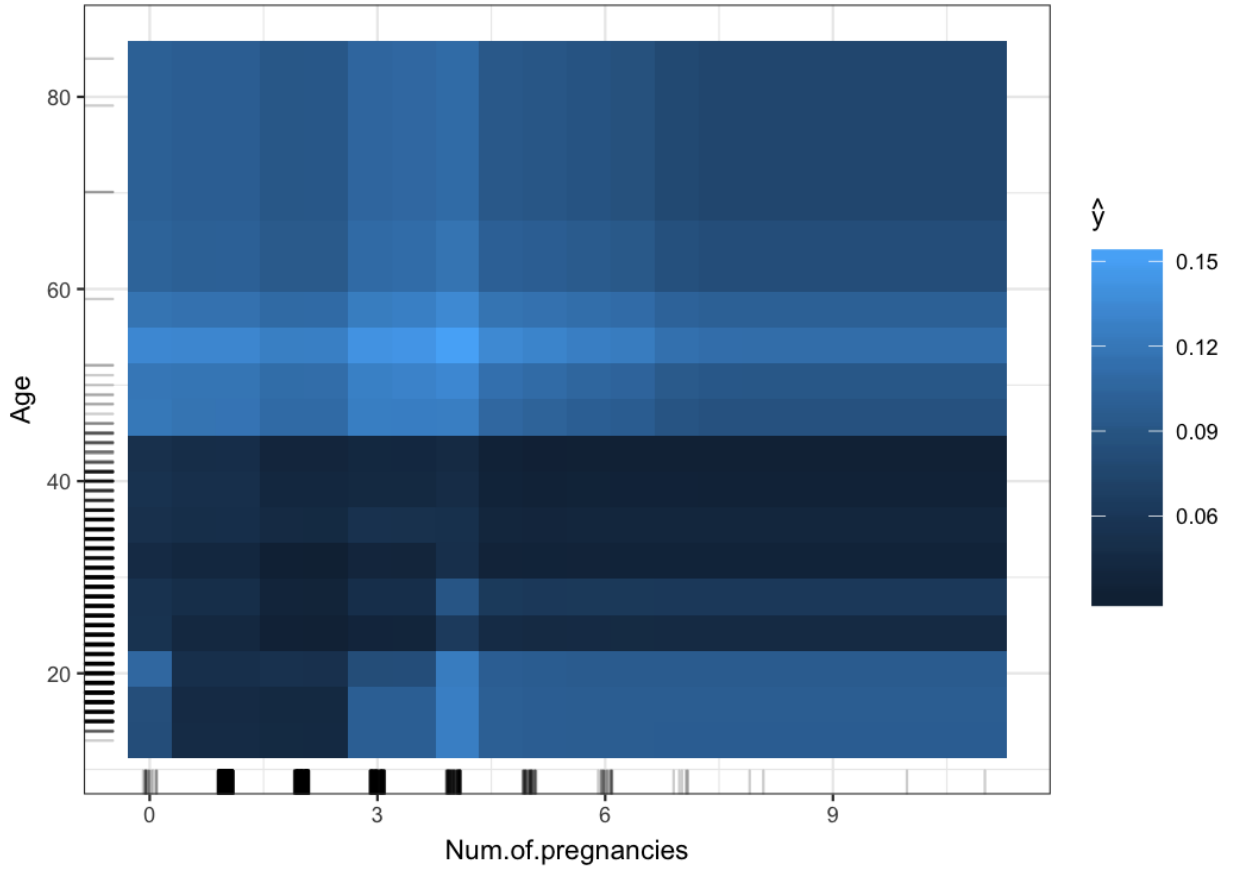
to drop with any increase in windspeed, especially when the windspeed is very high.

We also compute the partial dependence for cervical cancer classification. Again, we fit a RandomForest to predict whether a woman has cervical cancer given some risk factors. Given the model, we compute and visualize the partial dependence of the cancer probability on different features:



Partial dependence plot of cancer probability and the risk factors age and number of years with hormonal contraceptives. For the age feature, the partial dependence plot shows that on average the cancer probability is until 40 and increases after that. The sparseness of data points after age of 50 indicates that the model did not have many data points to learn from above that age. The number of years on hormonal contraceptives is associated with a higher cancer risk after 10 years. But again, there are not many data points in that region, which implies that we might not be able to rely on the machine learning model predictions for >10 years on contraceptives.

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



Partial dependence plot of cancer probability and the interaction of age and number of pregnancies. The plot shows the increase in cancer probability at 45, regardless of number of pregnancies. An interesting interaction happens at ages below 25: Young women who had 1 or 2 pregnancies have a lower predicted cancer risk, compared with women who had zero or above two pregnancies. The model predicts a - kind of - protective effect against cancer for 1 or 2 pregnancies. But be careful with drawing conclusions: This might just be a correlation and not causal! The cancer risk and number of pregnancies could be caused by another, unmeasured factor in which the young women differ.

Advantages

- The computation of partial dependence plots is **intuitive**: The partial dependence curve at a certain feature value represents the average prediction when we force all data points to take on that feature value. In my experience, laypersons usually grasp the idea of PDPs quickly.
- If the feature for which you computed the PDP is uncorrelated with the other model features, then the PDPs are perfectly representing how the feature influences the target on average. In this uncorrelated case the **interpretation is clear**: The partial dependence plots shows how on average the prediction changes in your dataset, when the j-th feature is changed. It's complicated when features are correlated, see also disadvantages.
- Partial dependence plots are **simple to implement**.
- **Causal interpretation** : The calculation for the partial dependence plots has a causal interpretation: We intervene on x_j and measure the changes in the predictions. By doing this, we analyse

the causal relationship between the feature and the outcome.[45] The relationship is causal for the model - because we explicitly model the outcome on the feature - but not necessarily for the real world!

Disadvantages

- The **maximum number of features** you can look at jointly is - realistically - two and - if you are stubborn and pretend that 3D plots on a 2D medium are useful - three. That's not the fault of PDPs, but of the 2-dimensional representation (paper or screen) and also our inability to imagine more than 3 dimensions.
- Some PD visualisations don't include the **feature distribution**. Omitting the distribution can be misleading, because you might over-interpret the line in regions, with almost no feature values. This problem is easy to fix by showing a rug (indicators for data points on the x-axis) or a histogram.
- The **assumption of independence** poses the biggest issue. The feature(s), for which the partial dependence is computed, is/are assumed to be independently distributed from the other model features we average over. For example: Assume 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, let's say height, we assume that the other features (weight) are not correlated with height, which is obviously a wrong assumption. For the computation of the PDP at some height (for example at height = 200cm) we average over the marginal distribution of weight, which might include a weight below 50kg, which is unrealistic for a 2 meter person. In other words: When the features are correlated, we put weight on areas of the feature distribution where the actual probability mass is very low (for example it is unlikely that someone is 2 meters tall but weighting below 50kg). A solution to this problem are ALEPlots[46], that only average over close data points.
- **Heterogenous effects might be hidden** because the PDP only shows the average over the observations. Assume that for feature x_j half your data points have a positive assocation with the outcome - the greater x_j the greater $\hat{y}$ - and the other half has negative assocation - the smaller x_j the greater $\hat{y}$. The PDP curve might be a straight, horizontal line, because the effects of both dataset halves cancel each other out. You then conclude that the feature has no effect on the outcome. By plotting the individiual conditional expectation curves instead of the aggregated line, we can uncover heterogeneous effects.

[45] Zhao, Q., & Hastie, T. (2016). Causal interpretations of black-box models. Technical Report.

[46] Apley, D. W. (n.d.). Visualizing the Effects of Predictor Variables in Black Box Supervised Learning Models, 1â€“36. Retrieved from https://arxiv.org/ftp/arxiv/papers/1612/1612.08468.pdf

Individual Conditional Expectation (ICE)

For a chosen feature, Individual Conditional Expectation (ICE) plots draw one line per instance, representing how the instance's prediction changes when the feature changes.

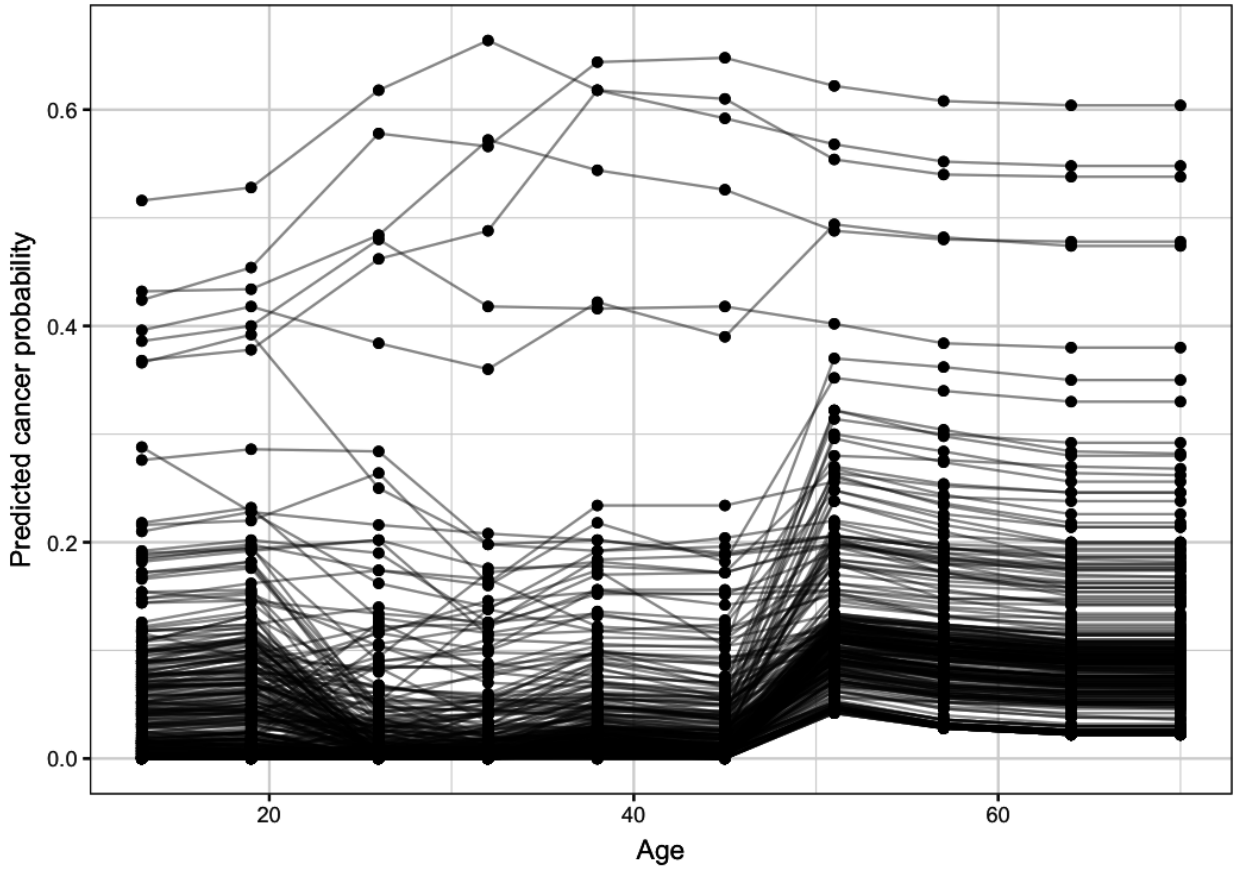
The partial dependence plot for visualizing the average effect of a feature is a global method, because it does not focus on specific instances, but on an overall average. The equivalent to a PDP for local expectations is called individual conditional expectation (ICE) plot (Goldstein et al. 2015[^Goldstein2015]). An ICE plot visualizes the dependence of the predicted response on a feature for EACH instance separately, resulting in multiple lines, one for each instance, compared to one line 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 leaving all other features the same, creating variants of this instance by replacing the feature's value with values from a grid and letting the black box make the predictions with 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.

So, what do you gain by looking at individual expectations, instead of partial dependencies? Partial dependence plots can obfuscate a heterogeneous relationship that comes from interactions. PDPs can show you how the average relationship between feature x_S and $\hat{y}$ looks like. This works only well in cases where the interactions between x_S and the remaining x_C are weak. In case of interactions, the ICE plot will give a lot 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} is kept fixed.

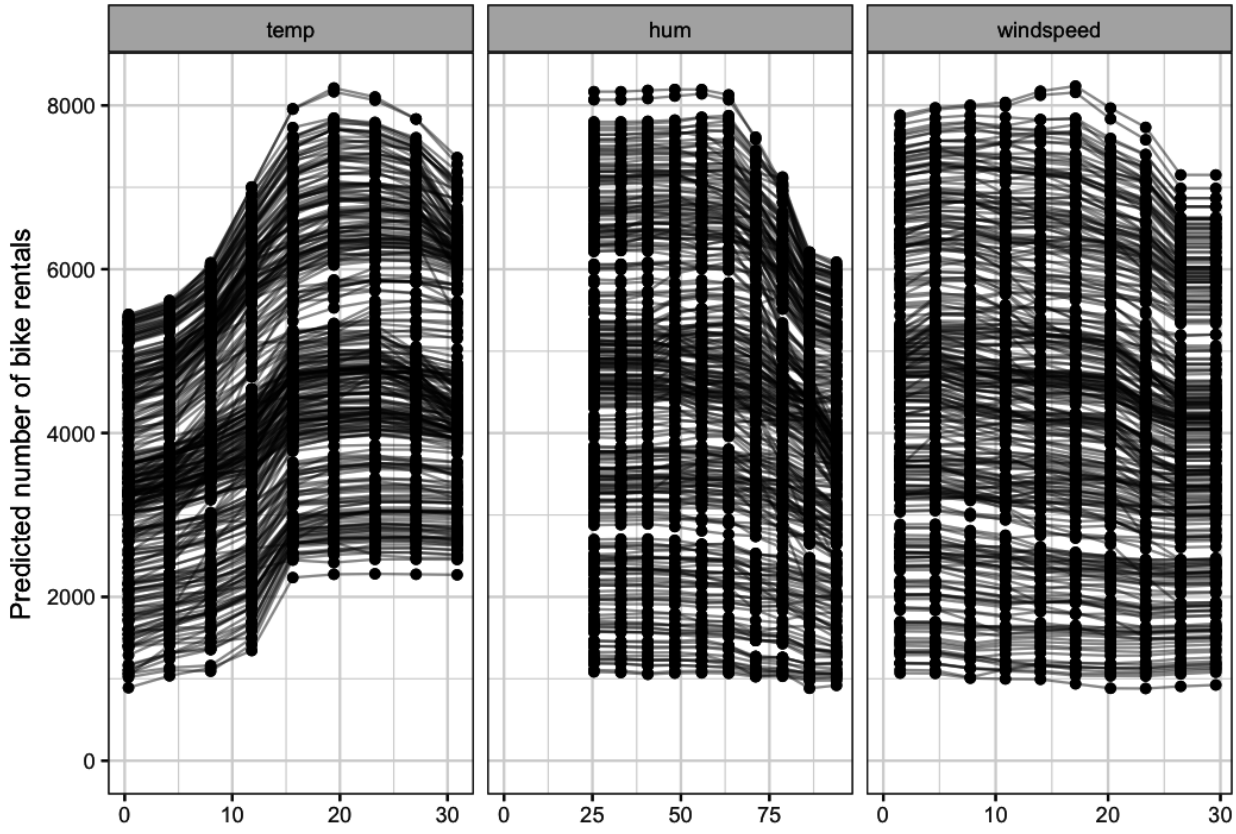
Example

Let's go back to the dataset about risk factors for cervical cancer and see how each instance's prediction is associated with the feature 'Age'. The model we will analyze is a RandomForest 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 does it hold true for each woman in the dataset? The ICE plot reveals that the most women's predicted probability follows the average pattern of increase at 50, but there are a few exceptions: For the few women that have a high predicted probability at a young age, the predicted cancer probability does not change much with increasing age.



Individual conditional expectation plot of cervical cancer probability by age. Each line represents the conditional expectation for one woman. Most women with a low cancer probability in younger years see an increase in predicted cancer probability, given all other feature value stay the same. Interestingly for a few women that have a high estimated cancer probability bigger than 0.4, the estimated probability does not change much with higher age.

The next figures shows an ICE plot for the bike rental prediction (the underlying prediction model is a RandomForest).



Individual conditional expectation plot of expected bike rentals and weather conditions. The same effects as in the partial dependence plots can be observed.

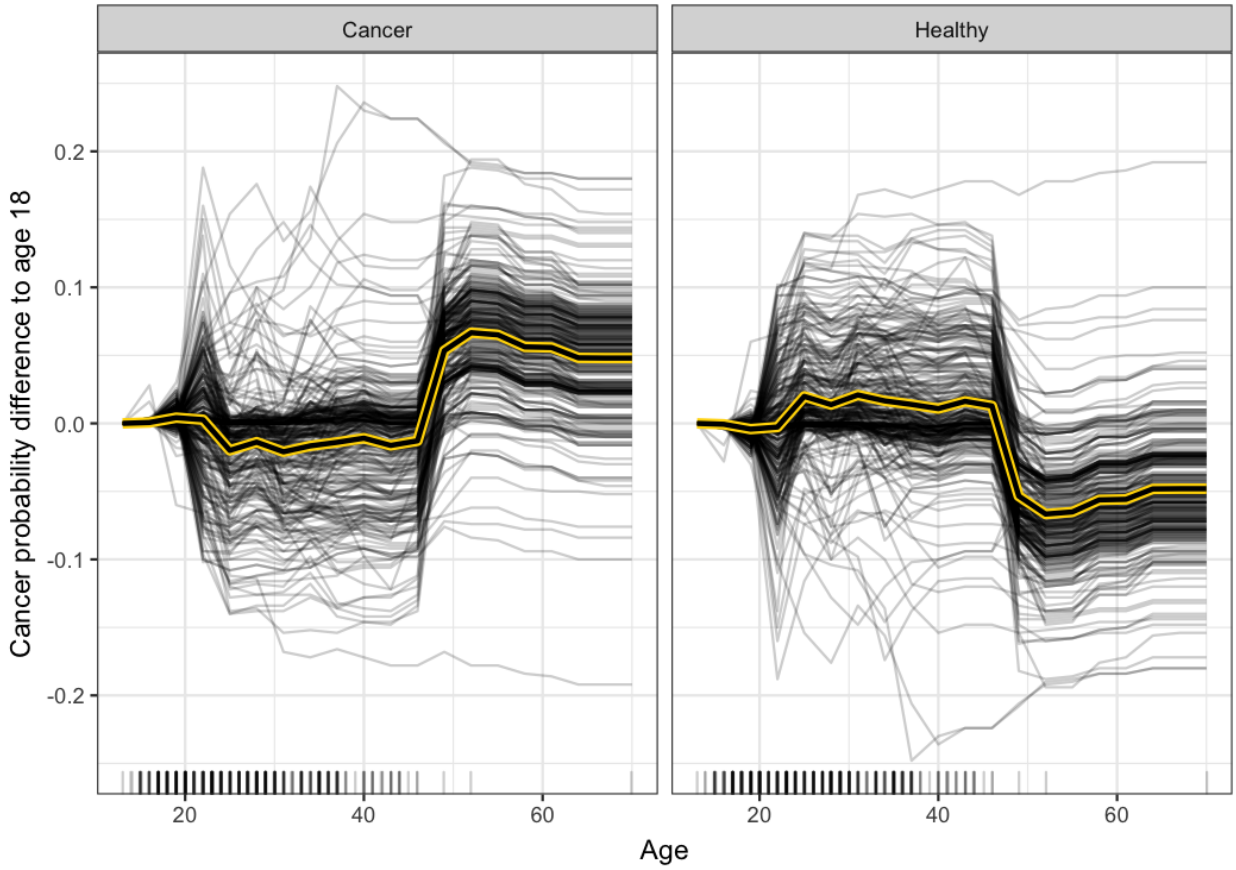
All curves seem to follow the same course, so there seem to be no obvious interactions. That means that the PDP is already a good summary of the relationships of the displayed features and the predicted bike rentals.

Centered ICE Plot

There is one issue with ICE plots: It can be hard to see if the individual conditional expectation curves differ between individuals, because they start at different $\hat{f}(x)$. An easy fix is to center the curves at a certain point in x_S and only display the difference in the predicted response. The resulting plot is called centered ICE plot (c-ICE). Anchoring the curves at the lower end of x_S is a good choice. The new curves are defined as: $\hat{f}_{cent}^{(i)} = \hat{f}_i - \mathbf{1}\hat{f}(x^*, x_{C_i})$ where $\mathbf{1}$ is a vector of 1's with the appropriate number of dimensions (usually one- or two-dimensional), $\hat{f}$ the fitted model and x^* the anchor point.

Example

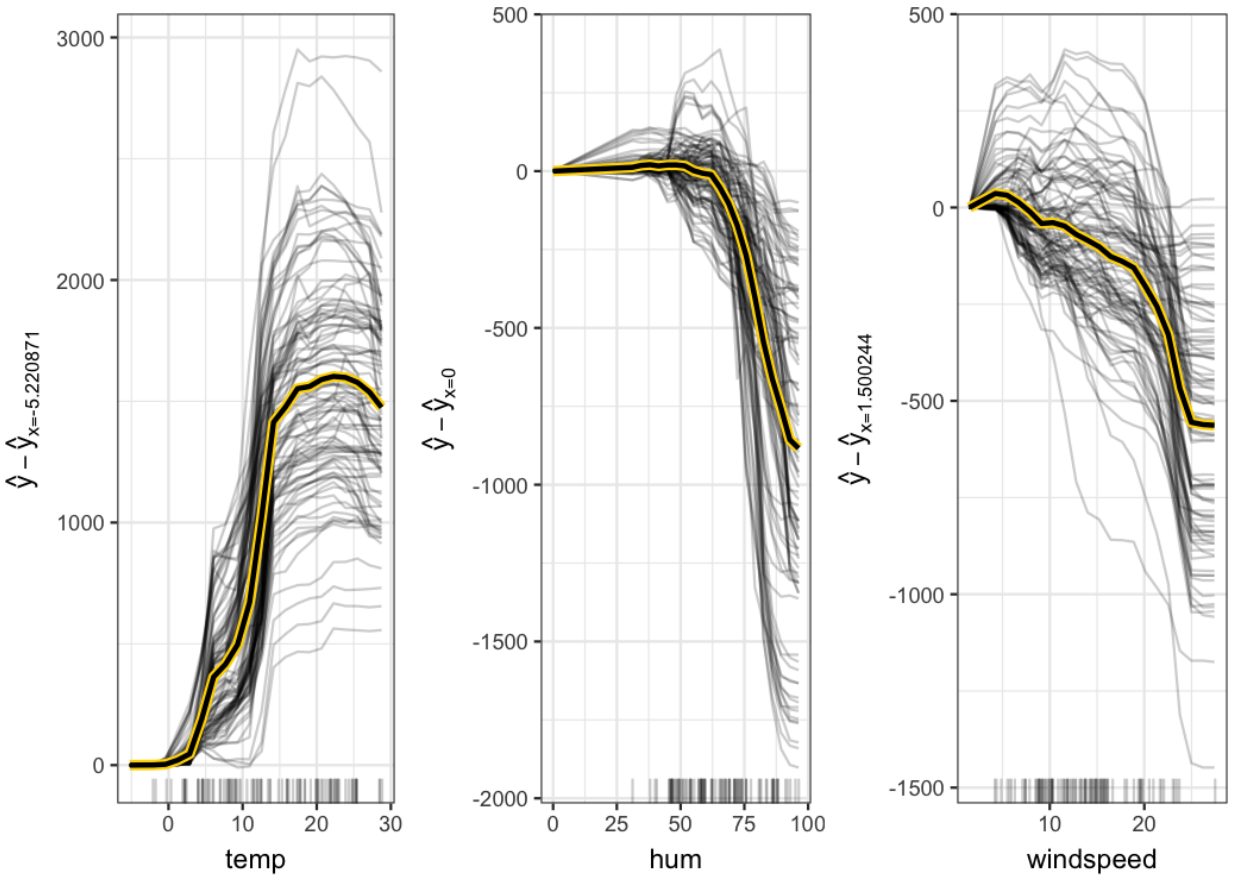
Taking for example the cervical cancer ICE plot for age and centering the lines at the youngest observed age yields:



Centered ICE plot for predicted cervical cancer risk probability by age. The lines are fixed to 0 at age 13 and each point shows the difference to the prediction with age 13. Compared to age 18, the predictions for most instances stay the same and see an increase up to 20 percent. A few cases show the opposite behavior: The predicted probability decreases with increasing age.

With the centered ICE plots it is easier to compare the curves of individual instances. This can be useful when we are not interested in seeing the absolute change of a predicted value, but rather the difference in prediction compared to a fixed point of the feature range.

The same for the bike dataset and count prediction model:



Centred individual conditional expectation plots of expected bike rentals by weather condition. The lines were fixed at value 0 for each feature and instance. The lines show the difference in prediction compared to the prediction with the respective feature value at their minimal feature value in the data.

Derivative ICE Plot

Another way to make it visually easier to spot heterogeneity is to look at the individual derivatives of $\hat{f}$ with respect to x_S instead of the predicted response $\hat{f}$. The resulting plot is called derivative ICE plot (d-ICE). The derivatives of a function (or curve) tell you in which direction changes occur and if any occur at all. With the derivative ICE plot it is easy to spot value ranges in a feature where the black box's predicted values change for (at least some) instances. If there is no interaction between x_S and x_C, then $\hat{f}$ 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 because of interactions and it will become visible in the d-ICE plot. In addition to displaying the individual curves for derivative $\hat{f}$, showing the standard deviation of derivative $\hat{f}$ helps to highlight regions in x_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 when we vary the feature of interest.
- In contrast to partial dependence plots they can **uncover heterogeneous relationships**.

Disadvantages

- ICE curves **can only display one feature** meaningfully, because two features would require drawing multiple, overlaying surfaces and there is no way you would still see anything in the plot.
- ICE curves suffer from the same problem as PDPs: When the feature of interest is correlated with the other features, then **not all points in the lines might be valid data points** according to the joint feature distribution.
- When many ICE curves are drawn the plot **can become overcrowded** and you don't see anything any more. The solution: either add some transparency to the lines or only draw a sample of the lines.
- In ICE plots it might not be easy to **see the average**. This has a simple solution: just combine individual conditional expectation curves with the partial dependence plot.

Feature Interaction

When features in a prediction model interact with each other, then the influence of the features on the prediction is not additive but more complex. It follows that Aristotle's predicate "the whole is greater than the sum of its parts." only applies in the presence of feature interactions.

Feature Interaction?

When a machine learning model makes a prediction based on two features, we can decompose the prediction into four terms: a constant term, one term for the first feature, one for the second feature and one for the interaction effect between the two features.
The interaction between two features is the change in the prediction that occurs by varying the features, after having accounted for the individual feature effects. It's the effect that comes on top of the sum of 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, amounting to four possible predictions:

Location	Size	Predicted value
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. Making a small house big always adds 100,000 to the predicted value, no matter the location. Also the difference in predicted value between a good and a bad location is 50,000, independent of the size.

Now let's consider an example with interaction:

Location	Size	Predicted value
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 extra term for the interaction: +100,000 if the house is big and in a good location. This is an interaction between the size and the location, because in this case, the difference

in predicted value 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 predicted outcome depends on the interaction of the features. This measurement is called H-statistic, introduced by Friedman and Popescu (2008)[47]

Theory: Friedman's H-statistic

We will look into two cases: The interaction between two features, which tells us if and how strongly two specific features interact with each other in the model; The interaction between a feature and all other features, which tells us if and how strongly (in total) the specific feature interacts in the model with all the other features. In theory, arbitrary interactions between any number of features can be measured, but those two cases represent the most interesting interaction cases.

If two features x_j and x_k don't interact, we can decompose the partial dependence function in the following way (assuming that 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.

Similarly, if a feature x_j 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 only depends on x_j and the second depends on all other features excluding x_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 excluding x_j.

This decomposition expresses the partial dependence (or full prediction) function without interactions (between features x_j and x_k or, respectively, x_j and x_{-j}). 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 (for measuring the interaction between two features) or of the complete function (for measuring the interaction between a feature and all other features). The amount of the variance that is explained by the interaction (difference between observed and no-interaction PD) is used as the interaction strength statistic. The statistic is 0 when 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.

In mathematical terms, the H-statistic for the interaction between feature x_j and x_k proposed by Friedman and Popescu is:

[47]Friedman, Jerome H, and Bogdan E Popescu. 2008. "Predictive Learning via Rule Ensembles." The Annals of Applied Statistics. JSTOR, 916–54.

$$H_{jk}^2 = \sum_{i=1}^{n} \left[PD_{jk}(x_j^{(i)}, x_k^{(i)}) - PD_j(x_j^{(i)}) - PD_k(x_k^{(i)}) \right] / \sum_{i=1}^{n} PD_{jk}^2(x_j^{(i)}, x_k^{(i)})$$

Similarly for measuring if a feature x_j interacts with any other feature:

$$H_j^2 = \sum_{i=1}^{n} \left[\hat{f}(x^{(i)}) - PD_j(x_j^{(i)}) - PD_{-j}(x_{-j}^{(i)}) \right] / \sum_{i=1}^{n} \hat{f}^2(x^{(i)})$$

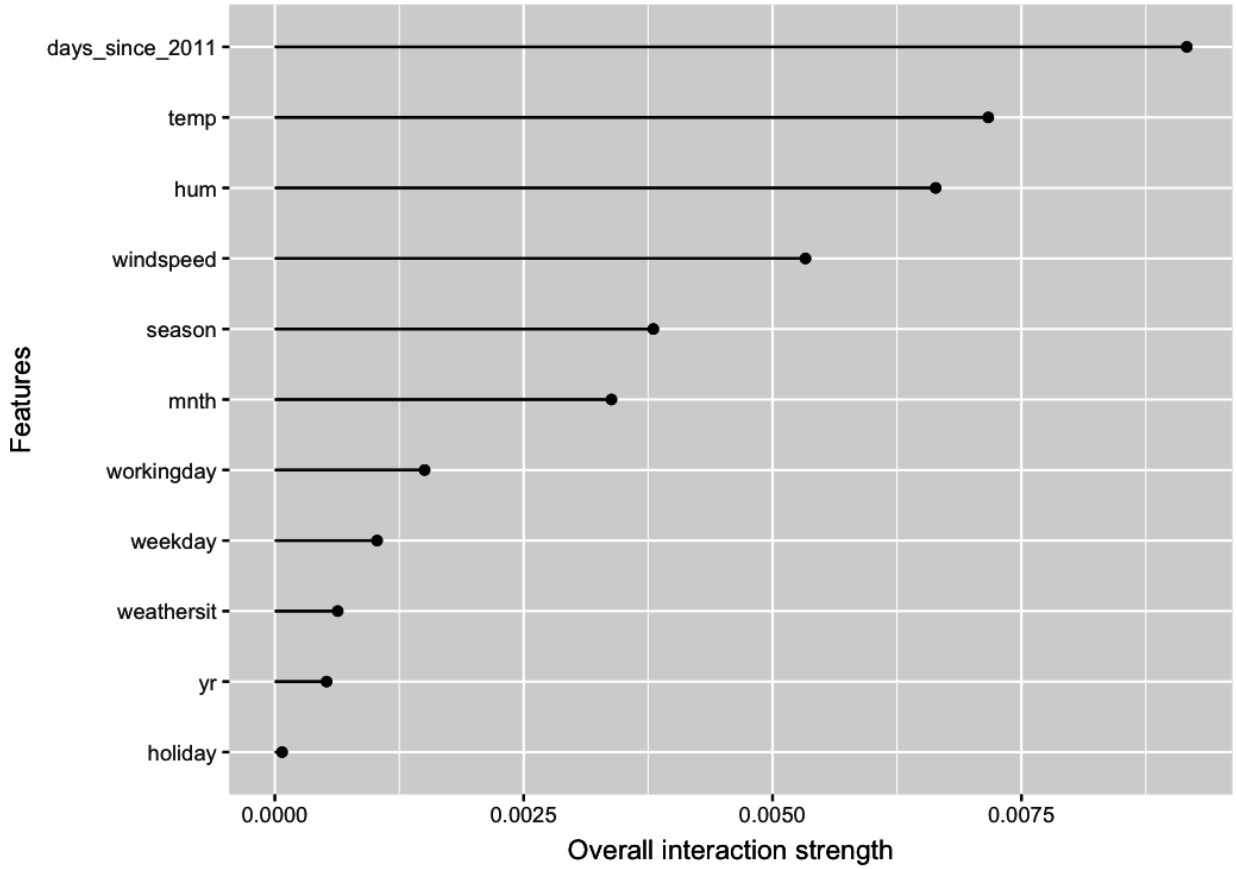
The H-statistic is expensive to evaluate, because it iterates over all data points and at each point the partial dependence has to be evaluated which is done using - again - all n data points. In the worst case, we need $2n^2$ calls to the machine learning models predict function to compute the H_j^2-statistic and $3n^2$ for the H_{jk}^2-statistic. To speed up the computation, we can sample from the n data points . This has the drawback that it adds variance to the partial dependence estimates, which makes the statistic unstable. So when you are using sampling to reduce the computational burden, make sure to sample enough data points.

In their paper, Friedman and Popescu also propose a test for the H-statistic being significantly different from zero: The Null hypothesis is the absence of interaction. To generate the interaction statistic under the Null hypothesis, you have to be able to adjust the model so that it has no interaction between feature x_j and x_k or all others. This is not possible for all types of models, so running this test is model-specific and not model-agnostic and as such not covered here.

The interaction strength statistic can also be applied in a classification setting, when the predicted outcome is the probability for a class.

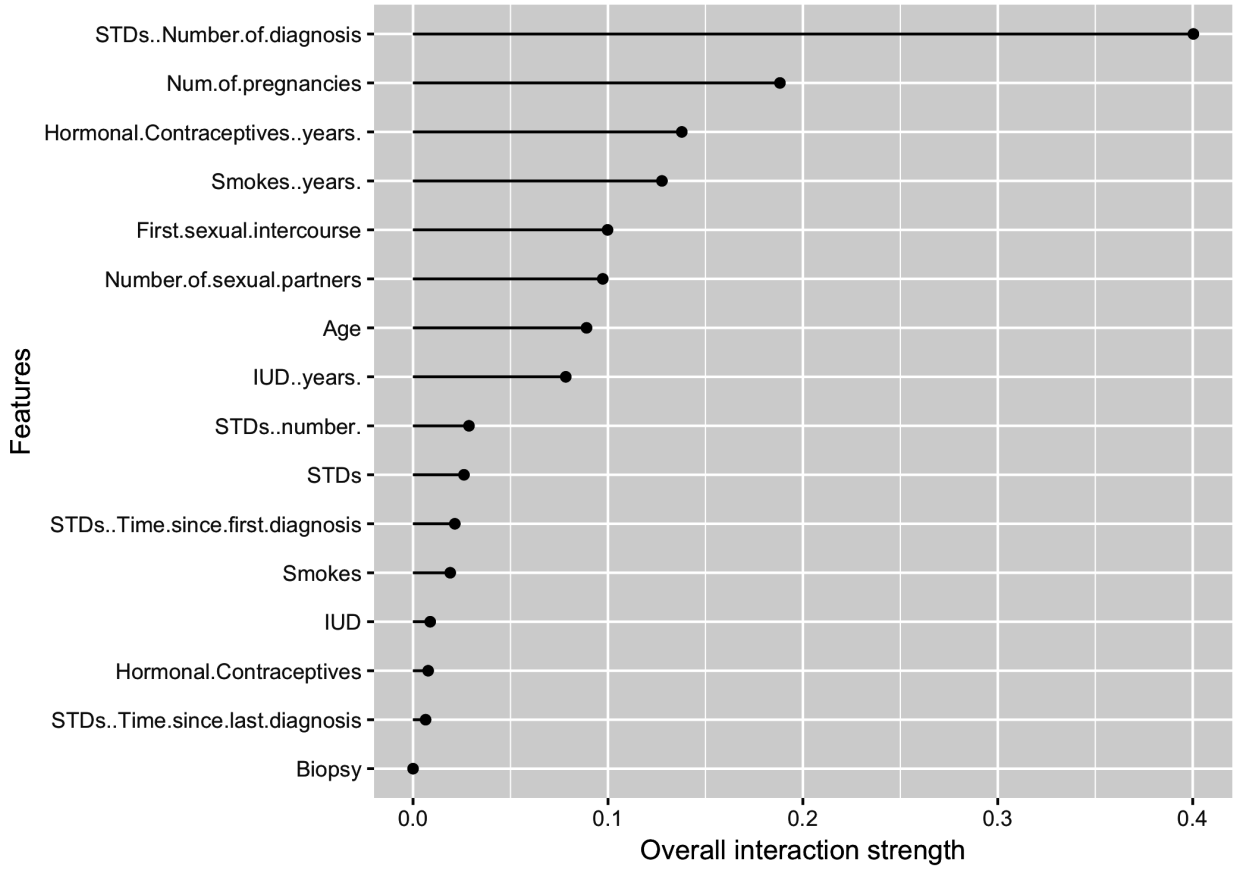
Examples

Let's see how feature interactions look like in practice! We measure the interaction strength of features in a support vector machine that predicts the number of bike rentals, given weather and calendrical features. The following plot shows the results of the feature interaction analysis:

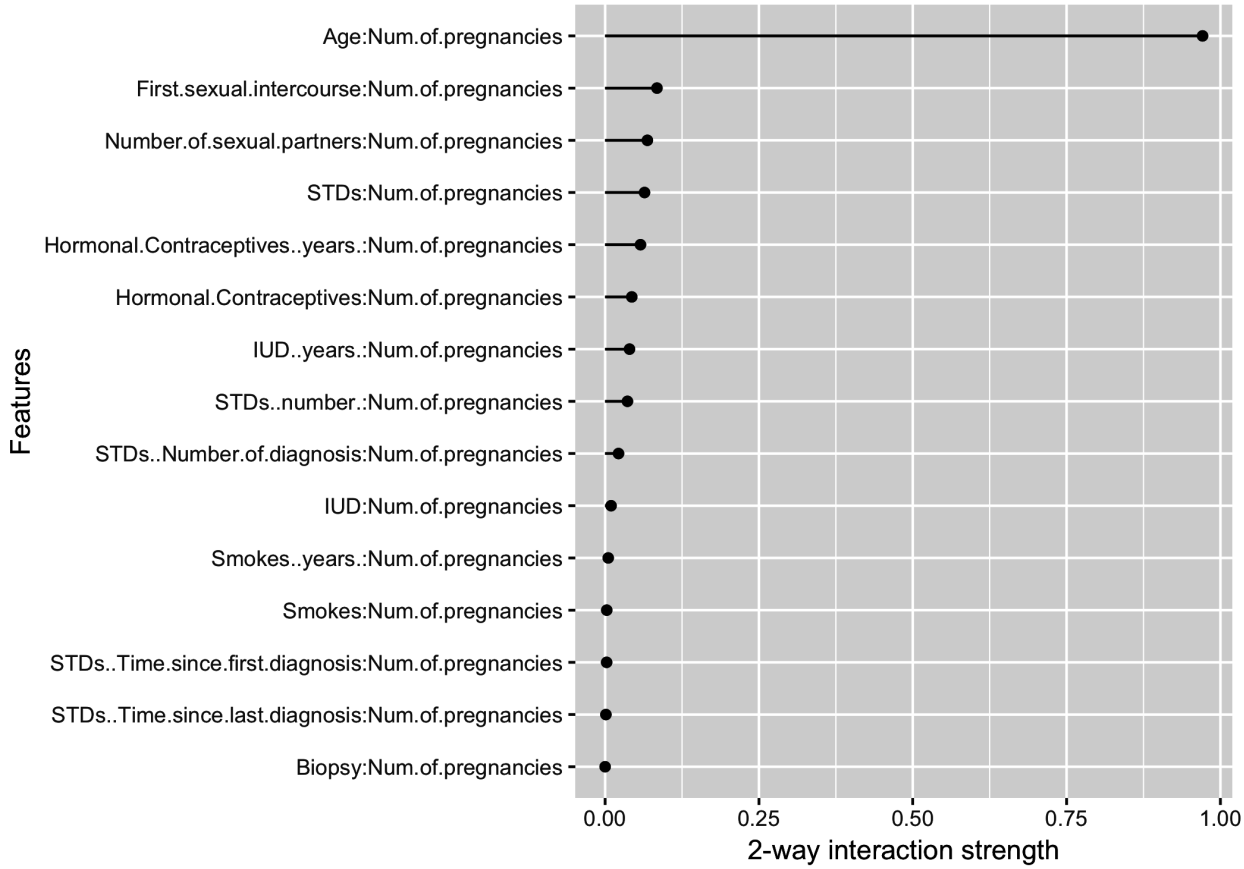


The interaction strength for each feature with all other features for a support vector machine predicting bike rentals. Overall the interaction effects between the features are very weak (below 1 percent of variance explained by each feature).

In this next example we calculate the interaction statistic for a classification problem, where we deal with the partial dependence of the predicted probability. We analyse the interactions between features in a random forest that is trained to predict cervical cancer, given some risk factors.



After looking into the feature interactions of each feature with all other features, we can pick one of the features and specifically dive deeper into all the 2-way interactions between the chosen feature with the other features:



The 2-way interactions between number of pregnancies with each other feature. There is a strong interaction between the number of pregnancies and the age.

Advantages

- The interaction statistic has an **underlying theory** through the partial dependence decomposition.
- The H-statistic has a **meaningful interpretation**: The interaction is defined as the portion of variance 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 a specific form.
- With the H-statistic it is also possible to analyze arbitrary **higher interactions**: For example 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's **computationally expensive**.

- Sometimes the results are weird and for small simulations **don't yield the expected results**. But this is more anecdotal evidence.
- The computation involves estimating marginal distributions. These **estimates also have some variance**, when we don't use all of the data points. This means when we sample points, the estimates will also vary from run to run and the results might **become unstable**. I recommend repeating it a few times to see if you have enough data included for a stable result.
- It is unclear whether an interaction is significantly bigger than 0. We would need to conduct a statistical test, but this **test is not (yet) available in a model-agnostic version**.
- Connected to the testing problem: It's hard to tell when the H-statistic is large enough that we would consider it a strong interaction.
- The H-statistic tells us how strong the interactions are, but it doesn't tell us how the interaction is shaped. That's 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 in which you are interested.
- The H-statistic can't be meaningfully applied if the inputs are pixels.
- The interaction statistic works under the assumption that we can independently shuffle features (the same problem that partial dependence plots have). When the features strongly correlate, the assumption is violated and **we integrate over feature combinations that are very unlikely in reality**.

Implementations

- For the examples in this book, I used the R package `iml`, which is available on CRAN[48] and the development version on Github[49].
- There are other implementations, which focus on a specific model:
  - The R package pre[50] implements the RuleFit algorithm plus the H-statistic.
  - The R package gbm[51] implements gradient boosted models plus the H-statistic.

Alternatives

The H-statistic is not the only way to measure interactions, there is also:

- Variable Interaction Networks (VIN) by Hooker (2004)[52]: An approach that decomposes the prediction function into main effects and feature interactions. The interactions between features are then visualized as a network. Unfortunately, there is no software available yet.

[48] https://cran.r-project.org/web/packages/iml
[49] https://github.com/christophM/iml
[50] https://cran.r-project.org/web/packages/pre/index.html
[51] https://github.com/gbm-developers/gbm3
[52] Hooker, G. (2004). Discovering Additive Structure in Black Box Functions. Knowledge Discovery and Data Mining, 575â€“580. http://doi.org/10.1145/1014052.1014122

- Partial dependence based feature interaction by Greenwell et. al (2018)[53]: For measuring the interaction between two features, this approach measures the feature importance (defined as the variance of the 1D partial dependence function) of one feature conditional on different, fixed points of the other feature. When the variance is high, then the features interact with each other, if it is zero, they don't interact. The R package `vip` is available on Github[54]. They also cover partial dependence plots and feature importance.

[53]Greenwell, B. M., Boehmke, B. C., & McCarthy, A. J. (2018). A Simple and Effective Model-Based Variable Importance Measure, 1â€“27. Retrieved from http://arxiv.org/abs/1805.04755

[54]https://github.com/koalaverse/vip

Feature Importance

A feature's importance is the increase in the model's prediction error after we permuted the feature's values (breaks the relationship between the feature and the outcome).

The Theory

The concept is really straightforward: We measure a feature's importance by calculating the increase of the model's prediction error after permuting the feature. A feature is "important" if permuting its values increases the model error, because the model relied on the feature for the prediction. A feature is "unimportant" if permuting its values keeps the model error unchanged, because the model ignored the feature for the prediction. The permutation feature importance measurement was introduced for Random Forests by Breiman (2001)[55]. Based on this idea, Fisher, Rudin, and Dominici (2018)[56] proposed a model-agnostic version of the feature importance - they called it model reliance. They also introduce more advanced ideas about feature importance, for example a (model-specific) version that accounts for the fact that many prediction models may fit the data well. Their paper is worth a read.

The permutation feature importance algorithm based on Breiman (2011) and Fisher, Rudin, and Dominici (2018):

Input: Trained model $\hat{f}$, feature matrix X, target vector Y, error measure $L(Y, \hat{Y})$

1. Estimate the original model error $e_{orig}(\hat{f}) = L(Y, \hat{f}(X))$ (e.g. mean squared error)
2. For each feature $j \in 1, \ldots, p$ do
    - Generate feature matrix X_{perm_j} by permuting feature X_j in X. This breaks the association between X_j and Y.
    - Estimate error $e_{perm} = L(Y, \hat{f}(X_{perm_j}))$ based on the predictions of the permuted data.
    - Calculate permutation feature importance $FI_j = e_{perm}(\hat{f})/e_{orig}(\hat{f})$. Alternatively, the difference can be used: $FI_j = e_{perm}(\hat{f}) - e_{orig}(\hat{f})$
3. Sort variables by descending FI.

In their paper, Fisher, Rudin, and Dominici (2018) propose to split the dataset in half and exchange the X_j values of the two halves instead of permuting X_j. This is exactly the same as permuting the feature X_j if you think about it. If you want to have a more accurate estimate, you can estimate the error of permuting X_j by pairing each instance with the X_j value 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 big amount of computation time. I can only recommend using the $n(n-1)$ - method when you are serious about getting extremely accurate estimates.

[55]Breiman, Leo. 2001. "Random Forests." Machine Learning 45 (1). Springer: 5–32.

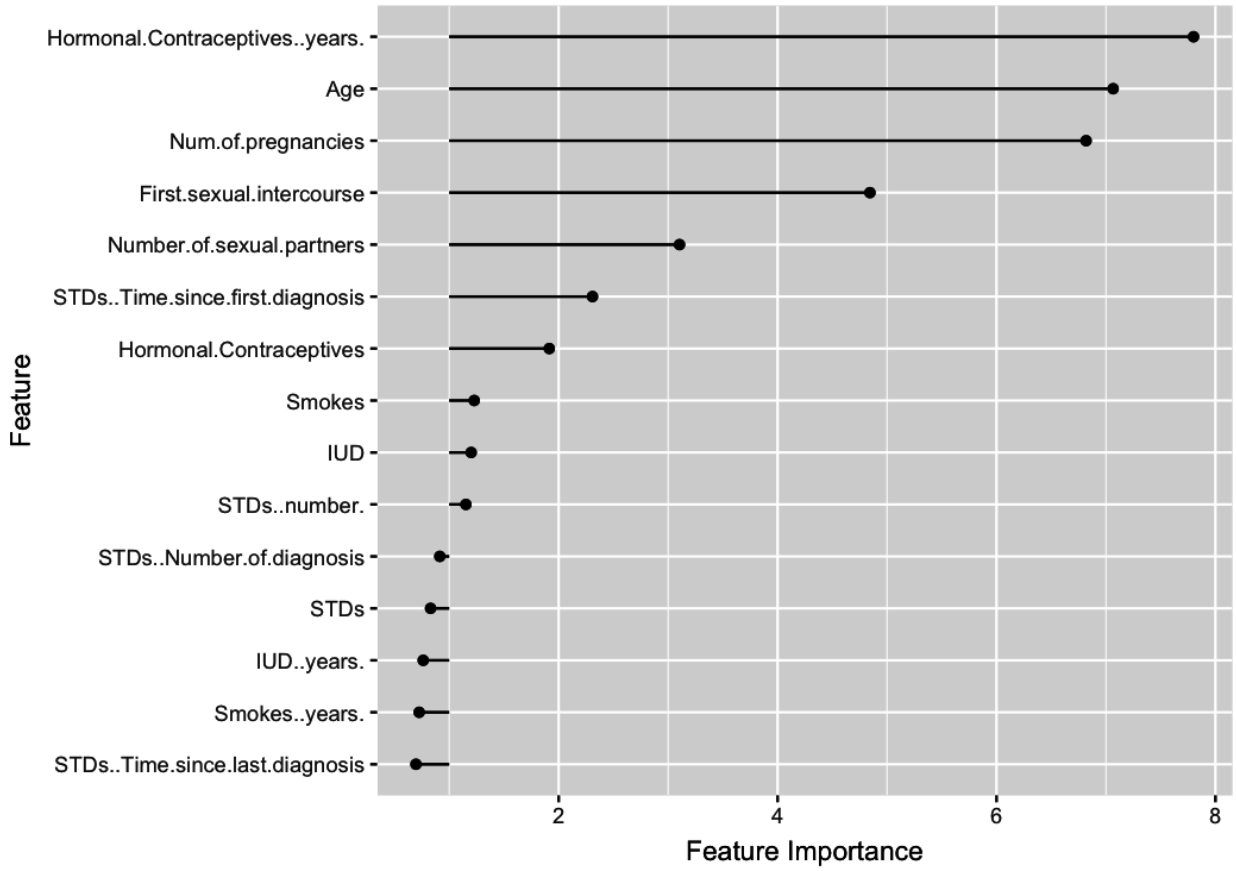
[56]Fisher, Aaron, Cynthia Rudin, and Francesca Dominici. 2018. "Model Class Reliance: Variable Importance Measures for any Machine Learning Model Class, from the 'Rashomon' Perspective." http://arxiv.org/abs/1801.01489.

Example and Interpretation

We 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: $1 - AUC$ (one minus the area under the ROC curve). Features that are associated model error increase by a factor of 1 (= no change) were not important for predicting cervical cancer.

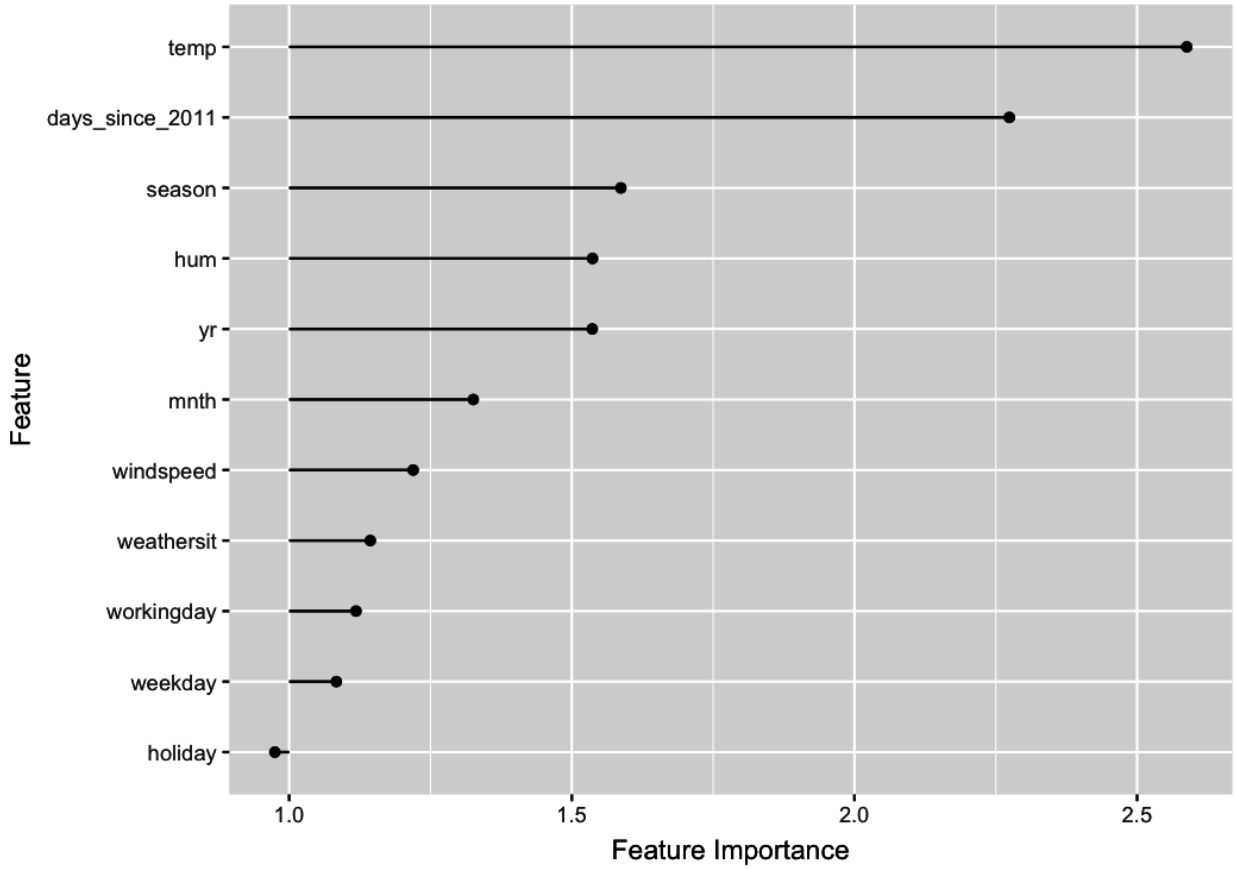


The importance for each of the features in predicting cervical cancer with a random forest. The importance is the factor by which the error is increased compared to the original model error.

The feature with the highest importance was associated with an error increase of 7.8 after permutation.

Bike rentals (Regression)

We fit a support vector machine model to predict bike rentals, given weather conditions and calendric information. As error measurement we use the mean absolute error.



The importance for each of the features in predicting bike rentals with a support vector machine.

Advantages

- Nice interpretation: Feature importance is the increase of 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.

Disadvantages

- The feature importance measure is tied to the error of the model. This is not inherently bad, but in some cases not what you need. In some cases you would prefer to know how much the model's output varies for one feature, ignoring what it means for the performance. For example: You want to find out how robust your model's output is, given someone manipulates the features. In this case, you wouldn't be interested in how much the model performance drops given the permutation of a feature, but rather 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 doesn't overfit).

- You need access to the actual outcome target. If someone only gives you the model and unlabeled data - but not the actual target - you can't compute the permutation feature importance.

Global Surrogate Models

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: When an outcome of interest is expensive, time-consuming or otherwise difficult to measure (for example because it comes from a complex computational simulation), a cheap and fast surrogate model of the outcome is used instead. The difference between the surrogate models used in engineering and for interpretable machine learning is that the underlying model is a machine learning model (not a simulation) and that the surrogate model has to be interpretable. The purpose of (interpretable) surrogate models is to approximate the predictions of the underlying model as closely as possible while being interpretable. You will find the idea of surrogate models under a variety of names: Approximation model, metamodel, response surface model, emulator, ...

So, about the theory: there is actually not much theory needed to understand surrogate models. We want to approximate our black box prediction function $\hat{f}(x)$ as closely as possible with the surrogate model prediction function $\hat{g}(x)$, under the constraint that g is interpretable. Any interpretable model - for example from the interpretable models chapter - can be used for the function g:

For example a linear model:

$$\hat{g}(x) = \beta_0 + \beta_1 x_1 + \ldots + \beta_p x_p$$

Or a decision tree:

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

Fitting a surrogate model is a model-agnostic method, since it requires no information about the inner workings of the black box model, only the relation of input and predicted output is used. If the underlying machine learning model would be exchanged for another, you could still apply the surrogate method. The choice of the black box model type and of the surrogate model type is decoupled.

Perform the following steps to get a surrogate model:

1. Choose a dataset X. This could be the same dataset that was used for training the black box model or a new dataset from the same distribution. You could even choose a subset of the data or a grid of points, depending on your application.

2. For the chosen dataset X, get the predictions $\hat{y}$ of the black box model.
3. Choose an interpretable model (linear model, decision tree, ...).
4. Train the interpretable model on the dataset X and its predictions $\hat{y}$.
5. Congratulations! You now have a surrogate model.
6. Measure how well the surrogate model replicates the prediction of the black box model.
7. Interpret / visualize the surrogate model.

You might find approaches for surrogate models which have some extra steps or differ a bit, but the general idea is usually the same as described here.

A 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 and respectively $\hat{y}_i$ of the black box model. The mean of the black box model predictions is $\bar{\hat{y}}$. 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 interpretable model. If the R squared is close to 1 (= low SSE), then the interpretable model approximates the behaviour of the black box model very well. If the interpretable model is that 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 haven't talked about the model performance of the underlying black box model, meaning how well or badly it performs at predicting the real outcome. For fitting the surrogate model, the performance of the black box model does not matter at all. 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 sucks, because then the black box model itself is irrelevant.

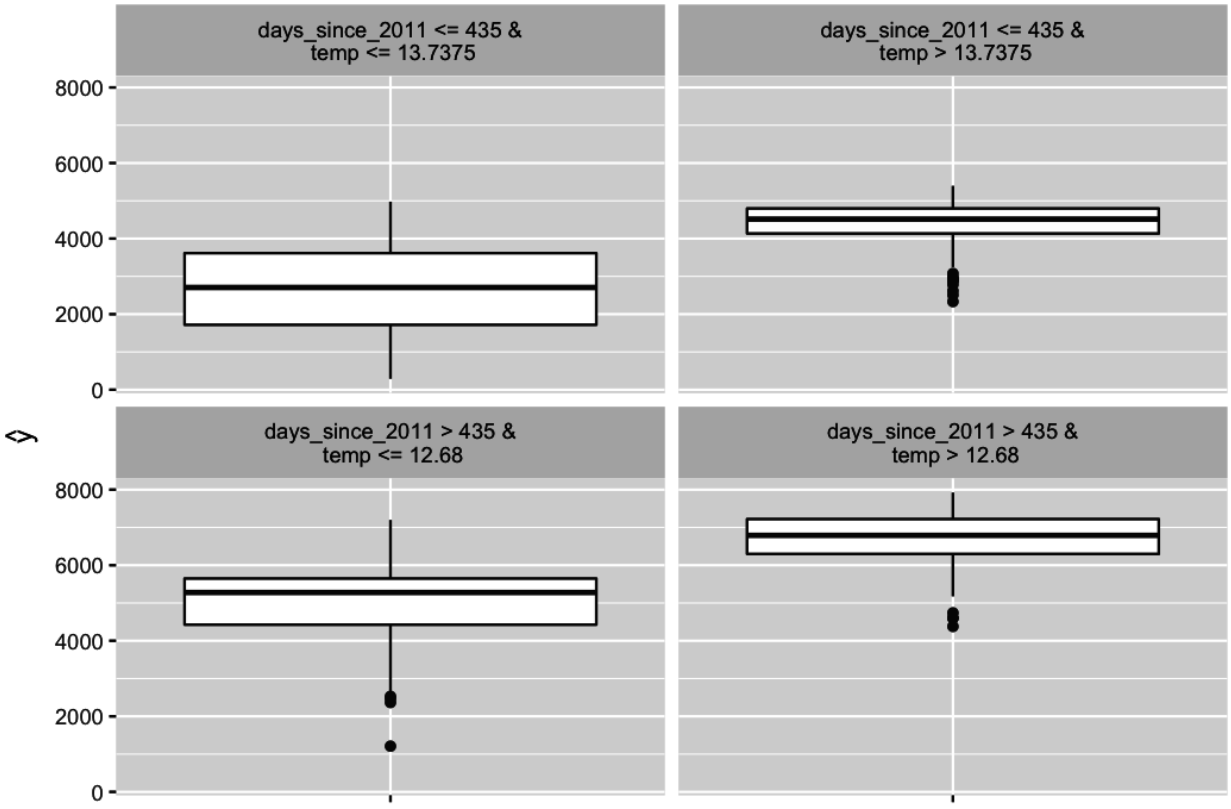
We could also build a surrogate model based on a subset of the original data or re-weight the instances. In this way, we change the distribution of the surrogate model's input, which changes the focus of the interpretation (then it is not really global any longer). When we weight the data locally around a certain instance of the data (the closer the instances to the chosen instance, the higher their weight) we get a local surrogate model, which can be used to explain the instance's individual prediction. Learn more about local models in the following chapter.

Example

To demonstrate the surrogate models, we look at a regression and a classification example.

First, we fit a support vector machine to predict the daily number of bike rentals given weather and calendrical information. The support vector machine is not very interpretable, so we fit a surrogate

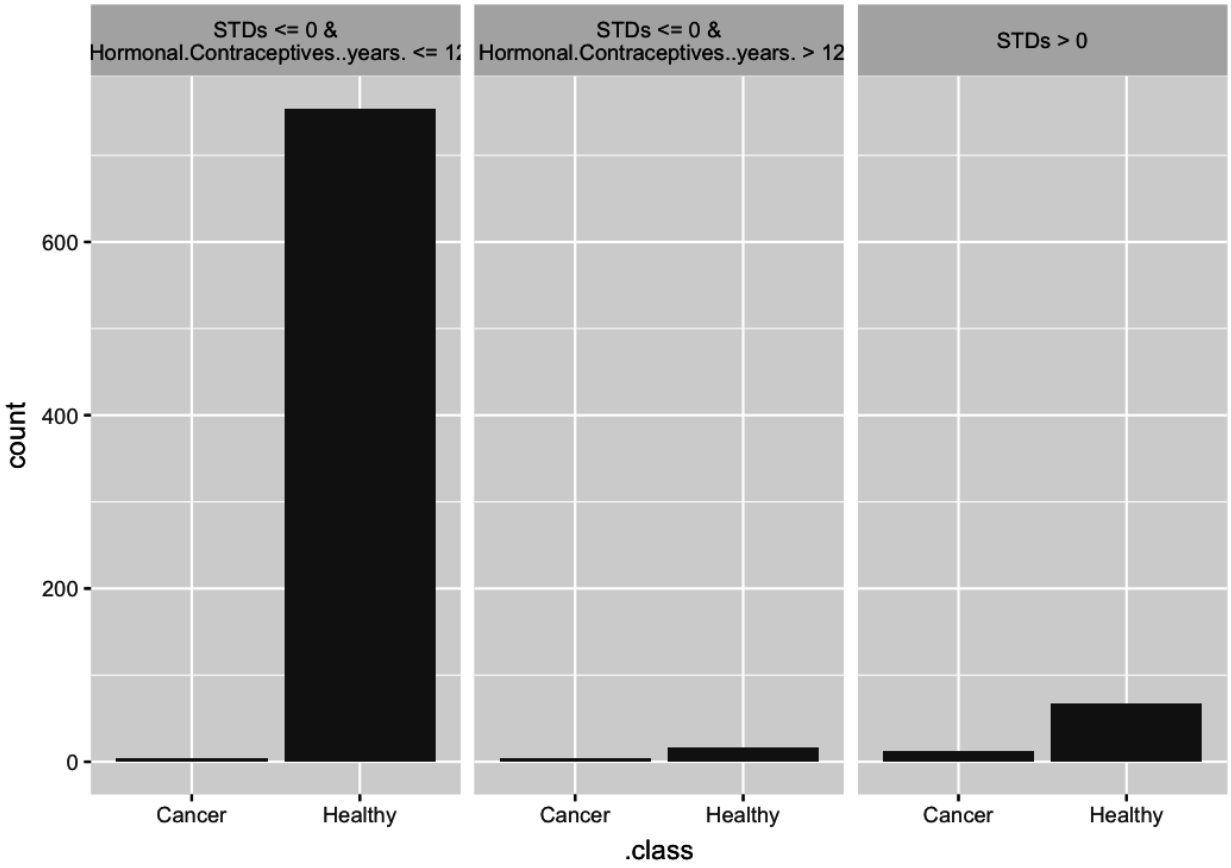
with a CART decision tree as interpretable model to approximate the behaviour of the support vector machine.



The terminal nodes of a surrogate tree that approximates the behaviour 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 bike rentals when the weather is above around 13 degrees (Celsius) and when the day was later in the 2 year period (cut point at 435 days).

The surrogate model has an R squared (variance explained) of 0.77 which means it approximates the underlying black box behaviour quite well, but not perfectly. If the fit would be perfect, we could actually throw away the support vector machine and use the tree instead.

In our second example, we predict the probability for cervical cancer with a random forest. Again we fit a decision tree, using 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 behaviour of a random forest trained on the cervical cancer dataset. The counts in the nodes show the distribution of the black box models classifications in the nodes.

The surrogate model has an R squared (variance explained) of 0.2 which means it doesn't approximate the random forest well and we should not over-interpret 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 swap not only the interpretable model, but also the underlying black box model. Let's say you build some complex model and you want to explain it to different teams in your company. One team is familiar with linear models; the other team prefers decision trees. You can fit 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 don't have to change the way you do interpretation, because you can use the same class of surrogate models.
- I'd 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 squard measure, we can easily **measure** how good our surrogate models are in terms of approximation of the black box predictions.

Disadvantages

- Be careful to draw **conclusions about the model**, **not the data**, since the surrogate model never sees the real outcome.
- It's 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's assume we are not very close, but close enough. It could happen that the interpretable model is very **close for a subset of the dataset**, **but wildly diverges 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 this method is, of course, not for you.

Local Surrogate Models (LIME)

Local interpretable model-agnostic explanations (LIME) (Ribeiro, M.T., Singh, S. and Guestrin, C., 2016[57]) is a method for fitting local, interpretable models that can explain single predictions of any black-box machine learning model. LIME explanations are local surrogate models. Surrogate models are interpretable models (like a linear model or decision tree) that are learned on the predictions of the original black box model. But instead of trying to fit a global surrogate model, LIME focuses on fitting local surrogate models to explain why single predictions were made.

The idea is quite intuitive. First of all, forget about the training data and imagine you only have the black box model where you can input data points and get the models predicted outcome. You can probe the box as often as you want. Your goal is to understand why the machine learning model gave the outcome it produced. LIME tests out what happens to the model's predictions when you feed variations of your data into the machine learning model. LIME generates a new dataset consisting of perturbed samples and the associated black box model's predictions. On this dataset LIME then trains an interpretable model weighted by the proximity of the sampled instances to the instance of interest. The interpretable model can basically be anything from this chapter, for example LASSO or a decision tree. The learned model should be a good approximation of the machine learning model locally, but it does not have to be so globally. This kind of accuracy is also called local fidelity.

The recipe for fitting local surrogate models:

- Choose 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 by their proximity to the instance of interest.
- Fit a weighted, interpretable model on the dataset with the variations.
- Explain prediction by interpreting the local model.

In the current implementations (R[58] and Python[59]) for example linear regression can be chosen as interpretable surrogate model. Upfront you have to choose K, the number of features that you want to have in your interpretable model. The lower the K, the easier the model is to interpret, higher K potentially creates models with higher fidelity. There are different methods for how to fit models with exactly K features. A solid choice is Lasso. A Lasso model with a high regularisation parameter λ yields a model with only the intercept. By refitting the Lasso models with slowly decreasing λ, one after each other, the features are getting weight estimates different from zero. When K features are in the model, you 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 testing which feature would create the biggest improvement

[57]Ribeiro, M.T., Singh, S. and Guestrin, C., 2016, August. Why should i trust you?: Explaining the predictions of any classifier. In Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (pp. 1135-1144). ACM.
[58]https://github.com/thomasp85/lime
[59]https://github.com/marcotcr/lime

when added or removed, until a model with K features is reached. Other interpretable models like decision trees are also possible.

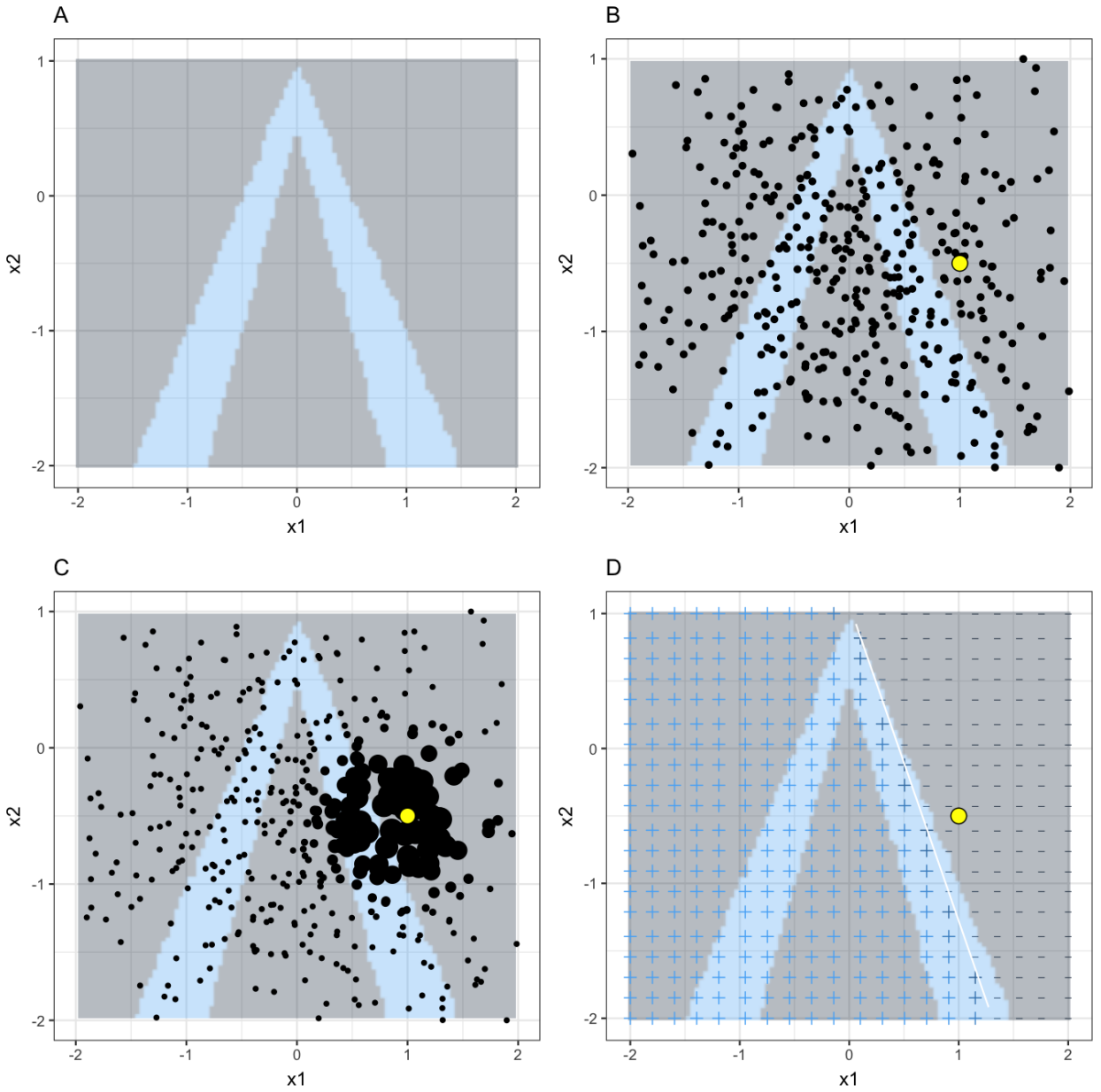
As always, the devil's in the details. In a high-dimensional space, defining a neighbourhood is not trivial. Distance measures are quite arbitrary and distances in different dimensions (aka features) might not be comparable at all. How big should the neighbourhood be? If it is too small, then there might be no difference in the predictions of the machine learning model at all. LIME currently has a hard coded kernel and kernel width, which define the neighbourhood, and there is no answer how to figure out the best kernel or how to find the optimal width. The other question is: How do you get the variations of the data? This differs depending on the type of data, which can be either text, an image or tabular data. For text and image the solution is turning off and on single words or super-pixels. In the case of tabular data, LIME creates new samples by perturbing each feature individually, by drawing from a normal distribution with mean and standard deviation from the feature.

LIME does a good job in creating selective explanations, which humans prefer. That's 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 causal attributions, so I don't see LIME in compliance scenarios, where you are 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 for Tabular Data

Tabular data means any data that comes in tables, where each row represents an instance and each column a feature. LIME sampling is not done around the instance of interest, but from the training data's mass centre, which is problematic. But it increases the likelihood that the outcome for some of the sampled points predictions differ from the data point of interest and that LIME can learn at least some explanation.

It's best to visually explain how the sampling and local model fitting works:



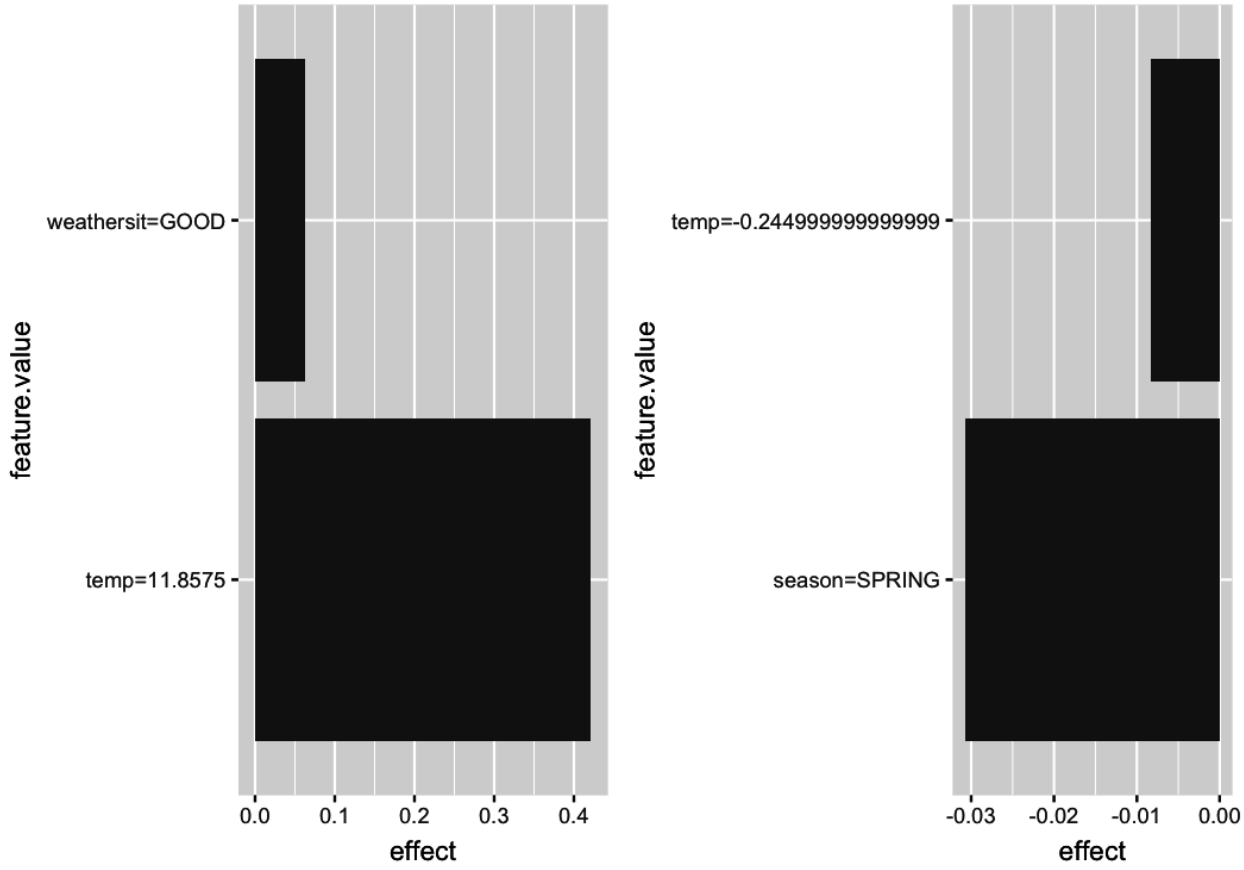
How LIME sampling works: A) The black box model predicts one of two classes given feature x1 and x2. Most data points have class 0 (darker colour), and the ones with class 1 are grouped in an upside-down V-shape (lighter colour). The plot displays the decision boundaries learned by a machine learning model. In this case it was a Random Forest, but it does not matter, because LIME is model-agnostic and we only care about the decision boundaries. B) The yellow point is the instance of interest, which we want to explain. The black dots are data sampled from a normal distribution around the means of the features in the training sample. This needs to be done only once and can be reused for other explanations. C) Introducing locality by giving points near the instance of interest higher weights. D) The colours and signs of the grid display the classifications of the locally learned model form the weighted samples. The white line marks the decision boundary (P(class) = 0.5) at which the classification of the local model changes.

Example

Let's look at a concrete example. We go back to the bike rental data and turn the prediction problem into a classification: After accounting for the trend that the bike rental get's more popular over time we want to know on a given day if the number of rented bikes will be above or below the trend line. You can also interpret 'above' as being above the mean bike counts, but adjusted for the trend.

First we train a Random Forest with 100 trees on the classification task. Given seasonal and weather information, on which day will the number of bike rentals be above the trend-free average?

The explanations are created with 2 features. The results of the sparse local linear model that was fitted 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.

It becomes clear from the figure, that it is easier to interpret categorical features than numerical features. A 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 created differently: Starting from the original text, new texts are created by randomly removing words from it. The dataset is represented with binary features for each word. A feature is 1 if the respective word is included and 0 if it was removed.

Example

In this example we classify spam vs. ham of YouTube comments.

The black box model is a decision tree on the document word matrix. Each comment is one document (= one row) and each column is the number of occurrences of a specific word. Decision trees are easy to understand, but in this case the tree is very deep. Also in the place of this tree there could have been a recurrent neural network or a support vector machine that was trained on the embeddings from word2vec. From the remaining comments two were selected for showing the explanations.

Let's look at two comments of this dataset and the corresponding classes:

	CONTENT	CLASS
267	PSY is a good guy	0
173	For Christmas Song visit my channel! ;)	1

In the next step we create some variations of the datasets, which are 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 indicates that the word is part of this variation and 0 indicates that the word has been removed. The corresponding sentence for the first variation is "`Christmas Song visit my ;)`".

And 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	my	0.000000
2	0.9939759	Christmas	0.000000

The word "channel" points to a high probability of spam.

LIME for Images

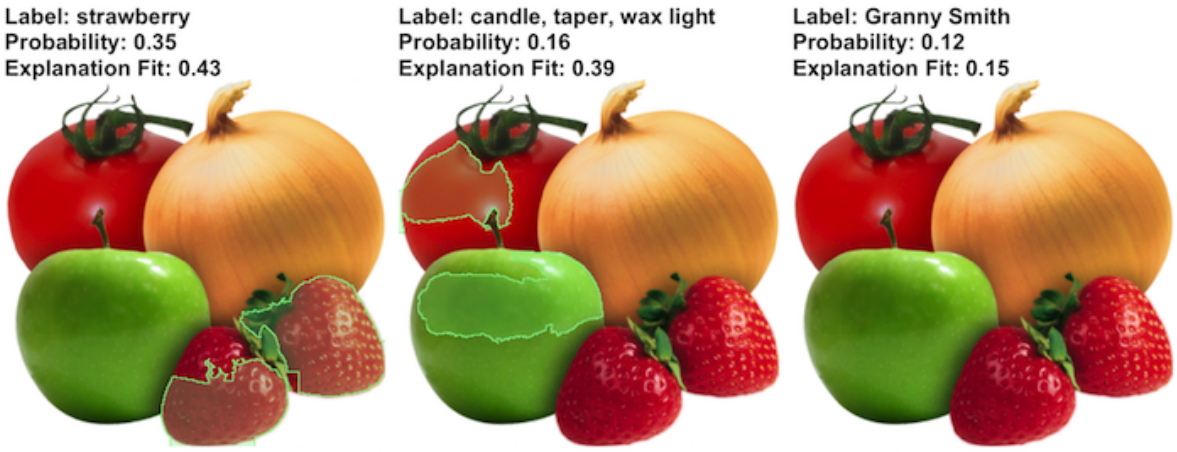
This section was written by Verena Haunschmid.

LIME for images works differently than for tabular data and text. Intuitively it would not make much sense to perturb single pixels, since a lot more than one pixel contribute to one class. By randomly changing individual pixels, the predictions would probably not change much. Therefore, variations of the samples (i.e. images) are created by performing superpixel segmentation and switching superpixels off. Superpixels are connected pixels with similar colors and can be turned off by replacing each pixel by a user provided color (e.g., a reasonable value would by gray). The user can also provide a probability for turning off a superpixel in each permutation.

Example

Since the computation of image explanations is rather slow, the lime R package[60] contains a precomputed example which we will also use to show the output of the method. Image data is great for visualization, the explanations can be displayed directly on the image samples. Since we can have several predicted labels per image (ordered by probability), we can explain the top `n_labels`. For the following image the top 3 predictions were *strawberry*; *candle, taper, wax light*; and *Granny Smith*. The prediction and the explanation in the first case are very reasonable. For the second prediction it is quite interesting to see which part of the image contributed to this class. We could conclude that there were objects labeled as *candle, taper, wax light* in the training set that looked shiny like the tomato.

[60] https://github.com/thomasp85/lime



LIME expalanations for image classification. The example is taken from the lime R package.

Shapley Value Explanations

Predictions can be explained by assuming that each feature 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.

The general idea

Assume the following scenario:

You 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, with a park nearby and cats are forbidden:

The predicted price for our apartment is 300,000€. It's a 50 square meter apartment on the second floor. There is a park nearby and cats are forbidden. Our goal is to explain how each of these features values contributed towards the predicted price of 300k€.

The average prediction for all apartments is 310,000€. How much did each feature value contribute to the prediction compared to the average prediction?

The answer is easy for linear regression models: The effect of each feature is the weight of the feature times the feature value minus the average effect of all apartments: This works only because of the linearity of the model. For more complex model we need a different solution. For example LIME suggests local models to estimate effects.

A different solution comes from cooperative game theory: The Shapley value, coined by Shapley (1953)[61], is a method for assigning payouts to players depending on their contribution towards the

[61]Shapley, Lloyd S. 1953. "A Value for N-Person Games." Contributions to the Theory of Games 2 (28): 307–17.

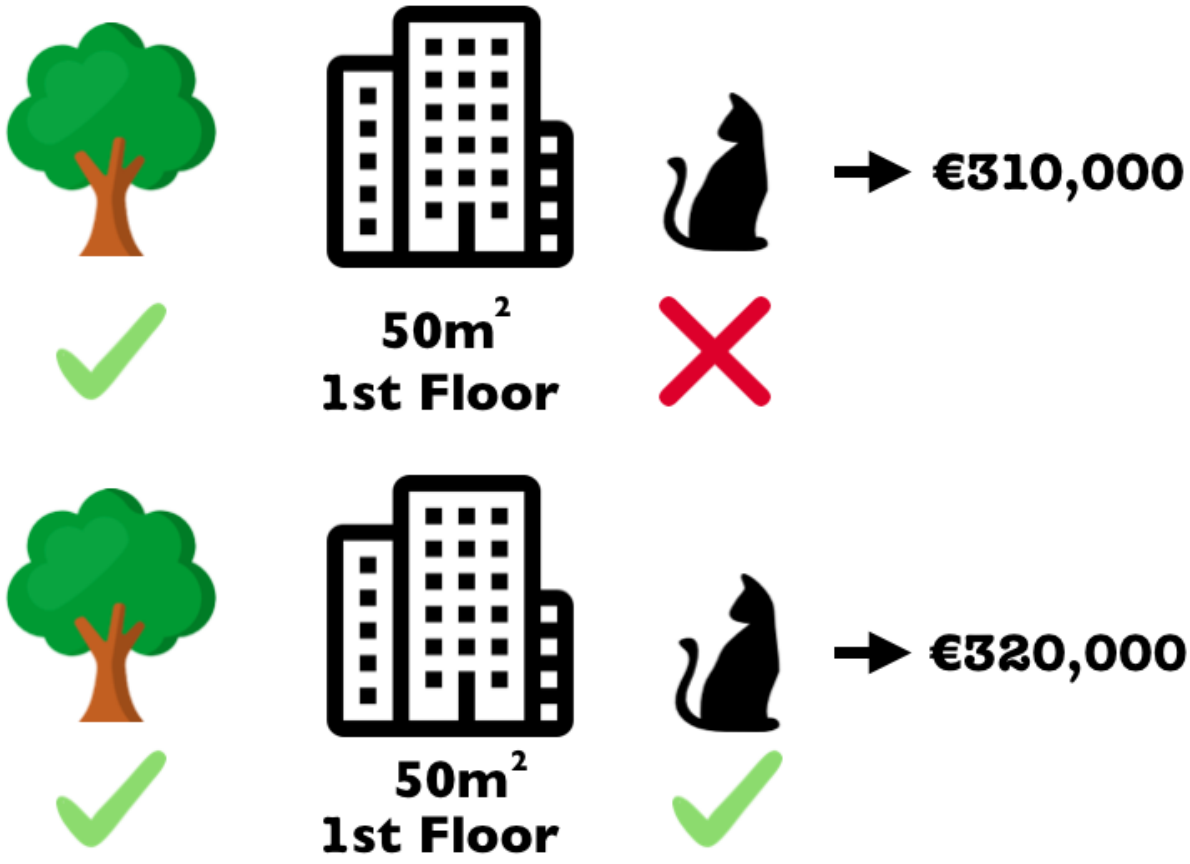
total payout. Players cooperate in a coalition and obtain a certain gain from that cooperation.

Players? Game? Payout? What's the connection to machine learning prediction 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 of all instances. The 'players' are the feature values of the instance, which collaborate to receive the gain (= predict a certain value). In our apartment example, the feature values 'park-allowed', 'cat-forbidden', 'area-50m²' and 'floor-2nd' worked together to achieve the prediction of 300,000€. Our goal is to explain the difference of the actual prediction (300,000€) and the average prediction (310,000€): a difference of -10,000€.

The answer might be: The 'park-nearby' contributed 30,000€; 'size-50m²' contributed 10,000€; 'floor-2nd' contributed 0€; 'cat-forbidden' contributed -50,000€. The contributions add up to -10,000€: the final prediction minus the average predicted apartment price.

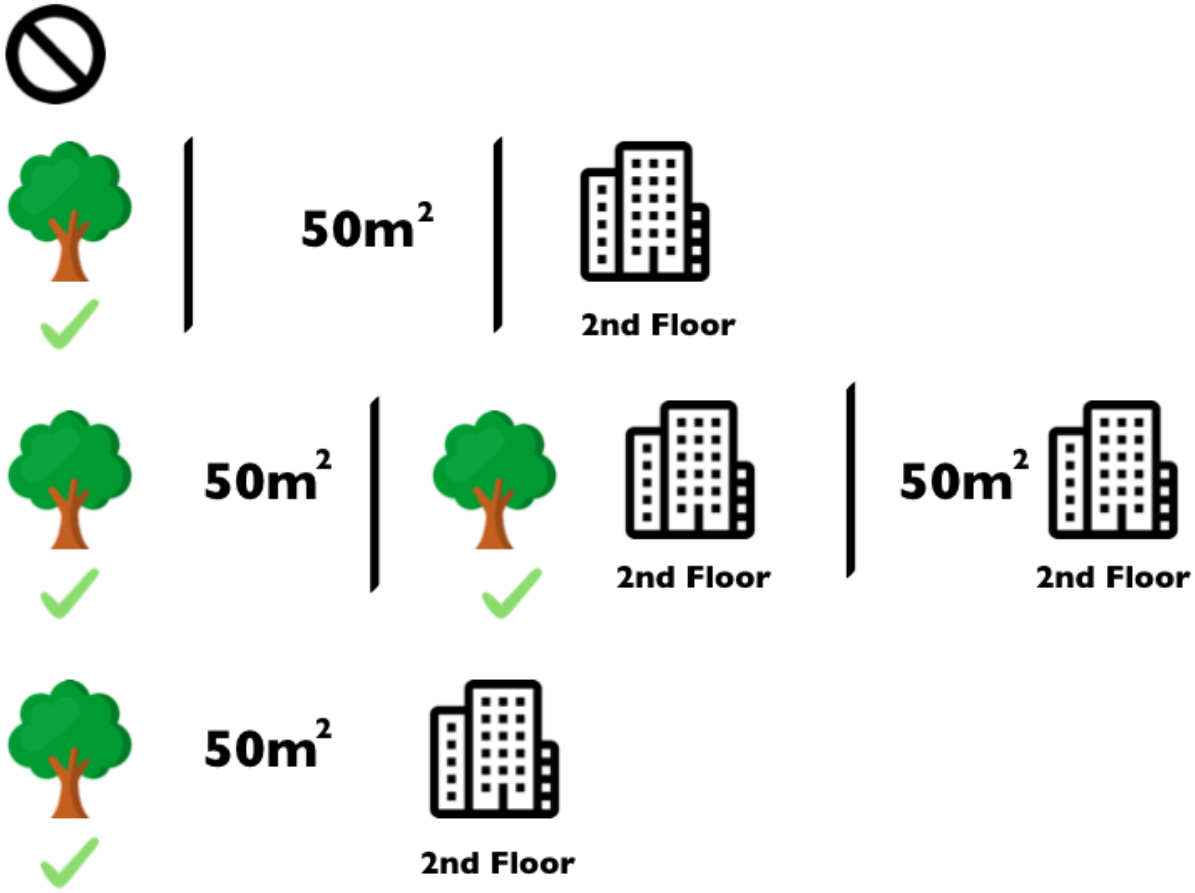
How do we calculate the Shapley value for one feature?

The Shapley value is the average marginal contribution of a feature value over all possible coalitions. All clear now? How to compute the contribution of a feature value to a coalition:



We assess the contribution of the 'cat-forbidden' feature value when added to a coalition of 'park-nearby', 'size-50m²'. We simulate that only 'park-nearby', 'cat-forbidden' and 'size-50m²' are in a coalition by randomly drawing the value for the floor feature. Then we predict the price of the apartment with this combination (310,000€). In a second step we remove 'cat-forbidden' from the coalition by replacing it with a random value of the cat allowed/forbidden feature from the randomly drawn apartment. In the example it was 'cat-allowed', but it could have been 'cat-forbidden' again. We predict the apartment price for the coalition of 'park-nearby' and 'size-50m²' (320,000€). The contribution of 'cat-forbidden' was 310,000€ - 320,000€ = -10.000€. This estimation depends on the sampled non-participating feature values and we get better estimates by repeating this procedure. This figure shows the computation of the marginal contribution for only one coalition. The Shapley value is the weighted average of marginal contributions over all coalitions.

We repeat this computation for all possible coalitions. The computation time increases exponentially with the number of features, so we have to sample from all possible coalitions. The Shapley value is the average over all the marginal contributions. Here are all coalitions for computing the Shapley value of the 'cat-forbidden' feature value:



All coalitions of feature values that are needed to assess the Shapley value for 'cat-forbidden'. The first row shows the coalition without any feature values. The 2nd, 3rd and 4th row show different coalitions - separated by '|' - with increasing coalition size. For each of those coalitions we compute the predicted apartment price with and without the 'cat-forbidden' feature value 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.

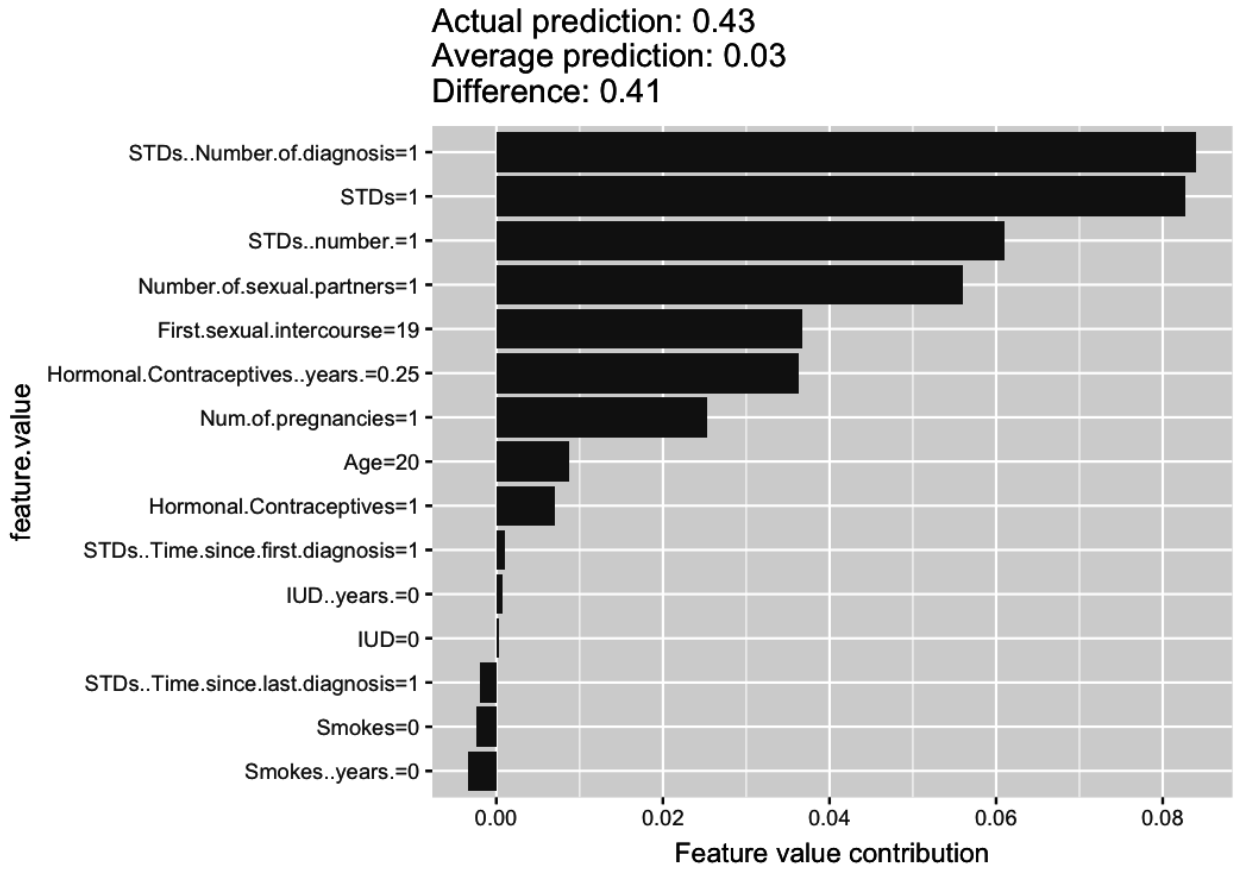
When we repeat the Shapley value for all feature values, we get the complete distribution of the prediction (minus the average) among the feature values.

Examples and Interpretation

The interpretation of the Shapley value ϕ_{ij} for feature j and instance i is: the feature value x_{ij} contributed ϕ_{ij} towards the prediction for instance i compared to the average prediction for the dataset.

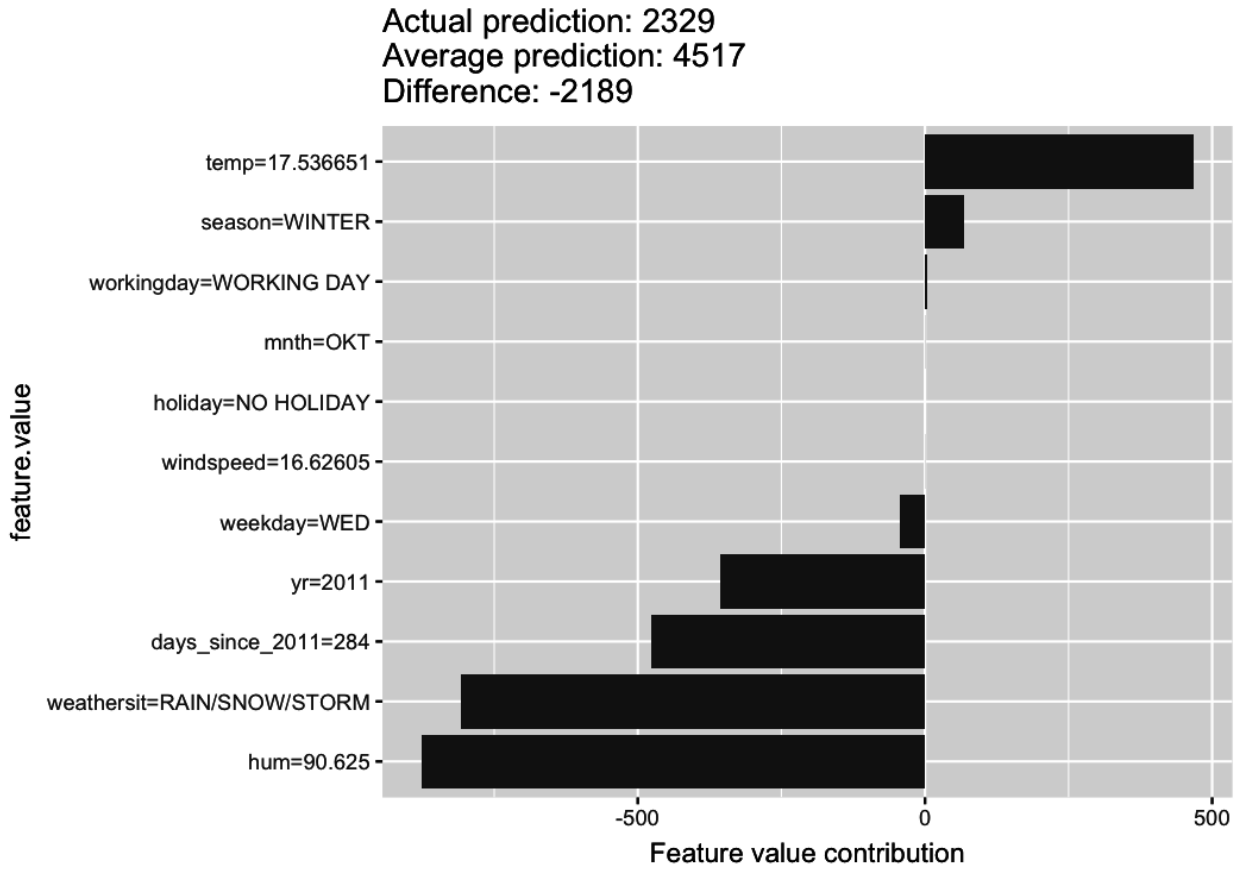
The Shapley value works for both classification (if we deal with probabilities) and regression.

We use the Shapley value to analyse the predictions of a Random Forest model predicting cervical cancer:



Feature value contributions for woman 326 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 feature value that increased the probability the most is the number of diagnosed STDs. The feature contributions sum up to the difference of 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 the weather conditions and calendric information. The explanations created for the Random Forest prediction of one specific day:



Feature value contributions for instance 285. With a predicted 2329 rented bikes, this day is -2189 below the average prediction of 4517. The feature values that had the most negative effects were the weather situation, humidity and the time trend (years since 2011). The temperature on that day had a positive effect compared to the average prediction. The feature contributions sum up to 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 towards the prediction in different coalitions. The Shapley value is NOT the difference in prediction when we would drop 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 part straight to 'Advantages and Disadvantages' if you are not interested in the technicalities.

We are interested in the effect each feature has on the prediction of a data point. In a linear model it is easy to calculate the individual effects. Here's how a linear model prediction looks like for one data instance:

$$\hat{f}(x_{i\cdot}) = \hat{f}(x_{i1}, \ldots, x_{ip}) = \beta_0 + \beta_1 x_{i1} + \ldots + \beta_p x_{ip}$$

where $x_{i\cdot}$ is the instance for which we want to compute the feature effects. Each x_{ij} is a feature value, with $j \in \{1, \ldots, p\}$. The β_j are the weights corresponding to x_{ij}.

The feature effect ϕ_{ij} of x_{ij} on the prediction $\hat{f}(x_{i\cdot})$ is:

$$\phi_{ij}(\hat{f}) = \beta_j x_{ij} - E(\beta_j X_j) = \beta_j x_{ij} - \beta_j E(X_j)$$

where $E(\beta_j X_j)$ is the mean effect estimate for feature X_j. The effect is the difference between the feature contribution to the equation minus the average contribution. Nice! Now we know how much each feature contributed towards the prediction. If we sum up all the feature effects over all features for one instance, the result is:

$$\sum_{j=1}^{p} \phi_{ij}(\hat{f}) = \sum_{j=1}^{p} (\beta_j x_{ij} - E(\beta_j X_j)) = (\beta_0 + \sum_{j=1}^{p} \beta_j x_{ij}) - (\beta_0 + \sum_{j=1}^{p} E(\beta_j X_j)) = \hat{f}(x_{i\cdot}) - E(\hat{f}(X))$$

This is the predicted value for the data point $x_{i\cdot}$ minus the average predicted value. Feature effects ϕ_{ij} can be negative.

Now, can we do the same for any type of model? It would be great to have this as a model-agnostic tool. Since we don't have the β's from a linear equation 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 effects $\phi_{ij}(\hat{f})$ for single predictions for any machine learning model $\hat{f}$.

The Shapley Value

The Shapley value is defined via a value function val over players in S.

The Shapley value of a feature value x_{ij} is it's contribution to the payed outcome, weighted and summed over all possible feature value combinations:

$$\phi_{ij}(val) = \sum_{S \subseteq \{x_{i1}, \ldots, x_{ip}\} \setminus \{x_{ij}\}} \frac{|S|!\,(p - |S| - 1)!}{p!} \left(val\left(S \cup \{x_{ij}\}\right) - val(S)\right)$$

where S is a subset of the features used in the model, $x_{i\cdot}$ is the vector feature values of instance i and p the number of features. $val_{x_i}(S)$ is the prediction for feature values in set S, marginalised over features not in S:

$$val_{x_i}(S) = \int \hat{f}(x_{i1}, \ldots, x_{ip}) d\mathbb{P}_{X_{i\cdot} \notin S} - E_X(\hat{f}(X))$$

You actually do multiple integrations, for each feature not in S. One concrete example: The machine learning model works on 4 features $\{x_{i1}, x_{i2}, x_{i3}, x_{i4}\}$ and we evaluate $\hat{f}$ for the coalition S consisting of feature values x_{i1} and x_{i3}:

$$val_{x_i}(S) = val_{x_i}(\{x_{i1}, x_{i3}\}) = \int_{\mathbb{R}} \int_{\mathbb{R}} \hat{f}(x_{i1}, X_2, x_{i3}, X_4) d\mathbb{P}_{X_2,X_4} - E_X(\hat{f}(X))$$

This looks similar to the linear model feature effects!

Don't get confused by the many uses of the word 'value': The feature value is the numerical value of a feature and instance; the Shapley value is the feature contribution towards the prediction; the value function is the payout function given a certain coalition of players (feature values).

The Shapley value is the only attribution method that satisfies the following properties (which can be seen as a definition of a fair payout):

1. **Efficiency**: $\sum_{j=1}^{p} \phi_{ij} = \hat{f}(x_i) - E_X(\hat{f}(X))$. The feature effects have to sum up to the difference of prediction for $x_{i.}$ and the average.
2. **Symmetry**: If $val(S \cup \{x_{ij}\}) = val(S \cup \{x_{ik}\})$ for all $S \subseteq \{x_{i1}, \ldots, x_{ip}\} \setminus \{x_{ij}, x_{ik}\}$, then $\phi_{ij} = \phi_{ik}$. The contribution for two features should be the same if they contribute equally to all possible coalitions.
3. **Dummy**: If $val(S \cup \{x_{ij}\}) = val(S)$ for all $S \subseteq \{x_{i1}, \ldots, x_{ip}\}$, then $\phi_{ij} = 0$. A feature which does not change the predicted value - no matter to which coalition of feature values it is added - should have a Shapley value of 0.
4. **Additivity**: For a game with combined payouts $val + val^*$ the respective Shapley values are $\phi_{ij} + \phi_{ij}^*$. The additivity axiom has no practical relevance in the context of feature effects.

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 ϕ_{ij} is the average marginal contribution of feature value x_{ij} by joining whatever features already entered the room before, i.e.

$$\phi_{ij} = \sum_{\text{All.orderings}} val(\{\text{features.before.j}\} \cup x_{ij}) - val(\{\text{features.before.j}\})$$

Estimating the Shapley value

All possible coalitions (sets) of features have to be evaluated, with and without the feature of interest for calculating the exact Shapley value for one feature value. For more than a few features, the exact solution to this problem becomes intractable, because the number of possible coalitions increases exponentially by adding more features. Strumbelj et al. (2014)[62] suggest an approximation with Monte-Carlo sampling:

$$\hat{\phi}_{ij} = \frac{1}{M} \sum_{m=1}^{M} \left(\hat{f}(x^{*+j}) - \hat{f}(x^{*-j}) \right)$$

[62]Strumbelj, Erik, Igor Kononenko, Erik Štrumbelj, and Igor Kononenko. 2014. "Explaining prediction models and individual predictions with feature contributions." Knowledge and Information Systems 41 (3): 647–65. doi:10.1007/s10115-013-0679-x.

where $\hat{f}(x^{*+j})$ is the prediction for $x_{i.}$, but with a random number of features values replaced by feature values from a random data point x, excluding the feature value for x_{ij} The x-vector x^{*-j} is almost identical to x^{*+j}, but the value x_{ij} is also taken from the sampled x. Each of those M new instances are kind of 'Frankensteins', pieced together from two instances.

Approximate Shapley Estimation Algorithm: Each feature value x_{ij}'s contribution towards the difference $\hat{f}(x_{i.}) - \mathbb{E}(\hat{f})$ for instance $x_{i.} \in X$.

- Require: Number of iterations M, instance of interest x, data X, and machine learning model $\hat{f}$
- For all $j \in \{1, \ldots, p\}$:
  - For all $m \in \{1, \ldots, M\}$: - draw random instance z from X - choose a random permutation of feature $o \in \pi(S)$ - order instance x: $x_o = (x_{o_1}, \ldots, x_{o_j}, \ldots, x_{o_p})$ - order instance z: $z_o = (z_{o_1}, \ldots, z_{o_j}, \ldots, z_{o_p})$ - construct two new instances - $x^{*+j} = (x_{o_1}, \ldots, x_{o_{j-1}}, x_{o_j}, z_{o_{j+1}}, \ldots, z_{o_p})$ - $x^{*-j} = (x_{o_1}, \ldots, x_{o_{j-1}}, z_{o_j}, z_{o_{j+1}}, \ldots, z_{o_p})$ - $\phi_{ij}^{(m)} = \hat{f}(x^{*+j}) - \hat{f}(x^{*-j})$
  - Compute the Shapley value as the average: $\phi_{ij}(x) = \frac{1}{M} \sum_{m=1}^{M} \phi_{ij}^{(m)}$

First, select an instance of interest i, a feature j and the number of samples M. For each sample, a random instance from the data is chosen and the order of the features is mixed. From this instance, two new instances are created, by combining values from the instance of interest x and the sample. The first instance x^{*+j} is the instance of interest, but where all values in order before and including feature j are replaced by feature values from the sample. The second instance x^{*-j} is similar, but has all the values in order before, but excluding feature j, replaced by features from the sample. The difference in prediction from the black box is computed:

$$\phi_{ij}^{(m)} = \hat{f}(x^{*+j}) - \hat{f}(x^{*-j})$$

All these differences are averaged and result in

$$\phi_{ij}(x) = \frac{1}{M} \sum_{m=1}^{M} \phi_{ij}^{(m)}$$

Averaging implicitly weighs samples by the probability distribution of X.

That's not the only way to compute the Shapley value: For example, Lundberg and Lee (2016)[63] propose a computation method that includes weight kernels and regularised linear regression.

Advantages

- The difference between the prediction and the average prediction is fairly distributed among the features values of the instance - the shapley efficiency property. This property sets the

[63]Lundberg, Scott, and Su-In Lee. 2016. "An unexpected unity among methods for interpreting model predictions," no. Nips: 1–6. http://arxiv.org/abs/1611.07478.

Shapley value apart from other methods like LIME. LIME does not guarantee to perfectly distribute the effects. It might make the Shapley value the only method to deliver a full explanation. In situations that demand explainability by law - like EU's "right to explanations" - the Shapley value might actually be the only compliant method. 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 with the average prediction of the whole dataset, you could compare it to a subset or even to a single datapoint.
- 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 behaviour of the machine learning model locally but there is no theory why this should work or not.
- It's mind-blowing to explain a prediction as a game played by the feature values.

Disadvantages

- The Shapley value needs a lot of computation time. In 99.9% of the real world problems the approximate solution - not the exact one - is feasible. An accurate computation of the Shapley value is potentially computational expensive, because there are 2^k possible coalitions of features and the 'absence' of a feature has to be simulated by drawing random samples, which increases the variance for the estimate ϕ_{ij}. The exponential number of the coalitions is handled by sampling coalitions and fixing the number of samples M. Decreasing M reduces computation time, but increases the variance of ϕ_{ij}. It is unclear how to choose a sensitive M.
- The Shapley value can be misinterpreted: The Shapley value ϕ_{ij} of a feature j is not the difference in predicted value after the removal of feature j. The interpretation of the Shapley value is rather: Given the current set of feature values, the total contribution of feature value x_{ij} to the difference in the actual prediction and the mean prediction is ϕ_{ij}.
- The Shapley value is the wrong explanation method if you seek sparse explanations. Humans prefer selective explanations, like LIME produces, so especially for explanations facing lay-persons, LIME might be the better choice for feature effects computation.
- The Shapley value returns a simple value per feature, and not a prediction model like LIME. This means it can't be used to make statements about changes in the prediction for changes in the input like: "If I would earn 300 € more per year, my credit score should go up by 5 points."

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.

Keywords: example-based explanations, case-based reasoning (CBR), solving by analogy

Example-based explanations are mostly model-agnostic, because they make any machine learning model more interpretable. The difference with model-agnostic methods is that the example-based explanation 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's 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's start with some examples [64]:

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 is working 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!).

[64] Aamodt, A., & Plaza, E. (1994). Case-based reasoning : Foundational issues, methodological variations, and system approaches. AI Communications, 7(1), 39â€“59.

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 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 neighbours (knn) model works explicitly with example-based predictions. For a new instance, a knn model locates the k nearest neighbours (e.g. the k=3 closest instances) and returns the average of the outcomes of those neighbours as a prediction. The prediction of a knn can be explained by returning the k neighbours, 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 interpretability methods:

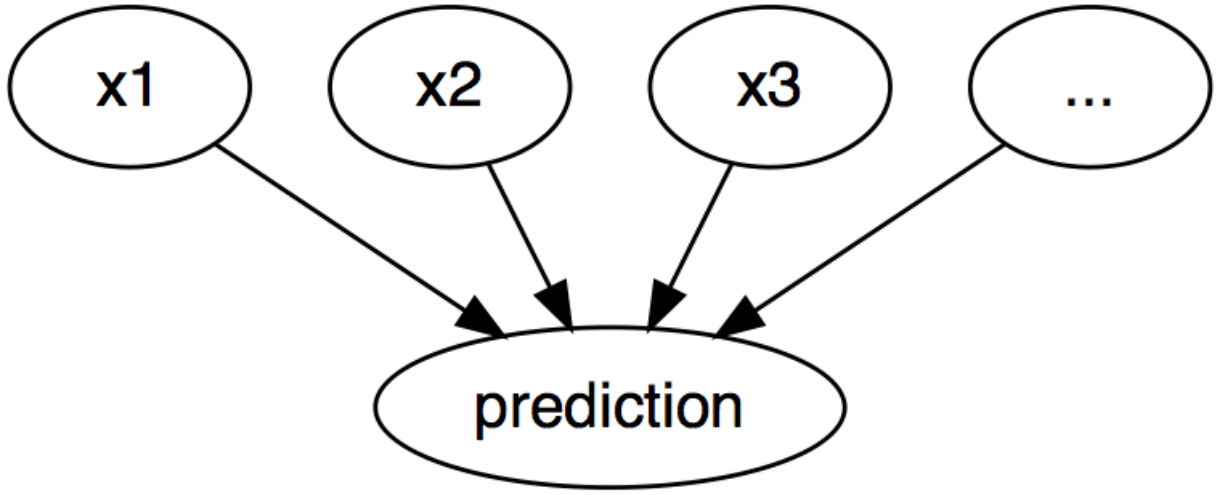
- Counterfactual instances tell us how an instance has to change to significantly change its prediction. By creating counterfactual instances, we learn about how the model makes its predictions and can explain individual predictions.
- Adversarial examples are counterfactuals used to fool machine learning models. The emphasis is on flipping the prediction and not explaining it.
- Prototypes and criticisms: Prototypes are a selection of representative instances from the data and criticisms are instances that are not well represented by those prototypes. [65]
- Influential instances are the training data points that were the most influential for the parameters of a prediction model or the predictions itself. Identifying and analysing influential instances helps to find problems with the data, debug the model and understand the model's behavior better.
- k-nearest neighbours model: An (interpretable) machine learning model based on examples.

[65] 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 for this event 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 counterfactual 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 analyse 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 don't have to be actual instances from the training data, but can be an 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 hadn't 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's 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's 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 square meters larger. Interesting, but non-actionable knowledge, because she can't 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 common class, the model could always classify an instance as the common 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 wouldn't 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 square meters 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 to 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[66]. 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

[66]Wachter, S., Mittelstadt, B., & Russell, C. (2017). Counterfactual Explanations without Opening the Black Box: Automated Decisions and the GDPR, (1), 1â€“47. https://doi.org/10.2139/ssrn.3063289

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 $\hat{f}$, 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:

$$MAD_j = median_{i \in \{1,\ldots,n\}}(|x_{i,j} - median_{l \in \{1,\ldots,n\}}(x_{l,j})|)$$

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 shouldn't 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 $|\hat{f}(x') - y'| > \epsilon$:
5. Increase λ.
6. Optimize the loss with the current counterfactual as starting point.
7. Return the counterfactual that minimizes the loss.
8. 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) [67].

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 normalised 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,

[67] Wachter, S., Mittelstadt, B., & Russell, C. (2017). Counterfactual Explanations without Opening the Black Box: Automated Decisions and the GDPR, (1), 1â€“47. https://doi.org/10.2139/ssrn.3063289

number of pregnancies and so on for women with 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 doesn't 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 don't 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's 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 to the complexity of the real world. It is also a practical challenge. Let's 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's not necessarily the fault of the method, but rather depends on the data.

The proposed method **doesn't 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)[68]. A good solution would be to use an optimizer that solves problems with a mix of continuous and discrete inputs.

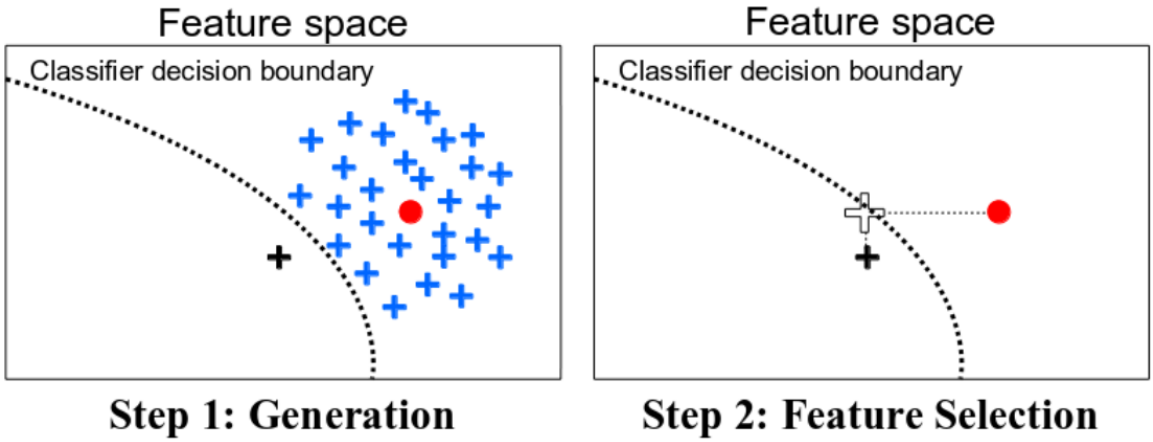
The counterfactuals method **lacks a general software implementation**. And a method is only useful if it is implemented. Fortunately, it should be easy to implement and hopefully I can remove this statement here soon.

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 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)[69]. 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 don't 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.

[68]Martens, D., & Provost, F. (2014). Explaining Data-Driven Document Classifications. MIS Quarterly, 38(1), 73â€“99. http://doi.org/10.25300/MISQ/2014/38.1.04

[69]Laugel, T., Lesot, M.-J., Marsala, C., Renard, X., & Detyniecki, M. (2017). Inverse Classification for Comparison-based Interpretability in Machine Learning. Retrieved from http://arxiv.org/abs/1712.08443



An illustration of growing spheres and selecting sparse counterfactuals by Laugel et. al (2017).

- Anchors by Ribeiro et. al 2018 [70] are the opposite of counterfactuals. Anchors answer the question: Which features are sufficient to anchor a prediction, i.e. changing the other features can't 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).

[70] Ribeiro, Marco Tulio, Sameer Singh, and Carlos Guestrin (2018). "Anchors: High-precision model-agnostic explanations." AAAI Conference on Artificial Intelligence.

Adversarial Examples

An adversarial example is an instance with small, intentional feature perturbations to 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 in which the aim is not to interpret a model but to deceive it.

Keywords: adversarial examples, adversarial machine learning, counterfactuals, evasion attacks, machine learning security

Why are we interested in adversarial examples that go beyond explaining single predictions? Aren't they 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 cars sign recognition model.

A spam detector incorrectly classifies spam as valid email. 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's take a look at some ways to create adversarial examples.

Methods and Examples

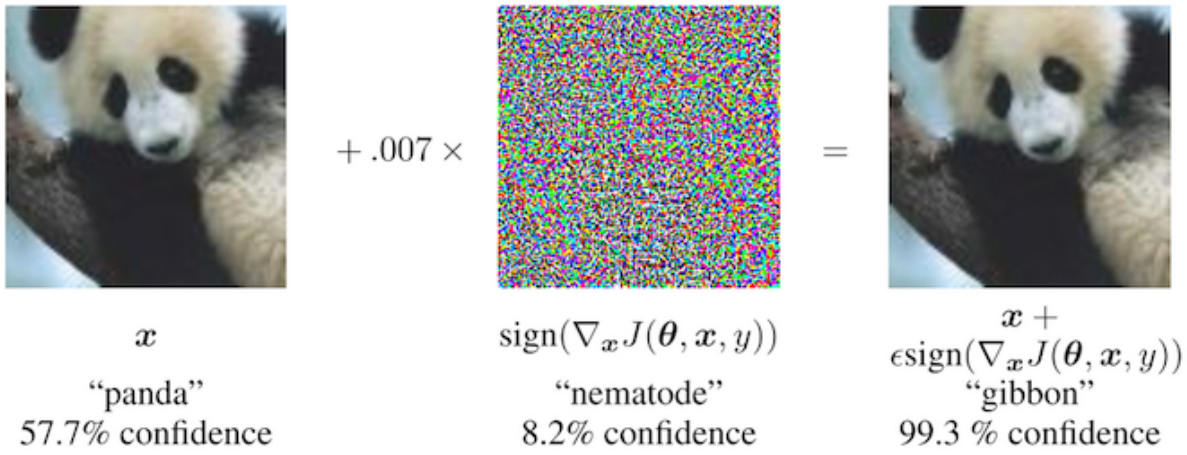
There are many techniques to create adversarial examples. Most approaches suggest minimizing the distance between the adversarial example and the instance to be manipulated, while shifting the prediction to the desired (wrong) outcome. Some methods require access to the gradients of the model, which 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 disturbed pixels which aim to deceive the model during test 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 haven't 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 for machines.

Disturbed panda: Fast gradient sign method

Goodfellow et. al (2014)[71] 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.

[71]Goodfellow, I. J., Shlens, J., & Szegedy, C. (2014). Explaining and Harnessing Adversarial Examples, 1–11. http://doi.org/10.1109/CVPR.2015.7298594

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 towards 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:

$$x' = x + \epsilon \cdot sign(\bigtriangledown_x J(\theta, x, y))$$

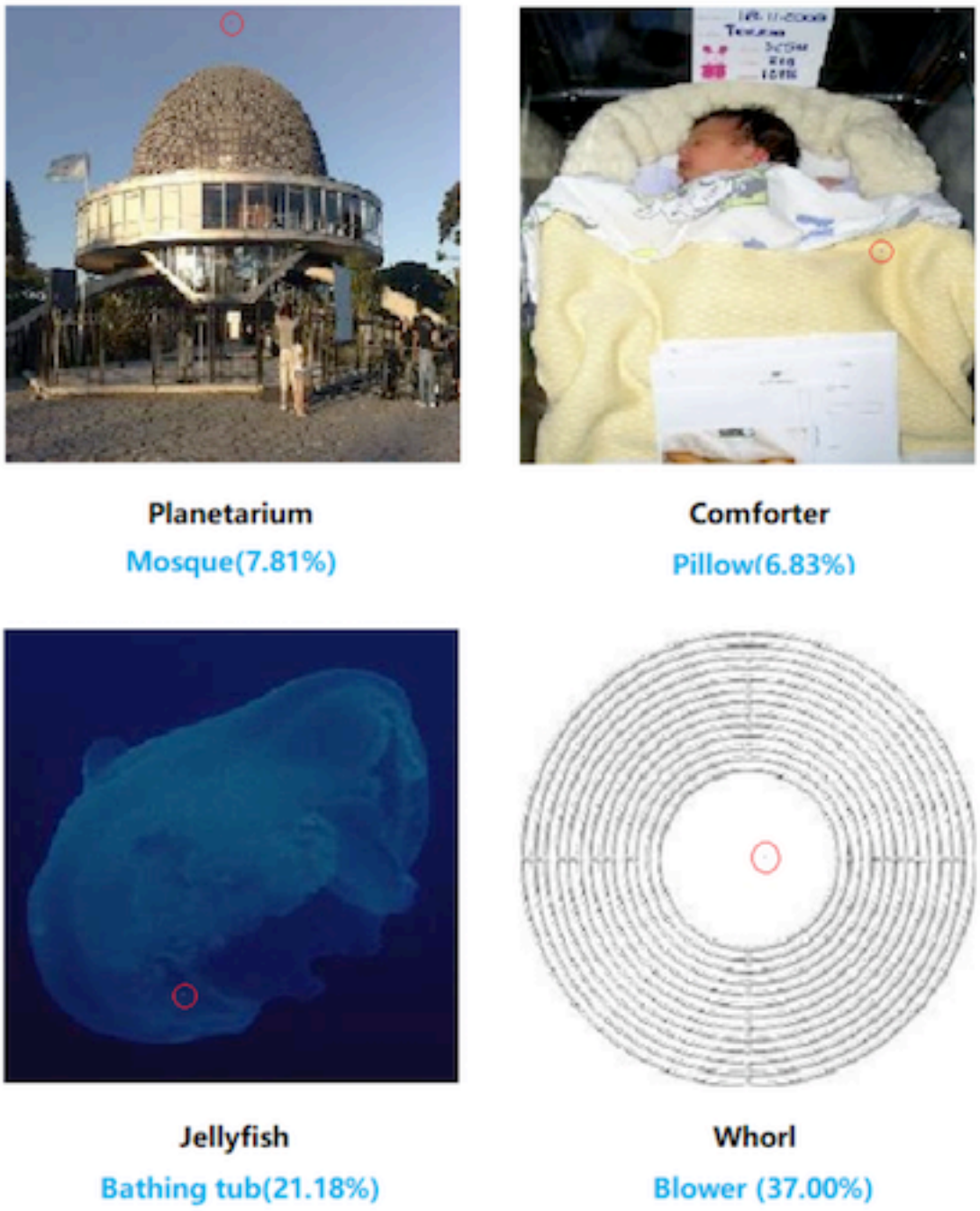
where $\bigtriangledown_x J$ is the gradient of the model's loss function with respect to the original input pixel vector x, y is the true label vector for x and θ is the model parameter vector. From the gradient vector (which is as long as the vector of the input pixels) we only need the sign: The sign 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.

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 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 (2017) [72] showed that it is actually possible to deceive image classifiers by changing a single pixel.

[72]Su, J., Vargas, D. V., & Kouichi, S. (2017). One pixel attack for fooling deep neural networks. Retrieved from http://arxiv.org/abs/1710.08864



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 (2017).

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 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) and creates a new generation of candidate solutions (children) from the parent generation using the following formula:

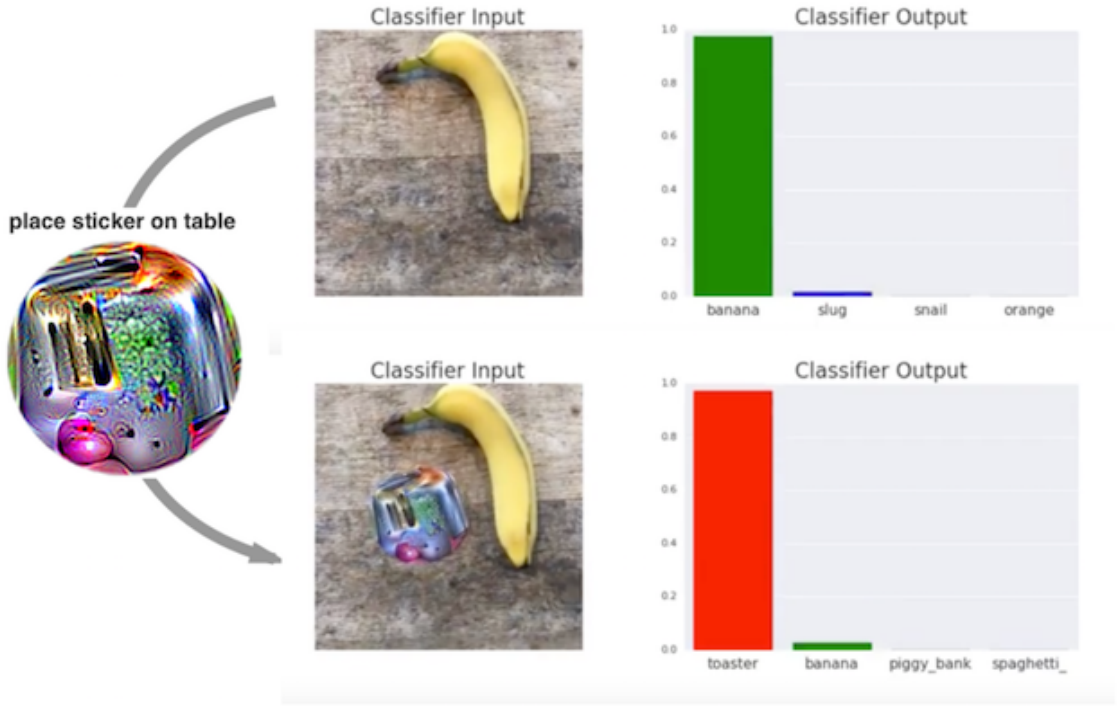
$x_i(g+1) = x_{r1}(g) + F \cdot (x_{r2}(g) + x_{r3}(g))$ where each x_i is an element of a candidate solution (either x-coordinate, y-coordinate, red, green or blue), g is the current generation, F is a scaling parameter (set to 0.5) and r1, r2 and r3 are different random numbers. Each new child candidate 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)[73] designed a printable label that can be stuck next to objects to make them look like toasters for an image classifier. Brilliant work!

[73]Brown, T. B., Mané, D., Roy, A., Abadi, M., & Gilmer, J. (2017). Adversarial Patch, (Nips). Retrieved from http://arxiv.org/abs/1712.09665



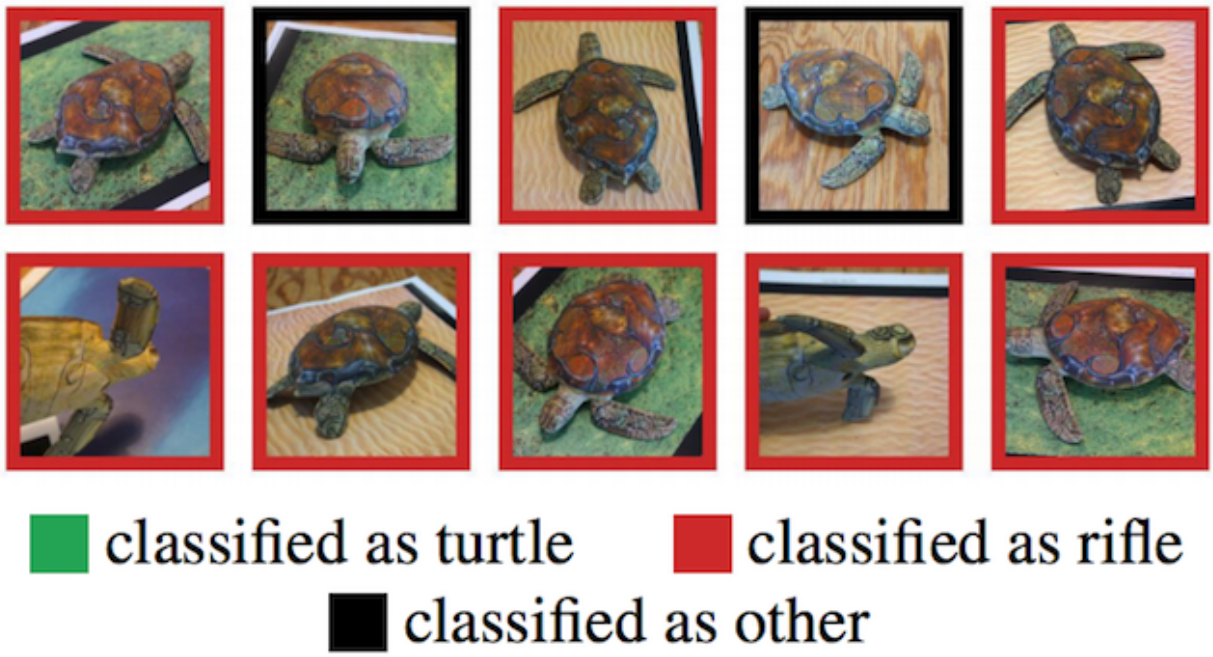
A sticker that makes a VGG16 classifier trained on ImageNet categorize an image of a banana as a toaster. Work by Brown et. al (2017).

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 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 for many 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's a good idea: Robust adversarial examples

The next method is literally adding another dimension to the toaster: Athalye et. al (2018)[74] 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 fucking rifle to the computer!

[74]Athalye, A., Engstrom, L., Ilyas, A., & Kwok, K. (2017). Synthesizing Robust Adversarial Examples. Retrieved from http://arxiv.org/abs/1707.07397



A 3D-printed turtle that is recognized as a rifle by TensorFlow's standard pre-trained InceptionV3 classifier. Work by Athalye et. al (2018)

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 (2018) propose the Expectation Over Transformation (EOT) algorithm, which is a method of 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:

$\mathbb{E}_{t\sim T}[d(t(x'), t(x))]$ 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's 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 don't 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)[75] 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.
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 can't be viewed 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.

[75] 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.

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 cyber security.

Biggio et. al (2017)[76] give a nice review of ten years of research on adversarial machine learning, on which this section is based. Cyber security is an arms-race in which attackers and defenders outwit each other time and again.

There are three golden rules in cyber security: 1) know your adversary 2) be proactive and 3) protect yourself.

The adversary depends on the application of the model. People who try to cheat other people out of their money via email 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 don't 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 analysing 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 them (more on this later). Using interpretability 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 cyber security. 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? One of the most obvious, proactive approaches 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 doesn't

[76] Biggio, B., & Roli, F. (2017). Wild Patterns: Ten Years After the Rise of Adversarial Machine Learning, 32–37. Retrieved from http://arxiv.org/abs/1712.03141

work well either is gradient masking, which constructs a model without useful gradients by using a nearest neighbor classifier instead of the original model.

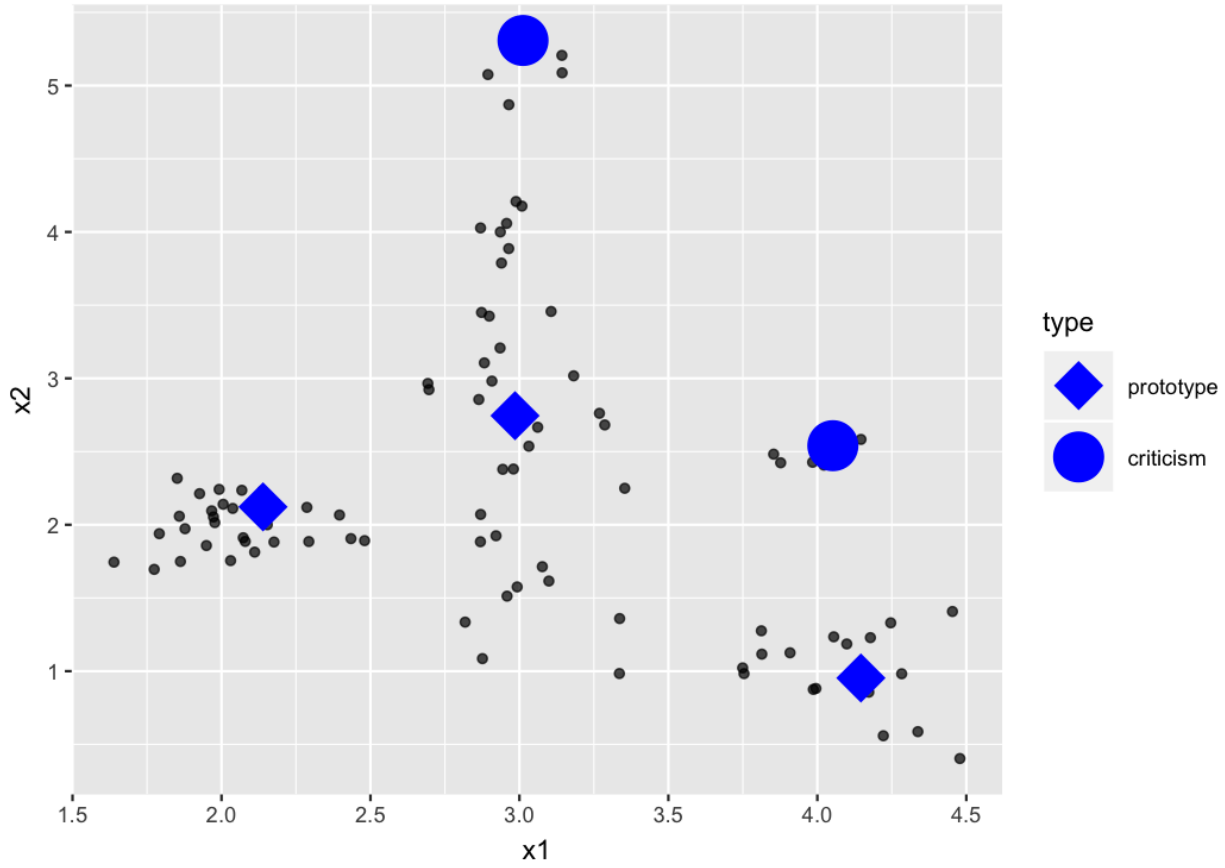
We can distinguish types of attacks by how much an attacker knows about the system. The attacker may have perfect knowledge (white box attack), meaning she knows everything about the model like the type of model, the parameters and the training data; The attacker may have partial knowledge (gray box attack), meaning she might only know the feature representation and the type of model that was used, but has no access to the training data or the parameters; The attacker may have zero knowledge (black box attack), meaning she can only query the model in a black box manner but has no access to the training data or information about the model parameters. Depending on the level of information, the attacker 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 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. Prototypes can improve the interpretability of complex data distributions, but they may not be sufficient to explain the data. They are only sufficient if the data distribution within the classes is very homogeneous. 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 don't 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 word '(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 an artificial 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 triangles are 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.

I selected the prototypes manually, which doesn't 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)[77], an approach that combines finding 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 data 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's delve deeper into the theory.

[77]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.

Theory

The MMD-critic procedure on a high-level can be summarized briefly:

1. Set the number of prototypes and criticisms you want to find.
2. Select prototypes with greedy search. Prototypes are selected so that the distribution of the prototypes is close to the data distribution.
3. Select criticisms with greedy search. Points are selected as criticisms where the distribution of prototypes differs from the distribution of the data.

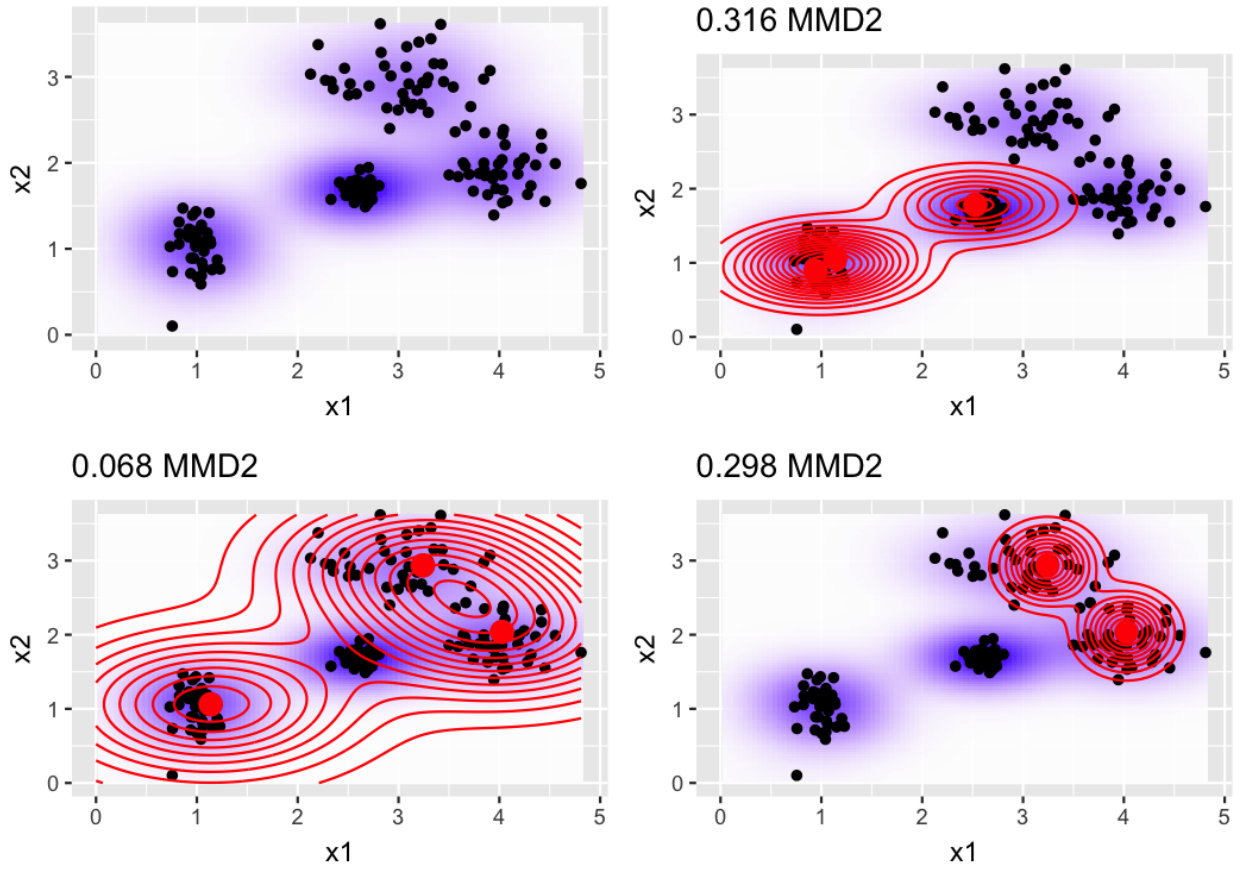
We need a couple of ingredients to find prototypes and criticisms for a dataset with MMD-critic. As the most basic ingredient, we need a **kernel function** to estimate the data densities. A kernel is a function that weights two data points according to their proximity. Based on density estimates, we 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 distribution 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's start with the **maximum mean discrepancy (MMD)**, which measures the discrepancy between two distributions. The selection of prototypes creates a density distribution of prototypes. The aim of MMD-critic is to minimize the discrepancy between the distribution of the selected prototypes and the data distribution. We want to evaluate whether the distributions are different, based on their empirical distributions which are estimated with the help of the kernel densities. 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 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 happens 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 cover 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'|| 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:

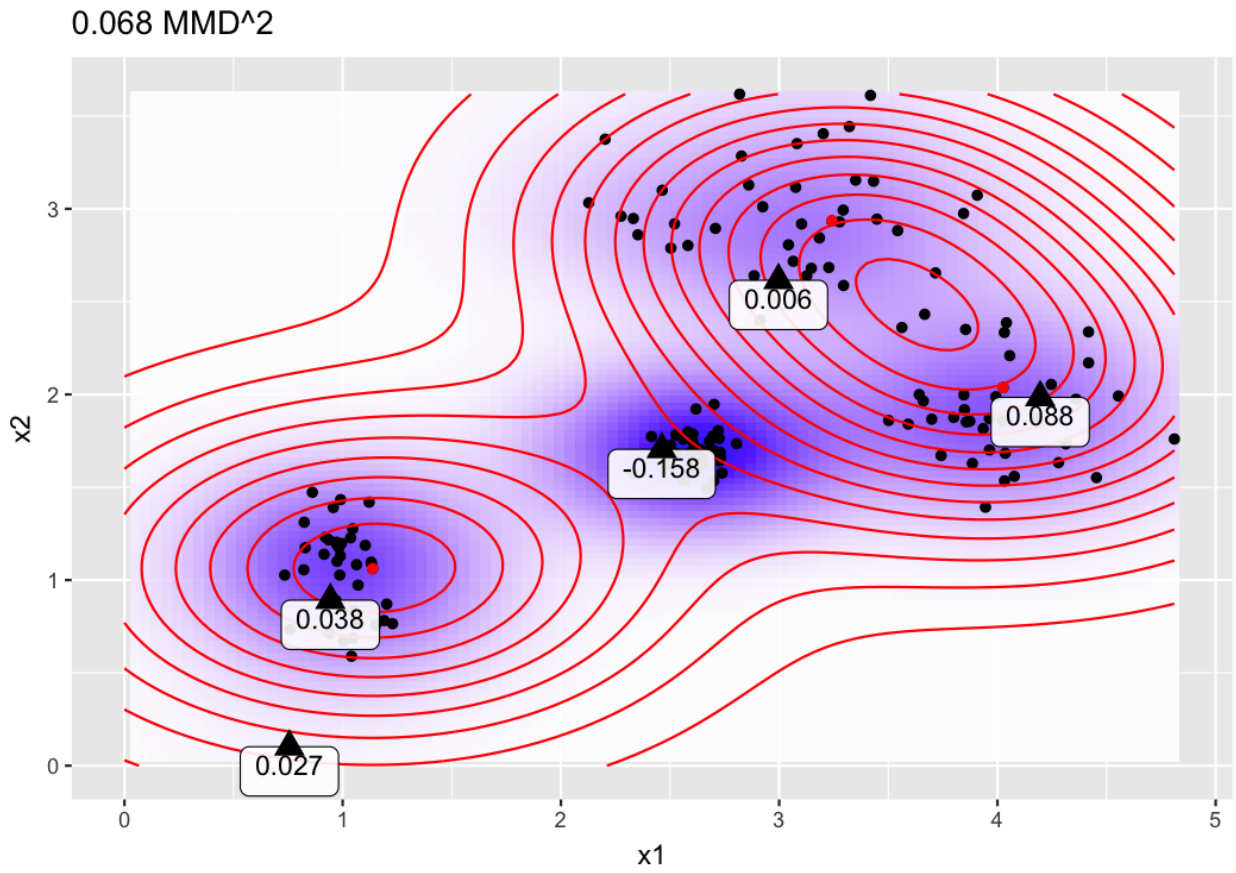
- Start with an empty list of prototypes.
- While the number of prototypes is below the chosen number m:

  - 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 more. 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 haven't selected any prototypes nearby).

To give you more intuition, let's 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 triangles. 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's 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 people with dark skin in the human class, 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' model prediction. I do not promise that MMD-critic would certainly intercept these kind of mistakes, but it's a good sanity check.

Examples

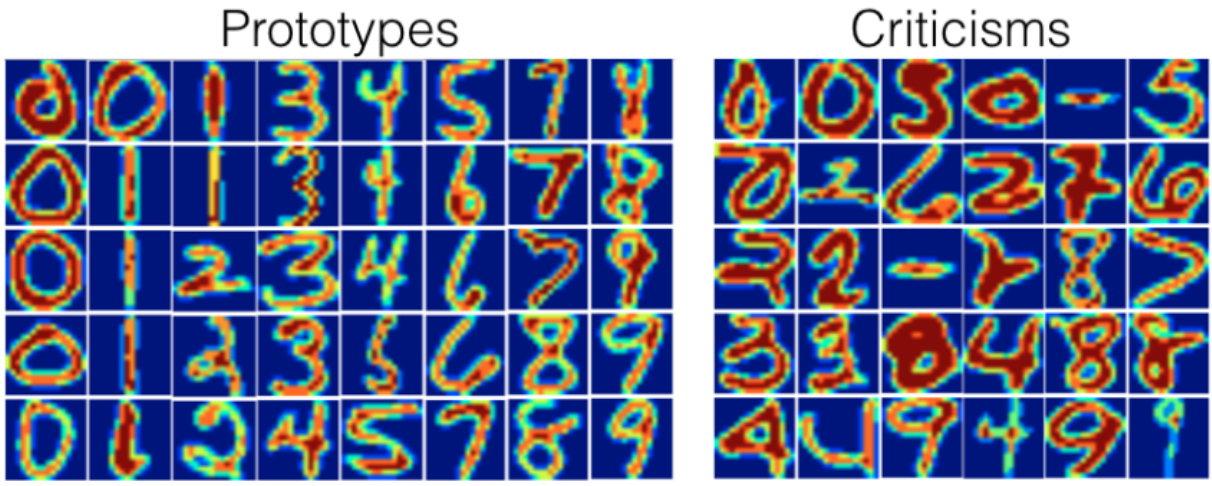
I have taken the examples from the MMD-critic paper. Both applications are based on image datasets. Each image was represented by image embeddings with 2048 dimensions. An image embedding is a vector with numbers which capture abstract attributes of an image. Embedding vectors are usually extracted from 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 application of MMD-critic features a handwritten digit dataset. Each image shows a handwritten digit.

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 the test persons images, which they had to visually match to one of two sets of images, each representing one of two classes (e.g. two dog breed). 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's also a disadvantage. How many prototypes and criticisms do we actually need? The more the better? The less the better? The number of prototypes and criticisms has to be determined by the user. One solution is to choose the number of prototypes and criticisms by seeing how much time humans have to look at them, 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[78].
- The simplest alternative to finding prototypes is k-medoids[79] by Kaufman et. al (1987).[80]

[78] https://github.com/BeenKim/MMD-critic
[79] https://en.wikipedia.org/wiki/K-medoids
[80] 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. 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.

Keywords: Influential instances, influence function, leave-one-out analysis, Cook's distance, deletion diagnostics, robust statistics

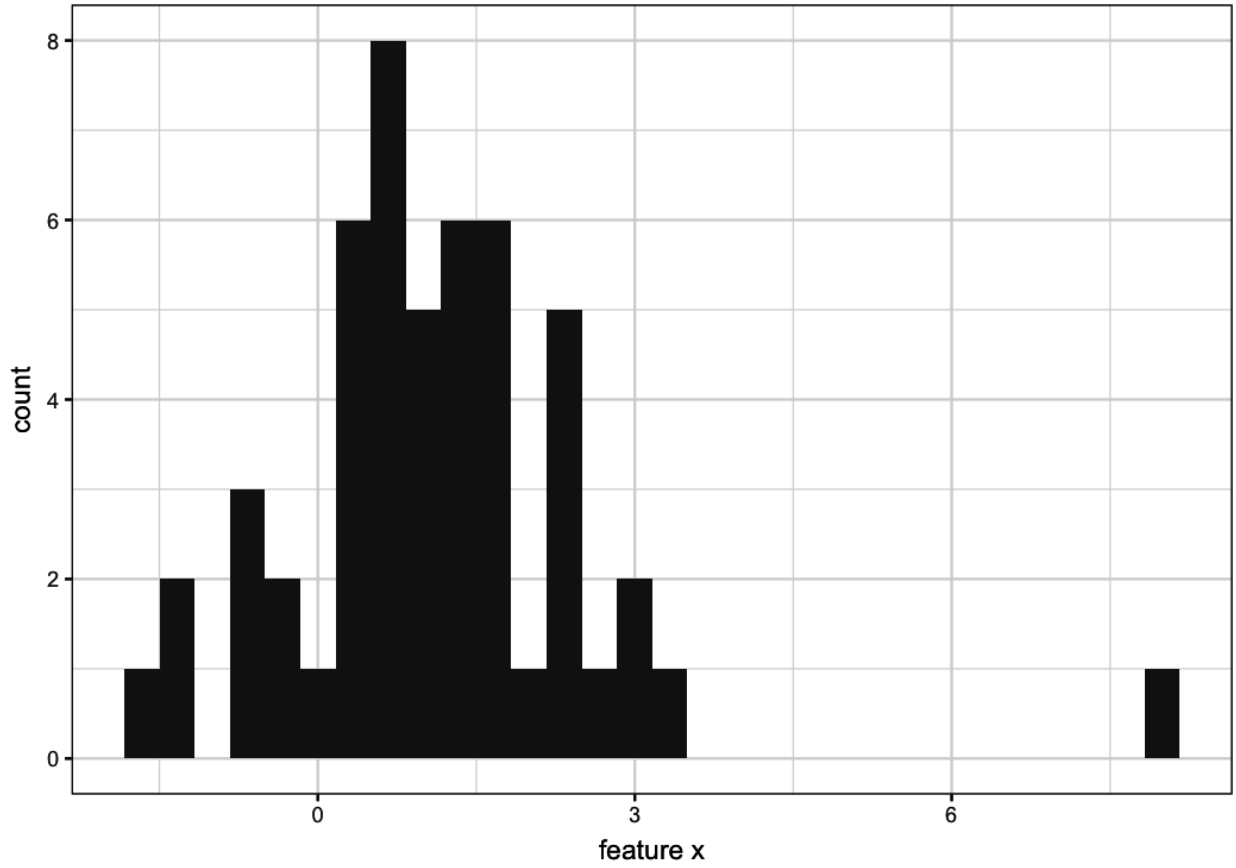
This chapter shows you two approaches for identifying influential instances, namely deletion diagnostics and influence functions. Both approaches are based on robust statistics, which provides statistical methods that are less affected by outliers or violations of model assumptions. Robust statistics 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 single persons 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.

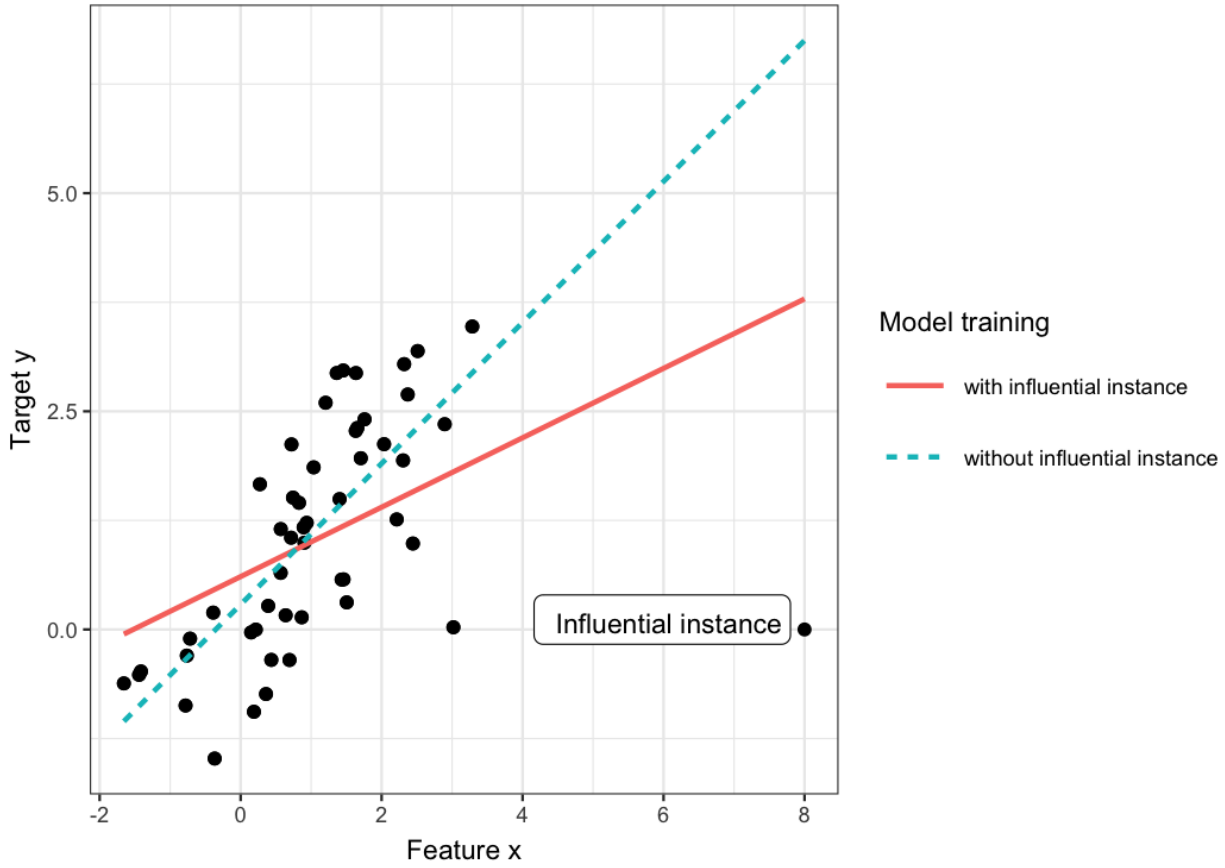


Feature x follows a Gaussian distribution with the exception of outlier x=8.

Outliers can be interesting data points (e.g. criticisms). When an outlier influences the model it is also an influential instance.

Influential Instance

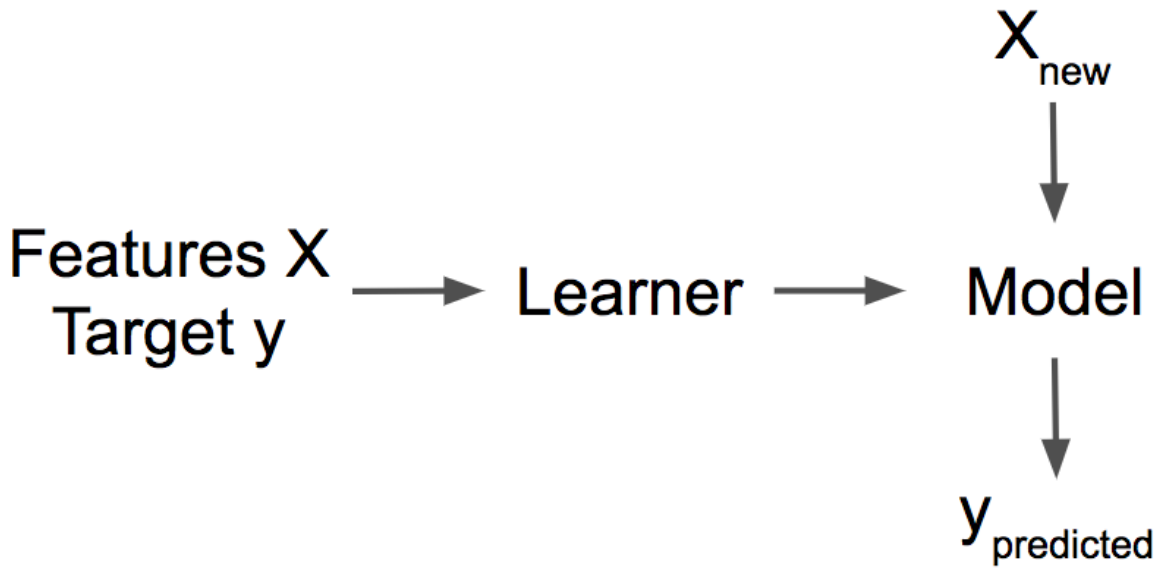
An influential instance is a data instance whose removal has a strong effect on the trained model. The more the model parameters or predictions change when the model is retrained with a particular instance removed from the training data, the more influential that instance is. Whether an instance is influential 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 featured 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 don't 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 prediction and parameters. At least that would make us investigate further.

But how exactly can we find those 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

Statistician 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[81]) measures the effect of deleting an instance on model predictions. For both measures we have to retrain the model 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 vector with the model parameters when the model is trained on all data instances, and $\beta^{(-i)}$ the model parameters 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-the 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:

[81]Cook, R. Dennis. "Detection of influential observation in linear regression." Technometrics 19.1 (1977): 15-18.

$$\text{Influence}^{(-i)} = \frac{1}{n}\sum_{j=1}^{n}\left|\hat{y}_j - \hat{y}_j^{(-i)}\right|$$

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.

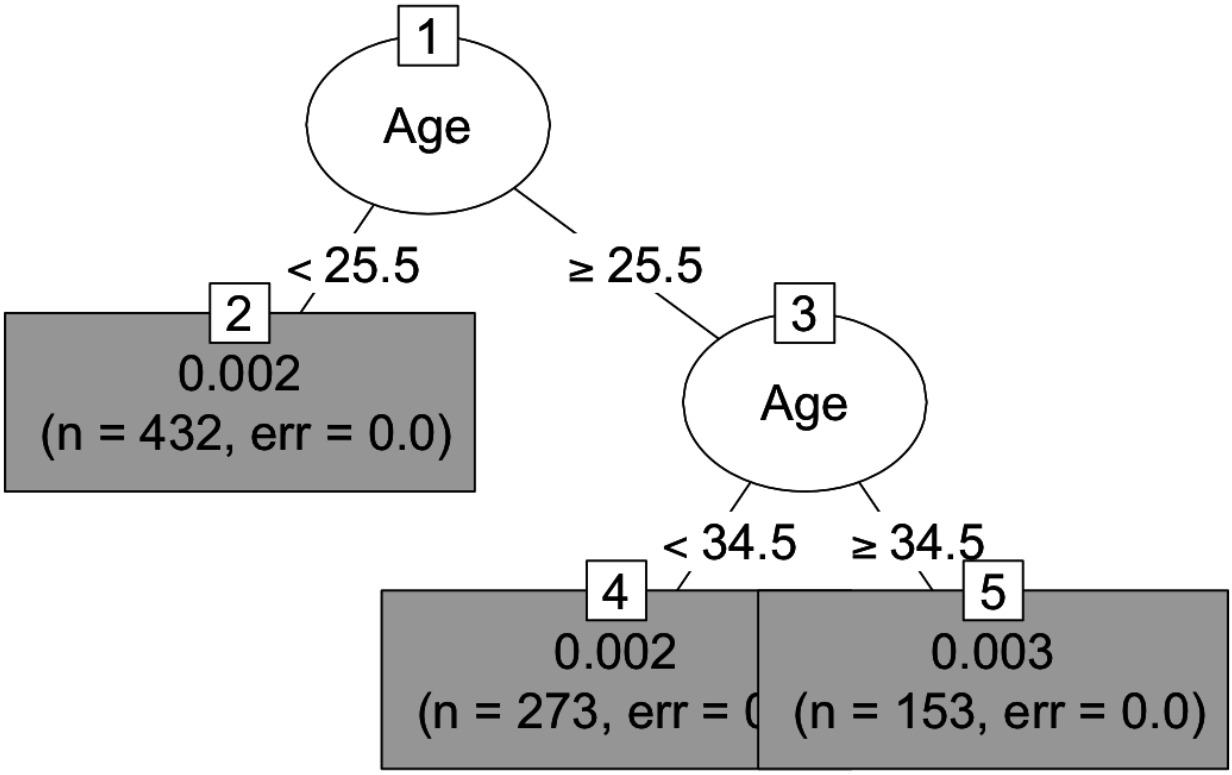
Deletion diagnostics example

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 to check 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 kinds 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

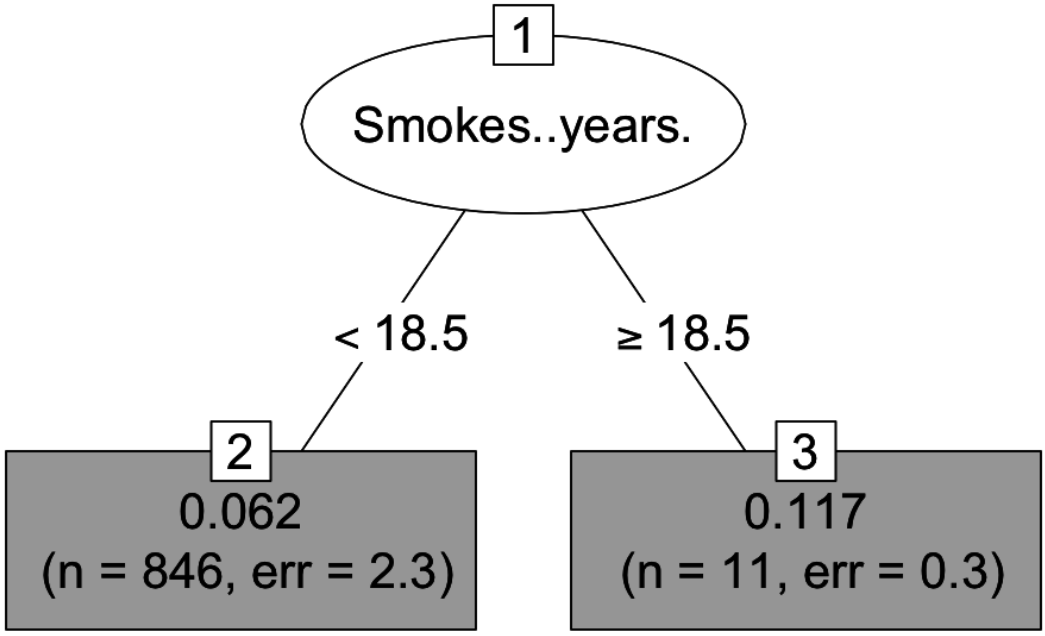
of age 35 and older were the most influential for the support vector machine. Of all the women in the dataset 153 out of 858 were older than 35. In the chapter on partial dependence plots we have seen that after 40 there is a sharp increase in the predicted probability of cancer and the feature importance 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 a higher ages. This in itself is valuable information. This means that errors in the these instances can have a strong effect on the model.



A decision tree that models the relationship between the influence of an instance and its features. The maximum depth of the tree is set to 2.

This first influence analysis asked which was 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's 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 analyse 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 couldn't 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 19 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. Maximal depth of the tree is set to 2.

Data instances of women smoked or have been smoking for 19 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 19 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's 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 doesn't 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)[82] 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 don't. 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.

[82]Koh, P. W., & Liang, P. (2017). Understanding Black-box Predictions via Influence Functions. Retrieved from http://arxiv.org/abs/1703.04730

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:

$$\hat{\theta}_{\epsilon,z} = \arg\min_{\theta\in\Theta}(1-\epsilon)\frac{1}{n}\sum_{i=1}^{n}L(z_i,\theta) + \epsilon L(z,\theta)$$

where θ is the model parameters 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 are 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, that is 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 colleagues 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.

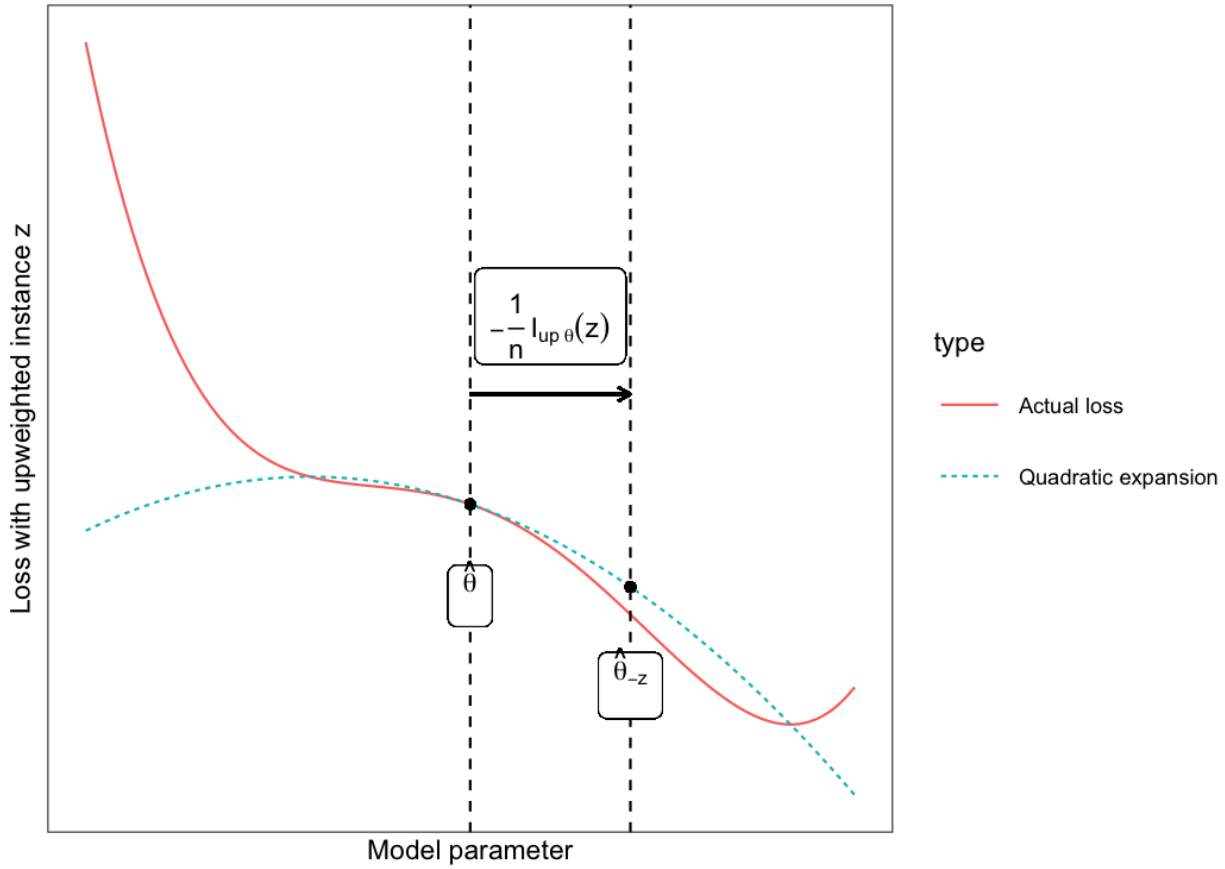
What intuition is behind this influence function formula? The formula comes from forming a quadratic expansion around the parameters $\hat{\theta}$. That means we don't actually know, or it's to complex to calculate how exactly the loss of instance z will change when it is removed/upweighted. We approximate 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's 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 don't 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 don't 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:

$$\begin{aligned} I_{up,loss}(z, z_{test}) &= \frac{dL(z_{test}, \hat{\theta}_{\epsilon,z})}{d\epsilon}\bigg|_{\epsilon=0} \\ &= \nabla_\theta L(z_{test}, \hat{\theta})^T \frac{d\hat{\theta}_{\epsilon,z}}{d\epsilon}\bigg|_{\epsilon=0} \\ &= -\nabla_\theta L(z_{test}, \hat{\theta})^T H_\theta^{-1} \nabla_\theta L(z, \hat{\theta}) \end{aligned}$$

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's 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 proportionate 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.

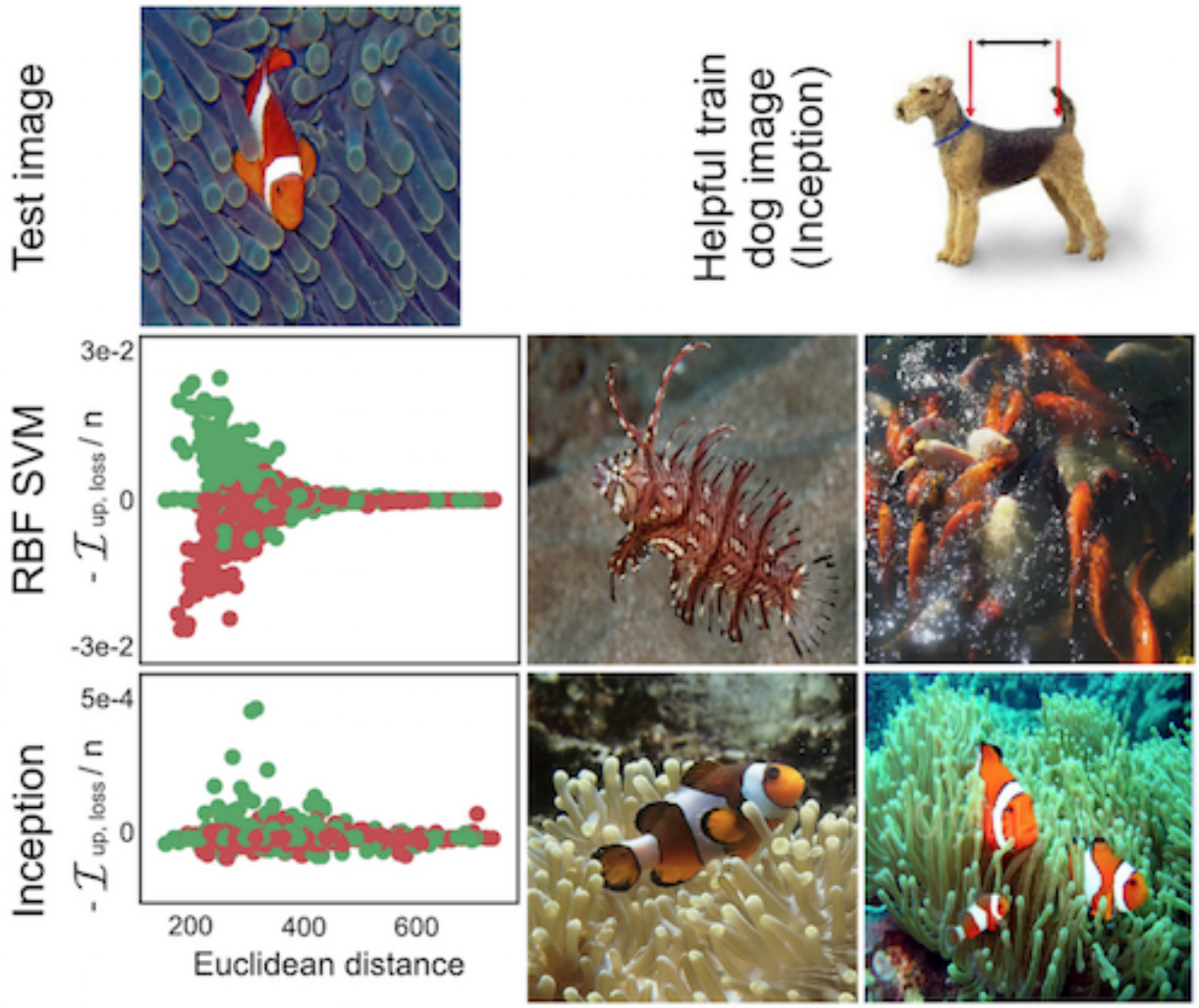
That's it with theory and intuition. The next section explains how influence functions can be applied.

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 understanding 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 analyse 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's a better choice to check the influential instances. For example, you train a model for predicting whether a patient should remain in the 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 don't 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 an 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 et. al (2018) 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 the training time, also known as poisoning attacks. If you are interested, read the paper by Koh and Liang (2017).

- For the deletion diagnostics and influence function, 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. If your model needs an hour to train and you have 10,000 instances, it takes longer than a year to sequentially calculate the influence of all instances by omitting them one after the other from the training data. But also 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 don't work for tree-based methods like random forest, 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, for example, if 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's 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 instance 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 guess 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[83] I wrote for the examples in this chapter.
- 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[84]. That's 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 can't use it directly for black box models using other frameworks, like sci-kit learn.
- Here is a great blog post for influence functions[85] 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.

[83] https://github.com/christophM/interpretable-ml-book/blob/master/manuscript/06.5-example-based-influence-fct.Rmd
[84] https://github.com/kohpangwei/influence-release
[85] 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 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, human behavior, industrial production processes and so on. The drivers of the digitization of everything are: Cheap computers/sensors/memory, 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 are taking 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 money transactions and so on. The automation of 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 even scientific discovery.

3. **Misspecification: We are not (and never will be?) 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, very happy, then the genie kills you. "I wish for world peace!" -> The genie kills all humans. We wrongly specify the goals either because we don't know all the constraints or because we can't measure them. Let's look at public limited companies as an example of imperfect goal specification: A public limited company 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 don't appreciate a company killing people to make money, poisoning rivers or to simply printing its own money. We have invented laws, regulations, penalties, compliance procedures, labor unions and more to patch up the imperfect goal specification. Another example you can live through yourself: Paperclips[86], a game in which you play a machine with the goal of producing as many paperclips as possible. WARNING: It's addictive. I don't want to spoil it too much, but let's say things get out of hand really quickly. In machine learning, the imperfections in goal specification come from imperfect data abstractions (biased population, measurement errors, ...), unconstrained loss functions, lack of knowledge of the constraints, shift in distribution between training and application data and much more.

Digitization is driving automation forward. Imperfect goal specification conflicts with automation. I claim that this conflict is mediated by interpretability methods.

The scene for our predictions is set, the crystal ball is ready, now let's take a look where the field could be going!

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 talk about interpretability.

Machine learning (or "AI") is associated 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's not possible, but because it simply 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 rather frustrating (in my experience): Often companies do not even have data in the required form and the data scientist waits inactive for months. Sometimes companies have such high expectation due to the AI hype in the media that the data scientists could never fulfill them. And often nobody knows how to integrate this new kind of people into existing structures and many more problems. That leads to my first prediction:

[86] http://www.decisionproblem.com/paperclips/index2.html

Machine learning will grow up slowly but steadily.

Digitization is advancing, and the temptation of automation 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 classical business intelligence with reporting and SQL queries instead of machine learning. But a few companies are already successfully applying machine learning, with the large Internet companies at the forefront. We need to find better ways to integrate machine learning into processes and products, train people and create machine learning tools that are easy to use. I believe that machine learning will become a lot easier to use: We can already see that machine learning becomes more accessible, for example through cloud services ("Machine Learning as a service" - just to spray a few buzz words). Once machine learning has matured - and this has baby already made its first steps - my next prediction is:

Machine learning will fuel (almost) everything.

Based on the principle "Whatever can be automated will be automated", I conclude that, whenever possible, tasks will be reformulated 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 being replaced by machine learning. Here are just a few examples:

- Automation of the sorting / deciding on / filling out 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. I am literally using this right now: A translation service (DeepL[87]) 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.

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 talked to do not use machine

[87] deepl.com

learning because they can't explain the model to others. I believe that interpretability will tackle this issue and make machine learning attractive to organisations and people that demand a degree of transparency. In addition to the misspecification of the problem, many industries require interpretability, whether for legal reasons, risk aversion or to gain insight into the underlying problem. Machine learning automates the modeling process and moves the human a bit further away from the data and the underlying problem: This increases the risk of problems with design of experiments, choice of training distribution, sampling procedure, data coding, feature engineering, etc. Interpretation tools make it easier to identify these problems

The Future of Interpretability

Let's take a look at where machine learning interpretability might be going.

The focus will be on model-agnostic interpretability tools.

It is way easier to automate interpretability if it is decoupled from the underlying machine learning model. The advantage of model-agnostic interpretability is the modularity: We can easily replace the underlying machine learning model. We can just as easily replace the interpretability method. For these reasons, model-agnostic methods will scale much better. That's why I believe that model-agnostic methods will become more dominant in the long term. But intrinsically interpretable methods will also have their place.

Machine learning will be automatic and so will be interpretability.

An already visible trend is the complete automation of model fitting: That includes the 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 on 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, fit a surrogate model and so on. Nobody stops you from automatically computing all these model interpretations. Humans will still be needed for the actual interpretation. Imagine yourself: You only upload a dataset, specify the prediction goal and, at the push of a button, the best prediction model is fitted and the program spits out all interpretations of the model. Solutions already exist and I argue that it will be sufficient for many applications to use these automated machine learning services. Today anyone can build websites without knowing HTML, CSS and Javascript but there are still web developers around. Similarly, I believe anyone 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 don't analyze data, we analyze models.

The raw data itself is always useless (I exaggerate on purpose). I don't care about the data; I do care about the knowledge distilled from 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 to the prediction (many models have built-in feature selection), the

model can automatically recognize how the relationships are represented the best and - if trained correctly - the final model is the best possible approximation to 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 the same as Pearson's correlation coefficient)
- …

So 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 restricting for the relationships the model can represent and hard to automate. Assumption-based models typically have worse predictive performance on untouched test data than black box machine learning models. This is only true for big data sets, since interpretable models with good assumptions will perform better than black box models with many parameters. The black box machine learning approach needs a lot of data to work well. Because of the digitization of everything, we will have bigger and bigger datasets and therefore the approach of machine learning becomes more attractive: We don't make assumptions, we approximate reality as closely as possible (while avoiding overfitting on 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's take a classic 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 don't 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 forecasting tasks. For this to happen the tasks must be well-defined and there must to be some processes and routine 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 that are currently being figured out 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 we will become better at creating intelligent machines.

In the end, all these predictions are speculations we have to see what the future really brings. Form your own opinion and continue learning!

Contribute

Thanks for reading this book on Interpretable Machine Learning. The book is under continuous development, it will be improved over time, and chapters will be added.

You are very welcome to contribute by, for example, writing a chapter, adding plots or fixing typos! All of the code for the book is open source and available on github.com[88]. On the Github page, you can propose fixes to the code or open issues[89] if you find some mistakes or that something is missing. The issues page is also the place for finding easy problems to get started with contributing. For bigger contributions, please send me a message: christoph.molnar.ai@gmail.com[90].

[88] https://github.com/christophM/interpretable-ml-book
[89] https://github.com/christophM/interpretable-ml-book/issues
[90] mailto:christoph.molnar.ai@gmail.com

Citation

Cite the book like this:

```
Molnar, C. (2018). Interpretable Machine Learning. Retrieved from 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 = {2018}
}
```

If you use the book as a reference, it would be great if you write me a line and tell me what for. This is of course optional and only serves to satisfy my own curiosity and perhaps to spark interesting exchanges. My mail is christoph.molnar.ai@gmail.com .

Acknowledgements

Writing this book was (and still is) a lot of fun. But it's also a lot of work and I am very happy about the support I received:

A special thanks goes to Verena Haunschmid for contributing the section about LIME explanations for images. She works in data science and I recommend following her on Twitter: @ExpectAPatronum[91]. I also want to thank all the early readers who contributed smaller fixes[92] on github!

Further, I want to thank everyone involved in creating illustrations: The cover was designed by my friend @ArbeitAmText[93]. The graphics in the Shapley Value chapter were made by Abi Aryan[94], using icons made by Freepik[95] from Flaticon[96]. The awesome images[97] in the chapter about the future of interpretability are designed by @TopeconHeroes[98]. Verena Haunschmid created the graphic in the RuleFit chapter. I often used images from the papers directly. I would like to thank all researchers who allowed me to use images from their publications.

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).

[91] https://twitter.com/ExpectAPatronum
[92] https://github.com/christophM/interpretable-ml-book/graphs/contributors
[93] https://twitter.com/ArbeitAmText
[94] https://twitter.com/GoAbiAryan
[95] http://www.freepik.com/
[96] https://www.flaticon.com/
[97] http://www.chojugiga.com/
[98] https://twitter.com/topeconheroes