

# **Interpretable Machine Learning**

**A Guide for Making Black Box Models Explainable**

Christoph Molnar

# Table of contents

<b>About the Book</b>	<b>10</b>
Summary . . . . .	10
Why I wrote the book . . . . .	10
Who this book is for . . . . .	11
What's new in the 3rd edition? . . . . .	11
About the author . . . . .	11
<b>1 Introduction</b>	<b>12</b>
1.1 Don't blindly trust model performance . . . . .	12
1.2 A young field with old roots . . . . .	13
1.3 How to read the book . . . . .	14
<b>2 Interpretability</b>	<b>15</b>
2.1 Importance of interpretability . . . . .	15
2.2 Human-friendly explanations . . . . .	20
2.2.1 What's a good explanation? . . . . .	20
<b>3 Goals of Interpretability</b>	<b>25</b>
3.1 Improving the model . . . . .	25
3.2 Justify model and predictions . . . . .	26
3.3 Discover insights . . . . .	27
<b>4 Methods Overview</b>	<b>28</b>
4.1 Interpretable models by design . . . . .	29
4.2 Post-hoc interpretability . . . . .	31
4.2.1 Model-agnostic post-hoc methods . . . . .	31
4.2.2 Model-specific post-hoc methods . . . . .	36
4.3 The lines are blurred . . . . .	37
<b>5 Data and Models</b>	<b>38</b>
5.1 Bike rentals (regression) . . . . .	38
5.1.1 Predicting bike rentals . . . . .	38
5.1.2 Feature dependence . . . . .	39
5.2 Palmer penguins (classification) . . . . .	42
5.2.1 Classifying penguin sex (male / female) . . . . .	42
5.2.2 Feature dependence . . . . .	43

<b>I Interpretable Models</b>	<b>46</b>
<b>6 Linear Regression</b>	<b>47</b>
6.1 Interpretation . . . . .	49
6.2 Example . . . . .	51
6.3 Weight and effect plot . . . . .	52
6.3.1 Explain individual predictions with effect plots . . . . .	54
6.4 Encoding categorical features . . . . .	56
6.5 Sparse linear models . . . . .	58
6.5.1 Lasso . . . . .	58
6.6 Strengths . . . . .	61
6.7 Limitations . . . . .	62
<b>7 Logistic Regression</b>	<b>64</b>
7.1 Don't use linear regression for classification . . . . .	64
7.2 Theory . . . . .	65
7.3 Interpretation . . . . .	66
7.4 Example . . . . .	69
7.5 Strengths . . . . .	70
7.6 Limitations . . . . .	70
7.7 Software . . . . .	70
<b>8 GLM, GAM and more</b>	<b>71</b>
8.1 Non-Gaussian outcomes - GLMs . . . . .	73
8.2 Interactions . . . . .	78
8.3 Nonlinear effects - GAMs . . . . .	82
8.4 Strengths . . . . .	87
8.5 Limitations . . . . .	88
8.6 Software . . . . .	88
8.7 Further extensions . . . . .	89
<b>9 Decision Tree</b>	<b>90</b>
9.1 Interpretation . . . . .	92
9.2 Example . . . . .	93
9.3 Strengths . . . . .	94
9.4 Limitations . . . . .	95
9.5 Software . . . . .	96
<b>10 Decision Rules</b>	<b>97</b>
10.1 Learn rules from a single feature (OneR) . . . . .	99
10.2 Sequential covering . . . . .	102
10.3 Bayesian Rule Lists . . . . .	106
10.4 Strengths . . . . .	111

10.5 Limitations . . . . .	112
10.6 Software and alternatives . . . . .	113
<b>11 RuleFit</b>	<b>114</b>
11.1 Interpretation and example . . . . .	115
11.2 Theory . . . . .	117
11.3 Strengths . . . . .	121
11.4 Limitations . . . . .	121
11.5 Software and alternatives . . . . .	122
<b>II Local Model-Agnostic Methods</b>	<b>123</b>
<b>12 Ceteris Paribus Plots</b>	<b>124</b>
12.1 Algorithm . . . . .	125
12.2 Examples . . . . .	125
12.3 Strengths . . . . .	129
12.4 Limitations . . . . .	130
12.5 Software and alternatives . . . . .	130
<b>13 Individual Conditional Expectation (ICE)</b>	<b>131</b>
13.1 Examples . . . . .	131
13.2 Centered ICE plot . . . . .	133
13.3 Derivative ICE plot . . . . .	134
13.4 Strengths . . . . .	135
13.5 Limitations . . . . .	135
13.6 Software and alternatives . . . . .	135
<b>14 LIME</b>	<b>136</b>
14.1 LIME for tabular data . . . . .	137
14.2 LIME for text data . . . . .	140
14.3 LIME for image data . . . . .	141
14.4 Strengths . . . . .	143
14.5 Limitations . . . . .	143
14.6 Software . . . . .	144
<b>15 Counterfactual Explanations</b>	<b>145</b>
15.1 Generating counterfactual explanations . . . . .	148
15.1.1 Method by Wachter et al. . . . .	148
15.1.2 Method by Dandl et al. . . . .	150
15.2 Example . . . . .	153
15.3 Strengths . . . . .	154
15.4 Limitations . . . . .	155
15.5 Software and alternatives . . . . .	155

<b>16 Scoped Rules (Anchors)</b>	<b>157</b>
16.1 Finding anchors . . . . .	159
16.2 Complexity and runtime . . . . .	161
16.3 Tabular data example . . . . .	162
16.4 Strengths . . . . .	165
16.5 Limitations . . . . .	165
16.6 Software and alternatives . . . . .	166
<b>17 Shapley Values</b>	<b>167</b>
17.1 General idea . . . . .	167
17.2 Examples and interpretation . . . . .	169
17.3 Shapley value theory . . . . .	173
17.3.1 Definition . . . . .	174
17.4 Estimating Shapley values . . . . .	176
17.5 Strengths . . . . .	177
17.6 Limitations . . . . .	178
17.7 Software and alternatives . . . . .	179
<b>18 SHAP</b>	<b>180</b>
18.1 SHAP theory . . . . .	180
18.2 SHAP estimation . . . . .	182
18.2.1 KernelSHAP . . . . .	183
18.2.2 TreeSHAP . . . . .	187
18.2.3 Permutation Method . . . . .	188
18.3 Example . . . . .	189
18.4 SHAP aggregation plots . . . . .	189
18.4.1 SHAP Feature Importance . . . . .	191
18.4.2 SHAP Summary Plot . . . . .	192
18.4.3 SHAP Dependence Plot . . . . .	192
18.4.4 SHAP Interaction Values . . . . .	193
18.4.5 Clustering Shapley Values . . . . .	195
18.5 Strengths . . . . .	195
18.6 Limitations . . . . .	196
18.7 Software . . . . .	197
<b>III Global Model-Agnostic Methods</b>	<b>198</b>
<b>19 Partial Dependence Plot (PDP)</b>	<b>199</b>
19.1 Definition and estimation . . . . .	199
19.2 Examples . . . . .	200
19.3 PDP-based feature importance . . . . .	204
19.4 Strengths . . . . .	205

19.5 Limitations . . . . .	205
19.6 Software and alternatives . . . . .	206
<b>20 Accumulated Local Effects (ALE) . . . . .</b>	<b>207</b>
20.1 Motivation and intuition . . . . .	207
20.2 Theory . . . . .	211
20.3 Estimation . . . . .	212
20.4 ALE versus PDP . . . . .	216
20.5 Examples . . . . .	218
20.6 Strengths . . . . .	220
20.7 Limitations . . . . .	221
20.8 Software and alternatives . . . . .	222
<b>21 Feature Interaction . . . . .</b>	<b>223</b>
21.1 What are feature interactions? . . . . .	223
21.2 Friedman's H-statistic . . . . .	224
21.3 Examples . . . . .	226
21.4 Strengths . . . . .	227
21.5 Limitations . . . . .	227
21.6 Software and alternatives . . . . .	228
<b>22 Functional Decomposition . . . . .</b>	<b>230</b>
22.1 Decomposing a function . . . . .	232
22.2 Functional ANOVA . . . . .	234
22.3 Generalized Functional ANOVA for dependent features . . . . .	236
22.4 Accumulated Local Effects . . . . .	237
22.5 Decomposing tree ensembles . . . . .	238
22.6 Statistical regression models . . . . .	238
22.7 Strengths . . . . .	239
22.8 Limitations . . . . .	240
<b>23 Permutation Feature Importance . . . . .</b>	<b>241</b>
23.1 Theory . . . . .	241
23.2 Example and interpretation . . . . .	242
23.3 Conditional feature importance . . . . .	244
23.4 Group-wise PFI example . . . . .	244
23.5 Strengths . . . . .	246
23.6 Limitations . . . . .	247
23.7 Software and alternatives . . . . .	248
<b>24 Leave One Feature Out (LOFO) Importance . . . . .</b>	<b>250</b>
24.1 Examples . . . . .	251
24.2 LOFO versus PFI . . . . .	254

24.3 Strengths . . . . .	254
24.4 Limitations . . . . .	255
24.5 Software and alternatives . . . . .	256
<b>25 Surrogate Models</b>	<b>257</b>
25.1 Theory . . . . .	257
25.2 Example . . . . .	259
25.3 Strengths . . . . .	260
25.4 Limitations . . . . .	261
25.5 Software . . . . .	261
<b>26 Prototypes and Criticisms</b>	<b>262</b>
26.1 Theory . . . . .	262
26.2 Examples . . . . .	268
26.3 Strengths . . . . .	268
26.4 Limitations . . . . .	269
26.5 Software and alternatives . . . . .	270
<b>IV Neural Network Interpretation</b>	<b>271</b>
<b>27 Learned Features</b>	<b>272</b>
27.1 Feature visualization . . . . .	272
27.1.1 Feature visualization through optimization . . . . .	273
27.1.2 Connection to adversarial examples . . . . .	275
27.1.3 Text and tabular data . . . . .	276
27.2 Network Dissection . . . . .	277
27.2.1 Algorithm . . . . .	277
27.2.2 Experiments . . . . .	279
27.3 Strengths . . . . .	281
27.4 Limitations . . . . .	282
27.5 Software and further material . . . . .	283
<b>28 Saliency Maps</b>	<b>284</b>
28.1 Vanilla Gradient . . . . .	286
28.2 DeconvNet . . . . .	288
28.3 Grad-CAM . . . . .	288
28.4 Guided Grad-CAM . . . . .	290
28.5 SmoothGrad . . . . .	290
28.6 Examples . . . . .	291
28.7 Strengths . . . . .	293
28.8 Limitations . . . . .	293
28.9 Software . . . . .	294

<b>29 Detecting Concepts</b>	<b>295</b>
29.1 TCAV: Testing with Concept Activation Vectors . . . . .	295
29.1.1 Concept Activation Vector (CAV) . . . . .	296
29.1.2 Testing with CAVs (TCAV) . . . . .	296
29.2 Example . . . . .	297
29.3 Strengths . . . . .	299
29.4 Limitations . . . . .	299
29.5 Other concept-based approaches . . . . .	300
29.6 Software . . . . .	300
<b>30 Adversarial Examples</b>	<b>301</b>
30.1 Methods and examples . . . . .	301
30.2 The cybersecurity perspective . . . . .	308
<b>31 Influential Instances</b>	<b>311</b>
31.1 Deletion Diagnostics . . . . .	314
31.2 Influence Functions . . . . .	318
31.3 Strengths . . . . .	324
31.4 Limitations . . . . .	324
31.5 Software and Alternatives . . . . .	325
<b>V Beyond the Methods</b>	<b>326</b>
<b>32 Evaluation of Interpretability Methods</b>	<b>327</b>
32.1 Levels of evaluation . . . . .	327
32.2 Properties of explanations . . . . .	328
<b>33 Story Time</b>	<b>331</b>
33.0.1 Lightning Never Strikes Twice . . . . .	331
33.0.2 Trust Fall . . . . .	333
33.0.3 Fermi's Paperclips . . . . .	334
<b>34 The Future of Interpretability</b>	<b>337</b>
34.1 The future of machine learning . . . . .	338
34.2 The future of interpretability . . . . .	340
<b>35 Translations</b>	<b>343</b>
<b>36 Citing this Book</b>	<b>345</b>
<b>37 Acknowledgments</b>	<b>346</b>

<b>Appendices</b>	<b>347</b>
<b>A Machine Learning Terms</b>	<b>347</b>
<b>B Math Terms</b>	<b>350</b>
<b>C R packages used</b>	<b>351</b>
<b>D References</b>	<b>353</b>

# About the Book

## Summary

Machine learning is part of our products, processes, and research. But **computers usually don't explain their predictions**, which can cause many problems, ranging from trust issues to undetected bugs. This book is about making machine learning models and their decisions interpretable.

After exploring the concepts of interpretability, you will learn about simple, **interpretable models** such as decision trees and linear regression. The focus of the book is on model-agnostic methods for **interpreting black box models**. Some model-agnostic methods like LIME and Shapley values can be used to explain individual predictions, while other methods like permutation feature importance and accumulated local effects can be used to get insights about the more general relations between features and predictions. In addition, the book presents methods specific to deep neural networks.

All interpretation methods are explained in depth and discussed critically. How do they work? What are their strengths and weaknesses? How do you interpret them? This book will enable you to select and correctly apply the interpretation method that is most suitable for your machine learning application. Reading the book is recommended for machine learning practitioners, data scientists, statisticians, and anyone else interested in making machine learning models interpretable.

## Why I wrote the book

This book began as a side project while I was working as a statistician in clinical research. On my free day, I explored topics that interested me, and interpretable machine learning eventually caught my focus. Expecting plenty of resources on interpreting machine learning models, I was surprised to find only scattered research papers and blog posts, with no comprehensive guide. This motivated me to create the resource I wished existed – a book to deepen my own understanding and share insights with others. Today this book is a go-to resource for interpreting machine learning models. Researchers have cited the book thousands of times, students messaged me that it was essential to their theses, instructors use it in their classes, and data scientists in industry rely on it for their daily work and recommend it to their colleagues. The book has also been the foundation of my own career; first, it inspired me to do a PhD on

interpretable machine learning, and later it encouraged me to become a self-employed writer, educator, and consultant.

## Who this book is for

This book is for practitioners looking for an overview of techniques to make machine learning models more interpretable. It's also valuable for students, teachers, researchers, and anyone interested in the topic. A basic understanding of machine learning and basic university-level math will help in following the theory, but the intuitive explanations at the start of each chapter should be accessible without math knowledge.

## What's new in the 3rd edition?

The 3rd edition is both a small and a big update. It's a small update because I only added two new method chapters, namely [LOFO](#) and [Ceteris Paribus](#), and two introductory chapters: [Methods Overview](#) and [Goals of Interpretability](#)). However, I also made some bigger changes to the book that are more subtle, but which I believe improve the book. I reorganized the introduction part so that it's leaner, yet more insightful. Further, I gave the data examples more depth (e.g., studying correlations and doing more nuanced analysis), and replaced the cancer dataset with the more accessible Palmer penguin dataset. To make the book more practical, I introduced tips and warning boxes to help interpreting machine learning models the right way. A huge change in the 3rd edition was also cleaning up the book's repository and rendering the book with Quarto instead of bookdown. For you as the reader, this is only visible in the appearance of the web version of the book, but it also means the book is now much easier to maintain for me, and this will benefit the Interpretable Machine Learning book in the long run. I also fixed a lot of small things, which you can see in the [book repository's README](#), section "Changelog".

## About the author

Hi! My name is Christoph Molnar. I write and teach about machine learning, specifically topics that go beyond merely predictive performance. I studied statistics, worked for a few years as a data scientist, did my PhD on interpretable machine learning, and now I'm a writer and also [offer workshops and consulting](#). To stay up to date with my work on machine learning, you can subscribe to my newsletter [Mindful Modeler](#).

# 1 Introduction

“What’s  $2 + 5$ ?” asked teacher Wilhelm van Osten. The answer, of course, was 7. The crowd that had gathered to witness this spectacle was amazed. Because it wasn’t a human who answered, but a horse called “Clever Hans”. Clever Hans could do math – or so it seemed.  $2 + 5$ ? That’s seven taps with the horse’s foot and not one more. Quite impressive for a horse.

And indeed, Clever Hans was very clever, as later investigations showed. But its skills were not in math, but in reading social cues. It turned out that an important success factor was that the human asking Hans knew the answer. Hans relied on the tiniest changes in the human’s body language and facial expressions to stop tapping at the right time.

## 1.1 Don’t blindly trust model performance

In machine learning, we have our own versions of this clever horse: Clever Hans Predictors, a term coined by Lapuschkin et al. (2019). Some examples:

- A machine learning model trained to detect whales learned to rely on artifacts in audio files instead of basing the classification on the audio content (DeLMA and Cukierski 2013).
- An image classifier learned to use text on images instead of visual features (Lapuschkin et al. 2019).
- A wolf versus dog classifier relied on snow in the background instead of image regions that showed the animals (Ribeiro, Singh, and Guestrin 2016b).

In all these examples, the flaws didn’t lower the predictive performance on the test set. So it’s not surprising that people are wary, even for well-performing models. They want to look inside the models, to make sure they are not taking shortcuts. And there are many other reasons to make models interpretable. For example, scientists are using machine learning in their work. In a survey asking scientists for their biggest concerns about using machine learning, the top answer was “Leads to more reliance on pattern recognition without understanding” (Van Noorden and Perkel 2023). This lack of understanding is not unique to science. If you work in marketing and build a churn model, you want to predict not only who is likely to churn, but also understand why. Otherwise, how would the marketing team know what the right response is? The team could send everyone a voucher, but what if the reason for high churn probability was that they are annoyed by the many emails? Good predictive performance alone wouldn’t be enough to make full use of the churn model.

Further, many data scientists and statisticians have told me that one reason they are using “simpler models” is that they couldn’t convince their boss to use a “black box model”. But what if the complex models make better predictions? Wouldn’t it be great if you could have both good performance **and** interpretability?

To solve trust issues, to provide insights into the models, and to better debug the models, you are reading the right book. Interpretable Machine Learning offers the tools to extract insights from the model.

## 1.2 A young field with old roots

Linear regression models were already used at the beginning of the 19th century. (Legendre 1806; Gauss 1877). Statistical modeling grew around that linear regression model, and today we have more options like generalized additive models and LASSO, to name some popular model classes. In classic statistics, we typically model distributions and rely on further assumptions that allow us to make conclusions about the world. To do that, interpretability is key. For example, if you model the effect of drinking alcohol on risk for cardiovascular problems, statisticians need to be able to extract that insight from the model. This is typically done by keeping the model interpretable and having a coefficient that can be interpreted as the effect of a feature on the outcome.

Machine learning has a different modeling approach. It’s more task-driven and prediction-focused, and the emphasis is on algorithms rather than distributions. Typically, machine learning produces more complex models. Foundational work in machine learning began in the mid-20th century, while later developments expanded the field further in the later half of the century. However, neural networks go back to the 1960s (Schmidhuber 2015), and rule-based machine learning, which is part of interpretable machine learning, is an active research area since the mid of the 20th century. While not the main focus, interpretability has always been a concern in machine learning, and researchers suggested ways to improve interpretability: An example would be the random forest (Breiman 2001) which already came with built-in feature importance measure.

Interpretable Machine Learning, or Explainable AI, has really exploded as a field around 2015 (Molnar, Casalicchio, and Bischl 2020a). Especially the subfield of model-agnostic interpretability, which offers methods that work for any model, gained a lot of attention. New methods for the interpretation of machine learning models are still being published at breakneck speed. To keep up with everything that is published would be madness and simply impossible. That’s why you will not find the most novel and fancy methods in this book, but established methods and basic concepts of machine learning interpretability. These basics prepare you for making machine learning models interpretable. Internalizing the basic concepts also empowers you to better understand and evaluate any new paper on interpretability published on the pre-print server arxiv.org in the last 5 minutes since you began reading this book (I might be exaggerating the publication rate).

## 1.3 How to read the book

You don't have to read the book cover to cover, since Interpretable Machine Learning is more of a reference book with most chapters describing one method. If you are new to interpretability, I would only recommend reading the chapters on [Interpretability](#), [Goals](#), and [Methods Overview](#) first to understand what interpretability is all about and to have a "map" where you can place each method.

The book is organized into the following parts:

- The introductory chapters, including interpretability definitions and methods overview
- Interpretable models
- Local model-agnostic methods
- Global model-agnostic methods
- Methods for neural networks
- Outlook
- Machine learning terminology

Each method chapter follows a similar structure: The first paragraph summarizes the method, followed by an intuitive explanation that doesn't rely on math. Then we look into the theory of the method to get a deeper understanding of how it works, including math and algorithms. I believe that a new method is best understood using examples. Therefore, each method is applied to real data. Some people say that statisticians are very critical people. For me, this is true because each chapter contains critical discussions about the pros and cons of the respective interpretation method. This book is not an advertisement for the methods, but it should help you decide whether a method is a good fit for your project or not. In the last section of each chapter, I listed available software implementations.

I hope you will enjoy the read!

## 2 Interpretability

This chapter introduces the concepts of interpretability. While it's difficult to define interpretability mathematically, I like the definition by Biran and Cotton (2017), which was also used by Miller (2019):

“Interpretability is the degree to which a human can understand the cause of a decision.”

Another good one is by Kim, Khanna, and Koyejo (2016):

“a method is interpretable if a user can correctly and efficiently predict the method’s results”

The more interpretable a machine learning model, the easier it is for someone to understand why certain decisions or predictions were made. A model is more interpretable than another model if its decisions are easier for a human to understand than the other model’s decisions.

Sometimes you will also see the term “explainable” in this context, as in “explainable AI”. What’s the difference between explainable and interpretable? In general, researchers in the field can’t seem to agree on a definition for either term (Flora et al. 2022). I agree with the definitions from Roscher et al. (2020):

- Interpretability is about mapping an abstract concept from the models into an understandable form.
- Explainability is a stronger term requiring interpretability and additional context.

Additionally, the term explanation is typically used for local methods, which are about “explaining” a prediction. Anyway, these terms are so fuzzy that the pragmatic approach is to see them as an umbrella term that captures the “extraction of relevant knowledge from a machine-learning model concerning relationships either contained in data or learned by the model” (Murdoch et al. 2019).

### 2.1 Importance of interpretability

If a machine learning model performs well, **why do we 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)

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

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

Closely related to learning is the human desire to **find meaning in the world**. We want to harmonize contradictions or inconsistencies between elements of our knowledge structures. "Why did my dog bite me even though it has never done so before?" a human might ask. There is a contradiction between the knowledge of the dog's past behavior and the newly made, unpleasant experience of the bite. The vet's explanation reconciles the dog owner's contradiction: "The dog was under stress and that's why it bit you." The more a machine's decision affects a person's life, the more important it is for the machine to explain its behavior. If a machine learning model rejects a loan application, this may be completely unexpected for the applicants. They can only reconcile this inconsistency between expectation and reality with some kind of explanation. The explanations don't actually have to fully explain the situation but should address a main cause. Another example is algorithmic product recommendation. Personally, I always think about why certain products or movies have been algorithmically recommended to me. Often it's clear: Advertising follows me on the Internet because I recently

bought a washing machine, and I know that for the next days I'll be followed by advertisements for washing machines. Yes, it makes sense to suggest gloves if I already have a winter hat in my shopping cart. The algorithm recommends this particular movie because users who liked other movies I liked also enjoyed the recommended movie. Increasingly, Internet companies are adding explanations to their recommendations. A good example is product recommendations, which are based on frequently purchased product combinations, as illustrated in Figure 2.1.

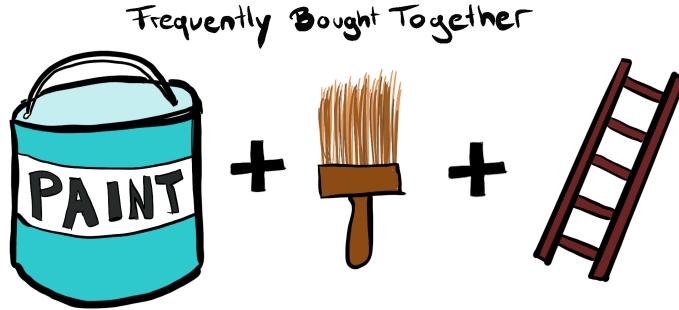


Figure 2.1: Illustration of recommended products that are frequently bought together.

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

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

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

machine learning model was optimized for.

The process of integrating machines and algorithms into our daily lives requires interpretability to increase **social acceptance**. People attribute beliefs, desires, intentions, and so on to objects. In a famous experiment, Heider and Simmel (1944) showed participants videos of shapes in which 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, assigning intentions and even emotions and personality traits to the shapes. Robots are a good example, like our vacuum cleaner, which we named “Doge”. If Doge gets stuck, I think: “Doge wants to keep cleaning, but asks me for help because it got stuck.” Later, when Doge finishes cleaning and searches the home base to recharge, I think: “Doge has a desire to recharge and intends to find the home base.” I also attribute personality traits: “Doge is a bit dumb, but in a cute way.” These are my thoughts, especially when I find out that Doge has knocked over a plant while dutifully vacuuming the house. A machine or algorithm that explains its predictions will find more acceptance.

Explanations are used to **manage social interactions**. By creating a shared meaning of something, the explainer influences the actions, emotions, and beliefs of the recipient of the explanation. For a machine to interact with us, it may need to shape our emotions and beliefs. Machines have to “persuade” us so that they can achieve their intended goal. I would not fully accept my robot vacuum cleaner if it didn’t explain its behavior to some degree. The vacuum cleaner creates a shared meaning of, for example, an “accident”, such as getting stuck on the bathroom carpet ... again, by explaining that it got stuck instead of simply stopping to work without comment. Interestingly, there may be a misalignment between the goal of the explaining machine (create trust) and the goal of the recipient (understand the prediction or behavior). Perhaps the full explanation for why Doge got stuck could be that the battery was very low, that one of the wheels is not working properly, and that there is a bug that makes the robot go to the same spot over and over again even though there was an obstacle. These reasons (and a few more) caused the robot to get stuck, but it only explained that something was in the way, and that was enough for me to trust its behavior and get a shared meaning of that accident. By the way, Doge got stuck in the bathroom again. Proof in Figure 2.2. We have to remove the carpets every time before we let Doge vacuum.

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



Figure 2.2: Doge, our vacuum cleaner, got stuck. As an explanation for the accident, Doge told us that it needs to be on an even surface.

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

- Fairness: Ensuring that predictions are unbiased and don't implicitly or explicitly discriminate against underrepresented groups.
- Privacy: Ensuring that sensitive information in the data is protected.
- Reliability or Robustness: Ensuring that small changes in the input don't lead to large changes in the prediction.
- Causality: Ensure that only causal relationships are picked up.
- Trust: It's easier for humans to trust a system that explains its decisions compared to a black box.

### Sometimes we don't need interpretability.

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

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

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

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

## 2.2 Human-friendly explanations

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

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

### 2.2.1 What's a good 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 have we not been contacted by alien life yet?

The first two questions can be answered with an “everyday” explanation, while the third one comes from the category “More general scientific phenomena and philosophical questions.” We focus on the “everyday”-type explanations because those are relevant to interpretable machine learning. In the following, the term “explanation” refers to the social and cognitive process of explaining, but also to the product of these processes. The explainer can be a human being or a machine. This section further condenses Miller’s summary on “good” explanations and adds concrete implications for interpretable machine learning.

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

**Explanations are selected.** People don’t expect explanations that cover the actual and

complete list of causes of an event. We are used to selecting one or two causes from a variety of possible causes as THE explanation. As proof, turn on the TV news: “The decline in stock prices is blamed on a growing backlash against the company’s product due to problems with the latest software update.” “Tsubasa and his team lost the match because of a weak defense: they gave their opponents too much room to play out their strategy.” “The increasing distrust of established institutions and our government are the main factors that have reduced voter turnout.” For machine learning models, it’s advantageous if a good prediction can be made from different features. Ensemble methods that combine multiple models with different features (different explanations) usually perform well because averaging over those “stories” makes the predictions more robust and accurate. But it also means that there is more than one selective explanation why a certain prediction was made. **What it means for interpretable machine learning:** Make the explanation short, give only 1 to 3 reasons, even if the world is more complex.

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

**Explanations focus on the abnormal.** People focus more on abnormal causes to explain events (Kahneman and Tversky 1982). These are causes that had a small probability but nevertheless happened. The elimination of these abnormal causes would have greatly changed the outcome (counterfactual explanation). Humans consider these kinds of “abnormal” causes as good explanations. An example from Štrumbelj and Kononenko (2011) is: Assume we have a dataset of test situations between teachers and students. Students attend a course and pass the course directly after successfully giving a presentation. The teacher has the option to additionally ask the student questions to test their knowledge. Students who cannot answer these questions will fail the course. Students can have different levels of preparation, which translates into different probabilities for correctly answering the teacher’s questions (if they decide to test the student). We want to predict whether a student will pass the course and explain our prediction. The chance of passing is 100% if the teacher does not ask any additional questions; otherwise, the probability of passing depends on the student’s level of preparation and the resulting probability of answering the questions correctly. Scenario 1: The teacher usually asks the students additional questions (e.g., 95 out of 100 times). A student who did not study (10% chance to pass the question part) was not one of the lucky ones and gets additional questions that he fails to answer correctly. Why did the student fail the course? I

would say that it was the student's fault to not study. Scenario 2: The teacher rarely asks additional questions (e.g. 2 out of 100 times). For a student who has not studied for the questions, we would predict a high probability of passing the course because questions are unlikely. Of course, one of the students did not prepare for the questions, which gives him a 10% chance of passing the questions. He is unlucky and the teacher asks additional questions that the student cannot answer, and he fails the course. What's the reason for the failure? I would argue that now, the better explanation is "because the teacher tested the student." It was unlikely that the teacher would test, so the teacher behaved abnormally. **What it means for interpretable machine learning:** If one of the input features for a prediction was abnormal in any sense (like a rare category of a categorical feature) and the feature influenced the prediction, it should be included in an explanation, even if other 'normal' features have the same influence on the prediction as the abnormal one. An abnormal feature in our house price prediction example might be that a rather expensive house has two balconies. Even if some attribution method finds that the two balconies contribute as much to the price difference as the above-average house size, the good neighborhood, or the recent renovation, the abnormal feature "two balconies" might be the best explanation for why the house is so expensive.

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

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

**Good explanations are general and probable.** A cause that can explain many events is called “general” and could be considered a good explanation. Note that this contradicts the claim that abnormal causes make good explanations. As I see it, abnormal causes beat general causes. Abnormal causes are by definition rare in the given scenario. In the absence of an abnormal event, a general explanation is considered a good explanation. Also remember that people tend to misjudge probabilities of joint events. (Joe is a librarian. Is he more likely to be a shy person or to be a shy person who likes to read books?) A good example is “The house is expensive because it is big,” which is a very general, good explanation of why houses are expensive or cheap. **What it means for interpretable machine learning:** Generality can easily be measured by the feature’s support, which is the number of instances to which the explanation applies divided by the total number of instances.

# 3 Goals of Interpretability

Interpretability is not an end in itself, but a means to an end. It depends on your specific goals which interpretability approach to use, which deserves more discussion.<sup>1</sup> Inspired by Adadi and Berrada (2018), I discuss three goals of interpretability: improve the model, justify the model and predictions, and discover insights.<sup>2</sup>

## 3.1 Improving the model

### 💡 Always evaluate performance

When determining your interpretability goals, evaluate your model’s performance metrics first. This can help you identify if your current goal should be model improvement.

You can use interpretability methods to improve the model. In the [introduction](#), we talked about Clever Hans predictors, which refers to models that have learned to take “shortcuts,” like relying on non-causal features to make predictions. The tricky thing about these shortcuts is that they often don’t decrease model performance – they might actually increase it. Like relying on snow in the background to classify whether a picture shows a wolf or a dog (Ribeiro, Singh, and Guestrin 2016b) or misleadingly predicting that asthma patients admitted to the emergency room are less likely to die of pneumonia (Caruana et al. 2015).<sup>3</sup> Interpretability helps to **debug the model** by identifying when the model takes such unwanted shortcuts or makes other mistakes. Some of the bugs may be as simple as the wrong encoding of the target feature, or an error in feature engineering. Do feature effects contradict your domain knowledge? You may have switched target classes. A feature you know is important isn’t

<sup>1</sup>When researchers propose new interpretable models or interpretability methods, they rarely discuss what *specific* goals they serve. I’m not excluding myself here: For example, I did not introduce this chapter on goals until the third edition of the book.

<sup>2</sup>The paper by Adadi and Berrada (2018) additionally introduced “control”, which refers to debugging the model and finding errors, but I subsumed this under “improvement”.

<sup>3</sup>A model predicted that patients who came to the emergency room with pneumonia were less likely to die from pneumonia when they had asthma, despite asthma being known as risk factor for pneumonia. Only thanks to using an interpretable model, the researchers found out that this model had learned this relationship. But it’s an unwanted “shortcut”: Asthma patients were treated earlier and more aggressively with antibiotics, so they were in fact less likely to develop severe pneumonia. The model learned the shortcut (asthma  $\Rightarrow$  lowered risk of dying from pneumonia), because it lacked features about the later treatment.

used by the model according to your investigations? You may have made a mistake in data processing or feature engineering.

I've also used interpretability methods in machine learning competitions to identify important features so that I can **get ideas for feature engineering**. For example, I participated in a competition to predict water supply, and through feature importance, I realized that the snow in the surrounding mountains was the most important feature. So I decided to try out alternative snow data sources that might make this feature even better, and invested time in feature engineering. In the end, it helped improve the model performance.

## 3.2 Justify model and predictions

Interpretable machine learning helps justify the model and its predictions to other people or entities. It's helpful to think of the stakeholders of a machine learning system (Tomsett et al. 2018):

- **Creators** build the system and train the model.
- **Operators** interact with the system directly.
- **Executors** make decisions based on the outputs.
- **Decision subjects** are affected by the decisions.
- **Auditors** audit and investigate the system.
- **Data subjects** are people whose data the model is trained on.

These stakeholders want justification of the model and its predictions and may require very different types of justification.

A deliverable of the machine learning competition I participated in was to generate reports that explain the water supply forecasts. These reports and explanations are for hydrologists and officials who have to make decisions based on these water supply forecasts (Executors). Let's say the model predicts an unusually low water supply; it would mean that officials would have to issue drought contingency plans and adjust water allocations. Rather than blindly trusting the predictions, the decision maker may want to verify the predictions by looking at the explanations of **why** this particular prediction was made. And if the explanation conflicts with domain knowledge, the decision maker might question the forecast and investigate.

In general, contesting a prediction made by a machine learning system requires interpretability of the system. Imagine that a machine learning system rejects a loan application. For a person (decision subject) to contest that rejection, there needs to be some justification for why that prediction was made. This concept is called recourse.

Another example: If a company wants to build a medical device, it has to go through a lot of regulatory processes to show that the device is safe and efficient. If the device relies on machine learning, things get more complicated since the company also has to show that the machine learning model works as intended. Interpretable machine learning is part of the solution to

justify the model to the regulators (Auditors) who will either approve or reject the medical device.

### 3.3 Discover insights

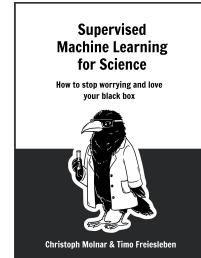
Machine learning models are not only used for making predictions; they can also be used to make decisions or to study the relationship between features and the target. In both cases, the predictions are not enough, but we want to extract additional insights from the model.

A churn prediction model predicts how likely a person is to cancel their mobile phone contract, for example. The marketing team may rely on the model to make decisions about marketing campaigns. But without knowing **why** a person is likely to churn, it's difficult to design an effective response.

More and more scientists are also applying machine learning to their research questions. For example, Zhang et al. (2019) used random forests to predict orchard almond yields based on fertilizer use. Prediction is not enough: they also used interpretability methods to extract how the different features, including fertilizer, affect the predicted yield. You need interpretability to extract the learned relationships between the features and the prediction. Otherwise, all you have is a function to make predictions.

#### 💡 How to use machine learning in science

Using machine learning in science is a much deeper philosophical question and requires more than just thinking about interpretability. That's why Timo Freiesleben and I have written a book dedicated to justifying machine learning for science. You can read it for free here: [ml-science-book.com](http://ml-science-book.com)



Interpretable machine learning is useful not only for learning about the data, but also for learning about the model. For example, if you want to learn about how convolutional neural networks work, you can use interpretability to study what concepts individual neurons react to.

What are your goals in your machine learning project, and how can interpretability help you? Your goals will determine which interpretability approaches and methods to use. In the next chapter, we will take a look at the landscape of methods and discuss how they relate to your goals.

# 4 Methods Overview

This chapter provides an overview of interpretability approaches. The goal is to give you a map so that when you dive into the individual models and methods, you can see the forest for the trees. Figure 4.1 provides a taxonomy of the different approaches.

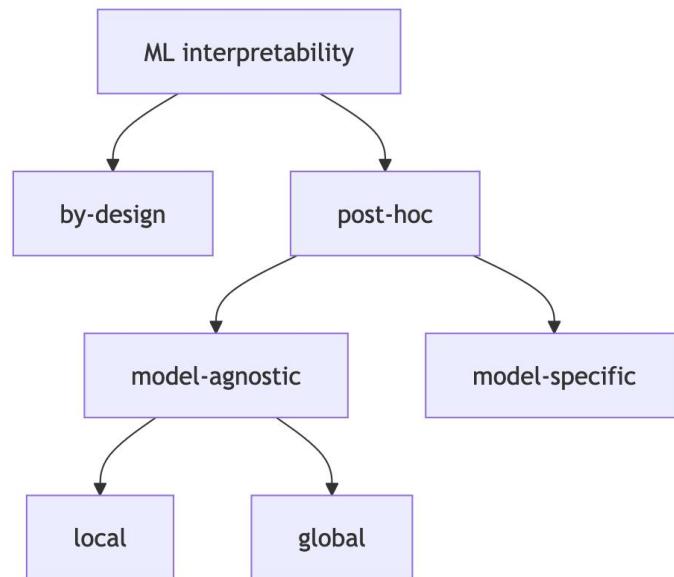


Figure 4.1: Short taxonomy of interpretability methods which reflects the structure of the book.

In general, we can distinguish between interpretability by design and post-hoc interpretability. **Interpretability by design** means that we train inherently interpretable models, such as using logistic regression instead of a random forest. **Post-hoc interpretability** means that we use an interpretability method after the model is trained. Post-hoc interpretation methods can be **model-agnostic**, such as permutation feature importance, or **model-specific**, such as analyzing the features learned by a neural network. Model-agnostic methods can be further divided into **local** methods which focus on explaining individual predictions, and **global** methods which focus on datasets. This book focuses on post-hoc model-agnostic methods but also covers basic models that are interpretable by design and model-specific methods for neural networks.

Let's look at each category of interpretability and also discuss strengths and weaknesses as they relate to your [interpretation goals](#).

## 4.1 Interpretable models by design

Interpretability by design is decided on the level of the machine learning *algorithm*. If you want a machine learning algorithm that produces interpretable models, the algorithm has to constrain the search of models to those that are interpretable. The simplest example is linear regression: When you use ordinary least squares to fit/train a linear regression model, you are using an algorithm that will produce find models that are linear in the input features. Models that are interpretable by design are also called **intrinsically** or **inherently** interpretable models, see Figure 4.2.

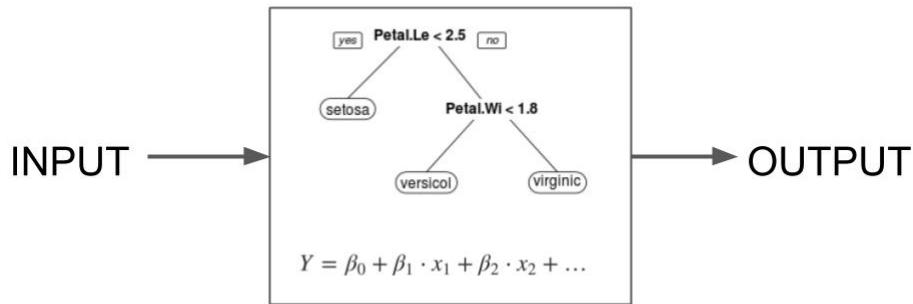


Figure 4.2: Interpretability by design means using machine learning algorithms that produce “inherently interpretable” models.

This book covers the most basic interpretability by design approaches:

- [Linear regression](#): Fit a linear model by minimizing the sum of squared errors.
- [Logistic regression](#): Extend linear regression for classification using a nonlinear transformation.
- [Linear model extensions](#): Add penalties, interactions, and nonlinear terms for more flexibility.
- [Decision trees](#): Recursively split data to create tree-based models.
- [Decision rules](#): Extract if-then rules from data.
- [RuleFit](#): Combine tree-based rules with Lasso regression to learn sparse rule-based models.

There are many more approaches to interpretable models, ranging from extensions of these basic approaches to very specialized approaches. Including all of them would be impossible,

so I have focused on the basic ones. Here are some examples of other interpretable-by-design approaches:

- Prototype-based neural networks for image classification, called ProtoViT (Ma et al. 2024). These neural networks are trained so that the image classification is a weighted sum of prototypes (special images from the training data) and sub-prototypes.
- Z. Yang et al. (2024) proposed inherently interpretable tree ensembles which are boosted trees (e.g., with XGBoost) with adjusted hyperparameters, such as low maximum tree depth, a different representation where feature effects are sorted into main effects and interactions, and pruning of effects. This approach mixes both interpretability by design and post-hoc interpretability.
- Model-based boosting is an additive modeling framework. The trained model is a weighted sum of linear effects, splines, tree stumps, and other so-called weak learners (Bühlmann and Hothorn 2007).
- Generalized additive models with automatic interaction detection (Caruana et al. 2015).

But how interpretable are intrinsically interpretable models? Approaches to interpretable models differ wildly and so do their interpretation. Let's talk about the scope of interpretability, which helps us sort the approaches:

- **The model is entirely interpretable.** Example: a small decision tree can be visualized and understood easily. Or a linear regression model with not too many coefficients. “Entirely interpretable” is a tough requirement, and again a bit fuzzy at the same time. My stance is that the term **entirely interpretable** may only be used for the simplest of models such as very sparse linear regression or very short trees, if at all.
- **Parts of the model are interpretable.** While a regression model with hundreds of features may not be “entirely interpretable”, we can still interpret the individual coefficients associated with the features. Or if you have a huge decision list, you can still inspect individual rules.
- **The model predictions are interpretable.** Some approaches allow us to interpret individual predictions. Let's say you would develop a  $k$ -nearest neighbor-like machine learning algorithm, but for images. To classify an image, take the  $k$  most similar images and return the most common class. A prediction is fully explained by showing the  $k$  similar images. Or for decision trees, a prediction is explained by returning the decision list that led to the prediction.

#### Assess interpretability scope of methods

When exploring a new interpretability approach, assess the scope of interpretability. Ask at which levels (entirely interpretable, partially interpretable, or interpretable predictions) the approach operates.

Models that are interpretable by design are usually easier to debug and improve because we get insights into their inner workings.

Interpretability by design also shines when it comes to justifying models and outputs, as they often faithfully explain how predictions were made. They also tend to make it easier to check with domain experts that the models are consistent with domain knowledge. Many data-driven fields already have established (interpretable) modeling approaches, such as logistic regression in medical research.

When it comes to discovering insights, interpretable models are a mixed bag. They make it easy to extract insights about the models themselves. But it gets trickier when it comes to data insights because of the need for a theoretical link between model structure and data. To interpret the model in place of the data, you have to assume that the model structure reflects the world – something statisticians work very hard on and need a lot of assumptions for. But what if there is a model with better predictive performance? You would have to argue why the interpretable model represents the data correctly, even though its predictive performance is inferior. In addition, there are often multiple models with similar performance but different interpretations, which makes our job more difficult. This is called the Rashomon effect. The problem with this model multiplicity is that it makes it very unclear which model to interpret.

### Rashomon

The Japanese movie Rashomon from 1950 tells four different versions of a murder story. While each version can explain the events equally well, they are incompatible with each other. This phenomenon was named the Rashomon effect.

## 4.2 Post-hoc interpretability

Post-hoc methods are applied after the model has been trained. These methods can be either model-agnostic or model-specific:

- Model-agnostic: We ignore what's *inside* the model and only analyze how the model output changes with respect to changes in the feature inputs. For example, permuting a feature and measuring how much the model error increases.
- Model-specific: We analyze parts of the model to better understand it. This can be analyzing which types of images a neuron in a neural network responds to the most, or the Gini importance in random forests.

### 4.2.1 Model-agnostic post-hoc methods

Model-agnostic methods work by the SIPA principle: **sample** from the data, perform an **intervention** on the data, get the **predictions** for the manipulated data, and **aggregate** the results (Scholbeck et al. 2020). An example is permutation feature importance: We take a

data sample, intervene on the data by permuting it, get the model predictions, and compute the model error again and compare it to the original loss (aggregation). What makes these methods model-agnostic is that they don't need to "look inside" the model, like reading out coefficients or weights, as visualized in Figure 4.3.

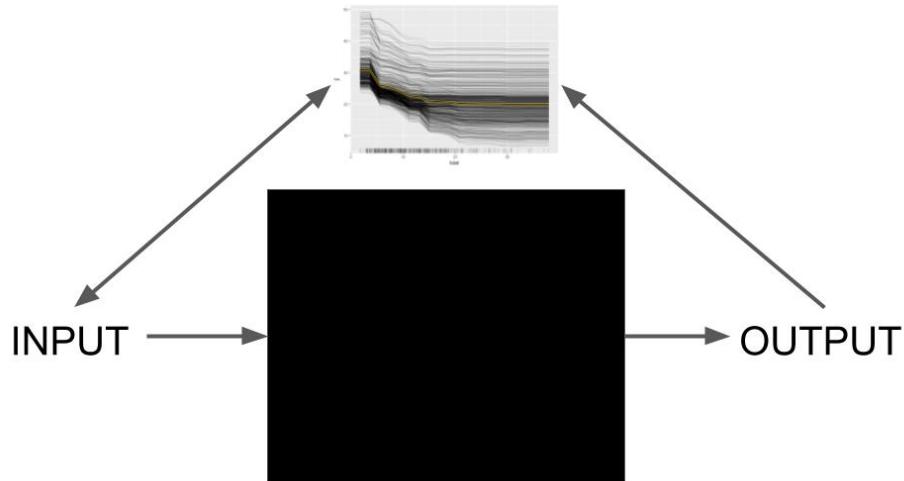


Figure 4.3: Model-agnostic interpretation methods work with inputs and outputs and ignore model internals.

Model-agnostic interpretation separates the model interpretation from the model training. Looking at this from a higher level, the modeling process gains another layer: It starts with the world, which we capture in the form of data, from which we learn a model. On top of that model, we have interpretability methods for humans to consume. See Figure 4.4. For model-agnostic methods, we have this separation, while for interpretability by design, we have model and interpretability layers merged into one.

Separating the explanations from the machine learning model (= model-agnostic interpretation methods) has some advantages (Ribeiro, Singh, and Guestrin 2016a). The biggest strength is flexibility in both the choice of model and the choice of interpretation method. For example, if you're visualizing feature effects of an XGBoost model with the partial dependence plot (PDP), you can even change the underlying model and still use the same type of interpretation. Or, if you no longer like the PDP, you can use accumulated local effects (ALE) without having to change the underlying XGBoost model. But if you are using a linear regression model and interpret the coefficients, switching to a rule-based classifier will also change the means of interpretation. Some model-agnostic methods even give you flexibility in the feature representation used to create the explanations: For example, you can create explanations based on image patches instead of pixels when explaining image classifier outputs.

Model-agnostic interpretation methods can be further divided into local and global methods.

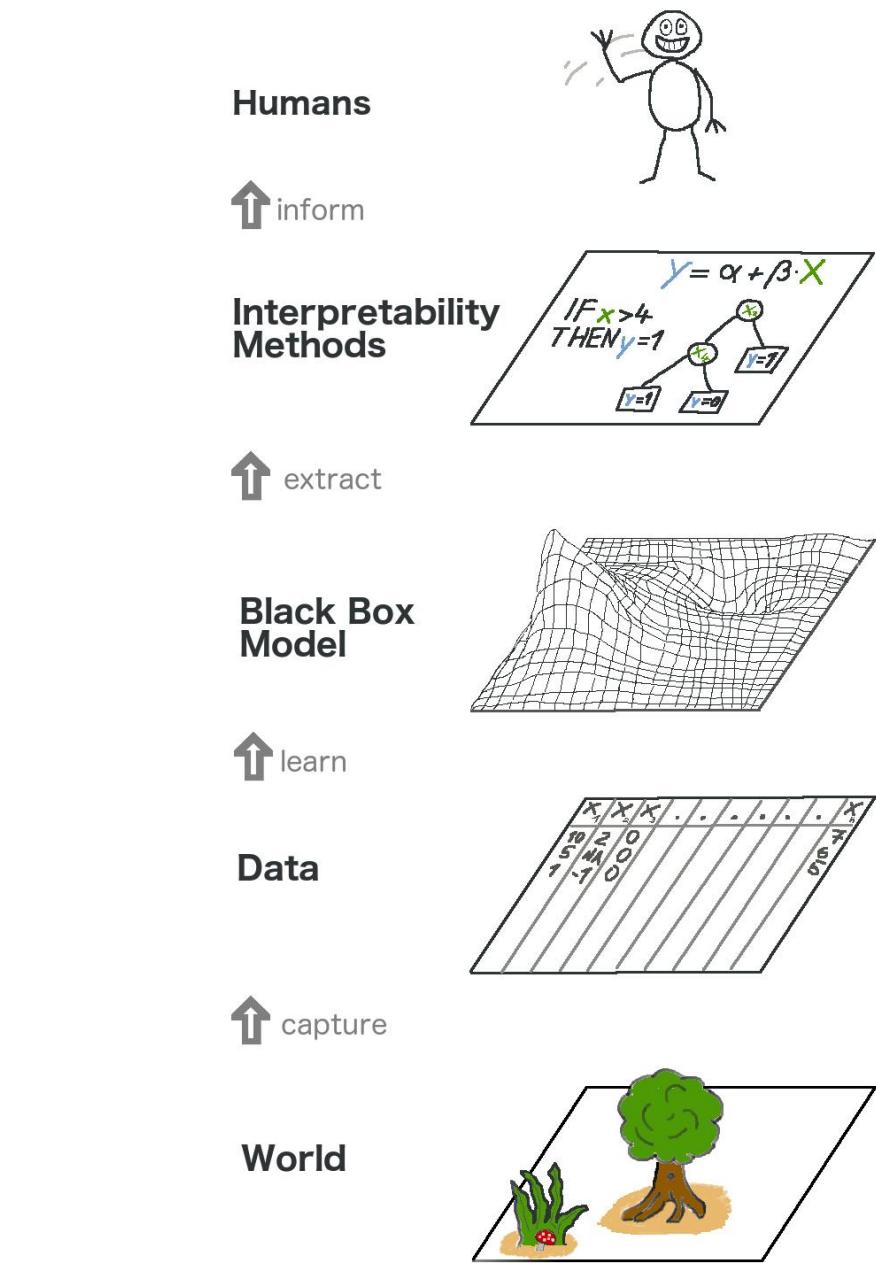


Figure 4.4: The big picture of (model-agnostic) interpretable machine learning. The real world goes through many layers before it reaches the human in the form of explanations.

[Local methods](#) aim to explain **individual predictions**, while [global methods](#) describe how features affect predictions **on average**.

#### 4.2.1.1 Local model-agnostic post hoc methods

Local interpretation methods explain individual predictions. Approaches in this category are quite diverse:

- [Ceteris paribus](#) plots show how changing a feature changes a prediction.
- [Individual conditional expectation curves](#) show how changing one feature changes the prediction of multiple data points.
- [Local surrogate models \(LIME\)](#) explain a prediction by replacing the complex model with a locally interpretable model.
- [Scoped rules \(anchors\)](#) are rules that describe which feature values “anchor” a prediction, meaning that no matter how many of the other features you change, the prediction remains fixed.
- [Counterfactual explanations](#) explain a prediction by examining which features would need to be changed to achieve a desired prediction.
- [Shapley values](#) fairly assign the prediction to individual features.
- [SHAP](#) is a computation method for Shapley values but also suggests global interpretation methods based on combinations of Shapley values across the data.

LIME and Shapley values (and SHAP) are attribution methods that explain a data point’s prediction as the sum of feature effects. Other methods, such as ceteris paribus and ICE, focus on individual features and how sensitive the prediction function is to those features. Methods such as counterfactual explanations and anchors fall somewhere in the middle, relying on a subset of the features to explain a prediction.

For model debugging, local methods provide a “zoomed in” view that can be useful for understanding edge cases or studying unusual predictions. For example, you can look at explanations for the prediction with the worst prediction error and see if it’s just a difficult data point to predict, or if maybe your model isn’t good enough, or the data point is mislabeled. Beyond that, it’s the global model-agnostic methods that are more useful for model improvements.

When it comes to using local interpretation methods to justify individual predictions, the usefulness is mixed: Methods such as ceteris paribus and counterfactual explanations can be very useful for justifying model predictions because they faithfully reflect the raw model predictions. Attribution methods like SHAP or LIME are themselves a kind of “model” (or at least more complex estimates) on top of the model being explained and therefore may not be as suitable for high-stakes justification purposes (Rudin 2019).

Local methods can be useful for data insights. Attribution methods such as Shapley values work with a reference dataset and therefore allow comparing the current prediction with

different subsets, allowing different questions to be asked. In general, the usefulness of model-agnostic interpretation for both local and global methods depends on model performance. Ceteris paribus plots and ICE are also useful for model insights.

#### 4.2.1.2 Global model-agnostic post-hoc methods

Global methods describe the average behavior of a machine learning model across a dataset. In this book, you will learn about the following model-agnostic global interpretation techniques:

- The [partial dependence plot](#) is a feature effect method.
- [Accumulated local effect plots](#) also visualize feature effects, designed also for correlated features.
- [Feature interaction \(H-statistic\)](#) quantifies the extent to which the prediction is the result of joint effects of the features.
- [Functional decomposition](#) is a central idea of interpretability and a technique for decomposing prediction functions into smaller parts.
- [Permutation feature importance](#) measures the importance of a feature as an increase in loss when the feature is permuted.
- [Leave one feature out \(LOFO\)](#) removes a feature and measures the increase in loss after retraining the model without that feature.
- [Surrogate models](#) replace the original model with a simpler model for interpretation.
- [Prototypes and criticisms](#) are representative data points of a distribution and can be used to improve interpretability.

Two broad categories within global model-agnostic methods are **feature effects** and **feature importance**. Feature effects (PDP, ALE, H-statistic, decomposition) are about showing the relationship between inputs and outputs. Feature importance (PFI, LOFO, SHAP importance, ...) is about ranking the features by importance, where importance is defined differently by each of the methods.

Since global interpretation methods describe average behavior, they are particularly useful when the modeler wants to debug a model. In particular, LOFO is related to feature selection methods and is particularly useful for model improvement.

To justify the models to stakeholders, global interpretation methods can provide some broad strokes such as which features were relevant. You can also use global methods in combination with inherently interpretable models. For example, while decision rule lists make it easy to justify individual predictions, you may also want to justify the model itself by showing which features were important overall.

Global methods are often expressed as expected values based on the distribution of the data. For example, the [partial dependence plot](#), a feature effect plot, is the expected prediction when all other features are marginalized out. This is what makes these methods so useful for understanding the general mechanisms in the data. My colleagues and I wrote papers about

the PDP and PFI, and how they can be used to infer properties about the data (Molnar, Freiesleben, et al. 2023; Freiesleben et al. 2024).

💡 Turn global into group-wise

By applying global methods to subsets of your data, you can turn global methods into “group-wise” or “regional” methods. We will see this in action in the examples in this book.

#### 4.2.2 Model-specific post-hoc methods

As the name implies, post-hoc model-specific methods are applied after model training but only work for specific machine learning models, as visualized in Figure 4.5. There are many such examples, ranging from Gini importance for random forests to computing odds ratios for logistic regression. This book focuses on post-hoc interpretation methods for neural networks.

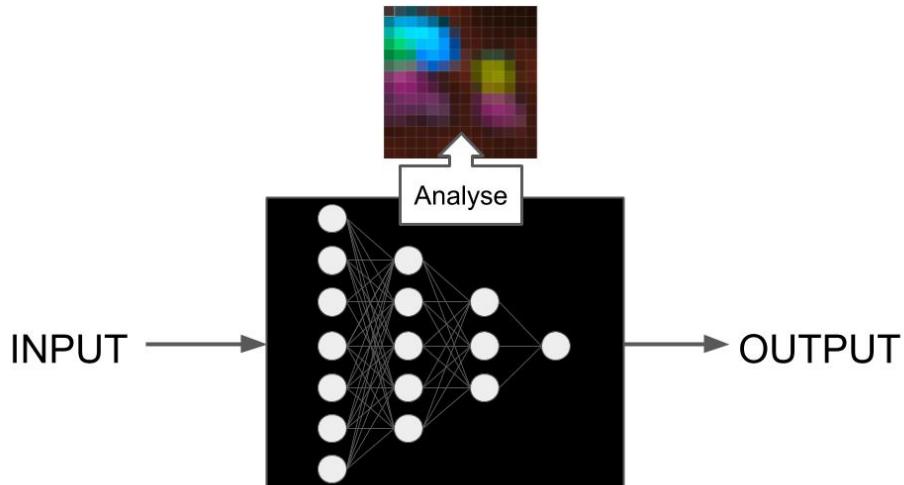


Figure 4.5: Model-specific methods make complex models more interpretable by analyzing the models.

To make predictions with a neural network, the input data is passed through many layers of multiplication with the learned weights and through non-linear transformations. A single prediction can involve millions of multiplications, depending on the architecture of the neural network. There's no chance that we humans can follow the exact mapping from data input to prediction. We would have to consider millions of weights interacting in complex ways to understand a neural network's prediction. To interpret the behavior and predictions of neural networks, we need specific interpretation methods. Neural networks are an interesting target

for interpretation because neural networks learn features and concepts in their hidden layers. Also, we can leverage their gradients for computationally efficient methods.

The neural network part covers the following techniques that answer different questions:

- **Learned Features:** What features did the neural network learn?
- **Saliency Maps:** How did each pixel contribute to a particular prediction?
- **Concepts:** Which concepts did the neural network learn?
- **Adversarial Examples:** How can we fool the neural network?
- **Influential Instances:** How influential was a training data point for a given prediction?

In general, the biggest strength of model-specific methods is the ability to learn about the models themselves. This can also help improve the model and justify it to others. When it comes to data insights, model-specific methods have similar problems as intrinsically interpretable models: They need a theoretical justification for why the model interpretation reflects the data.

### 4.3 The lines are blurred

I've presented different neat categories. But in reality, the lines between by design and post hoc are blurry. Just a few examples:

- Is logistic regression an intrinsically interpretable model? You have to post-process the coefficients to interpret the odds ratios. And if you want to interpret the model effects at the level of probabilities, you have to compute marginal effects, which can definitely be seen as a post-hoc interpretation method (which can also be applied to other models).
- Boosted tree ensembles are not considered to be interpretable. But if you set the maximum tree depth to 1, you get boosted tree stumps, which gives you something like a generalized additive model.
- To explain a linear regression prediction, you can multiply each feature value with its coefficient. These are then called effects. In addition, you can subtract from each effect the average effect from the data. If you do these things, you have computed Shapley values, which are typically considered to be model-agnostic.

The moral of the story. Interpretability is a fuzzy concept. Embrace that fuzziness, don't get too attached to one approach, but feel free to mix and match approaches.

# 5 Data and Models

Throughout the book, there are two datasets that you will encounter often. One about bikes, the other about penguins. I love them both. This chapter presents the data and models that we will interpret in this book.

## 5.1 Bike rentals (regression)

This dataset contains daily counts of rented bikes from the bike rental company [Capital Bikeshare](#) in Washington, D.C., along with weather and seasonal information. The data was kindly made openly available by Capital-Bikeshare. Fanaee-T and Gama (2014) added weather data and seasonal information. The data can be downloaded from the [UCI Machine Learning Repository](#). I did a bit of data processing and ended up with these columns:

- Count of bikes, including both casual and registered users. The count is used as the target in the regression task (`cnt`).
- The season. Either spring, summer, fall, or winter (`season`).
- Indicator of whether the day was a holiday or not (`holiday`).
- Indicator of whether the day was a workday or weekend (`workday`).
- The weather situation on that day. One of: Good, Misty, Bad (`weather`).
- Temperature in degrees Celsius (`temp`).
- Relative humidity in percent (0 to 100) (`hum`).
- Wind speed in km per hour (`windspeed`).
- Count of rented bikes two days before (`cnt_2d_bfr`).

I removed one day where the humidity was measured as 0, and the first two days due to missing count data two days before (`cnt_2d_bfr`). All in all, the processed data contains 728 days.

### 5.1.1 Predicting bike rentals

Since this example is just for showcasing the interpretability methods, I took some liberties. I pretend that the weather features are forecasts (they are not). That means our prediction task has the following shape: We predict tomorrow's number of rented bikes based on weather forecasts, seasonal information, and how many bikes were rented yesterday.

I trained all regression models using a simple holdout strategy: 2/3 of the data for training and 1/3 for testing. The machine learning algorithms were: random forest, CART decision tree, support vector machine, and linear regression. Table 5.1 shows that the support vector machine performed best, since it had the lowest root mean squared error (RMSE) and the lowest mean absolute error (MAE). The random forest was slightly worse, and the linear regression model even more so. Trailing very far behind is the decision tree, which didn't work out so well at all.

Table 5.1: Comparing bike rental model performance on test data with root mean squared error (RMSE) and mean absolute error (MAE).

Model	RMSE	MAE
SVM	852	628
Random Forest	881	672
Linear Regression	948	737
Decision Tree	1056	794

### 5.1.2 Feature dependence

For many interpretation methods, it's important to understand how the features are correlated. Therefore, let's have a look at the Pearson correlation for the numerical features. Table 5.2 shows that the only larger correlation is between count 2 days before and the temperature. But what about the categorical features? And what about non-linear correlation?

To understand the non-linear dependencies, we will do two things:

- Visualize the raw pairwise dependence (e.g., scatter plot)
- Compute the normalized mutual information (NMI) between two features.

Table 5.2: Pairwise Pearson correlation between the numerical bike rental features.

Variable 1	Variable 2	Correlation
temp	hum	0.13
temp	windspeed	-0.16
hum	windspeed	-0.25
temp	cnt_2d_bfr	0.60
hum	cnt_2d_bfr	0.06
windspeed	cnt_2d_bfr	-0.11

The normalized mutual information is a number between 0 and 1. An NMI of 0 means that the features share no information, while 1 means all variation stems from their dependence. The

NMI can be biased upwards for features with a large number of categories/bins (Mahmoudi and Jemielniak 2024). That means the more bins/categories, the less you should trust a large NMI value. That's why we also don't rely on NMI alone, but visualize the raw data and analyze Pearson correlation.

### Normalized Mutual Information

Mutual information between two categorical random variables  $X_j$  and  $X_k$  is given by:

$$MI(X_j, X_k) = \sum_{c,d} \mathbb{P}(c,d) \log \frac{\mathbb{P}(c,d)}{\mathbb{P}(c)\mathbb{P}(d)},$$

where  $c \in X_j$  and  $d \in X_k$ .  $\mathbb{P}(c) = \mathbb{P}(X_j = c)$  is the probability that feature  $X_j$  takes on category  $c$ , and the same for  $\mathbb{P}(d)$ .  $\mathbb{P}(c,d) = \mathbb{P}(X_j = c, X_k = d)$  is the joint probability that feature  $X_j$  takes on category  $c$  and  $X_k$  category  $d$ . Normalized mutual information scales MI from  $[0, \infty[$  to  $[0, 1]$  (NMI may exceed this range under certain circumstances):

$$NMI(X_j, X_k) = \frac{2 \cdot MI(X_j, X_k)}{H(X_j) + H(X_k)}$$

where

$$H(X_j) = - \sum_{c \in C_j} \mathbb{P}(c) \log \mathbb{P}(c).$$

Where  $C_j$  is the set of all categories  $X_j$  can take on. To use (normalized) mutual information with numerical features, we discretize the observed values  $\mathbf{x}_j$  of the feature  $X_j$  into equally sized bins. The number of bins is determined using the Freedman-Diaconis rule (Freedman and Diaconis 1981):

$$\text{nbins}_j = \left\lceil \frac{\max(\mathbf{x}_j) - \min(\mathbf{x}_j)}{2 \cdot \frac{\text{IQR}(\mathbf{x}_j)}{n^{1/3}}} \right\rceil$$

$\text{IQR}(\mathbf{x}_j)$  is the interquartile range of  $\mathbf{x}_j$ ,  $n$  the number of data instances, and we round up to the next larger integer.

Now let's have a look at the raw dependence data and the normalized mutual information for the features in the bike sharing data in Figure 5.1.

The NMI analysis overall confirms the impression from the correlation analysis. In addition, we get insights about the categorical features: The season shares information with temperature and count 2 days before, which isn't surprising. The pair plots also confirm that correlation coefficients are good measures of dependence for this dataset, since the numerical features don't show extravagant dependence patterns, but mostly linear ones. For the next data example, the dependence analysis has more surprises.

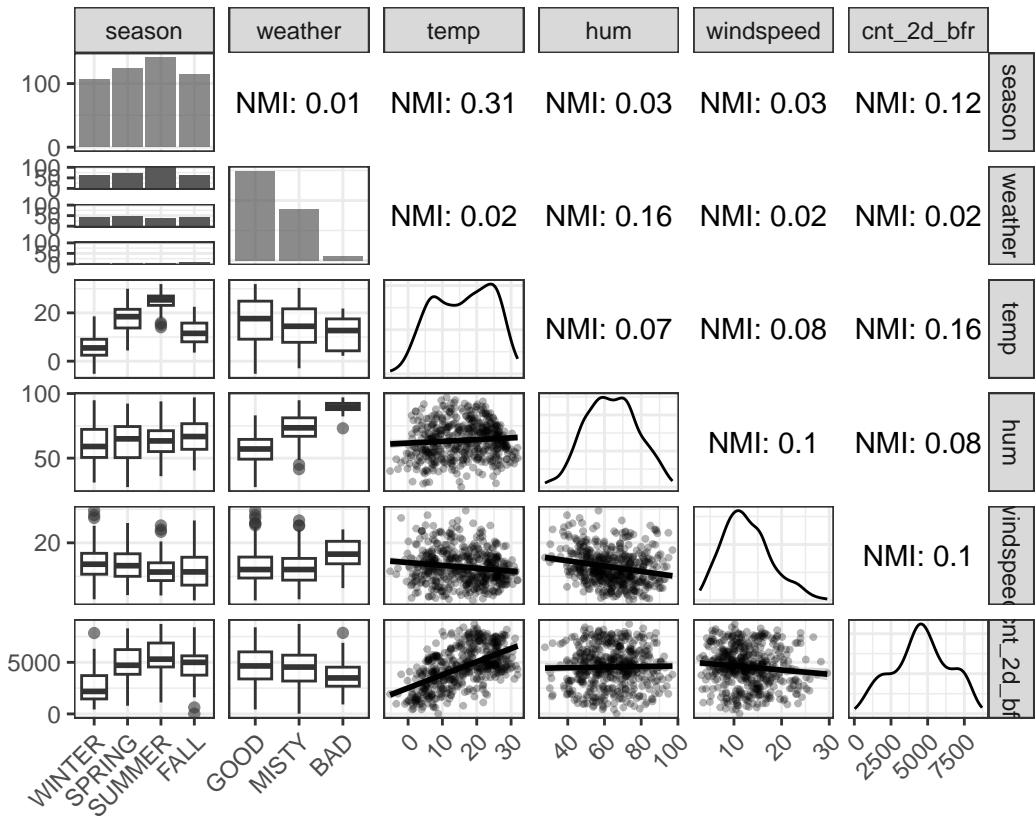
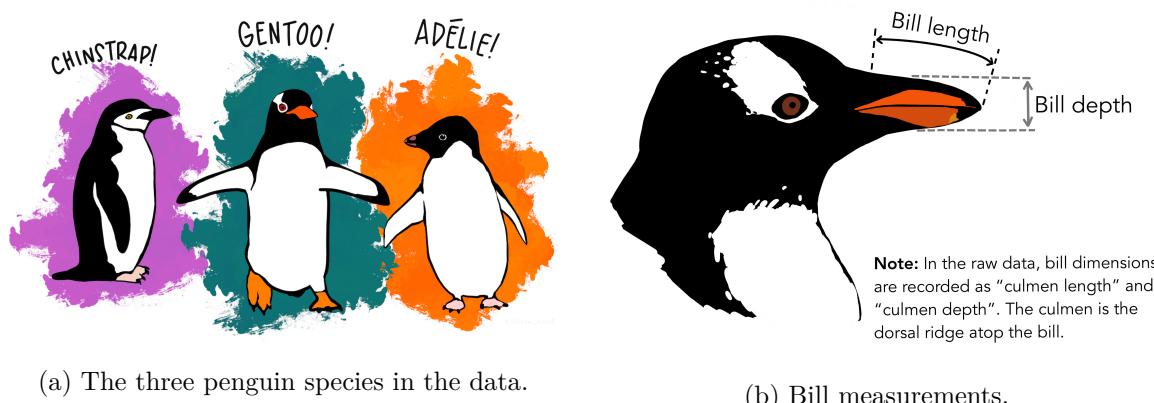


Figure 5.1: Normalized mutual information and pair plots for the features in the bike sharing data. I left out holiday and workday to make the plots easier to read.

## 5.2 Palmer penguins (classification)

For classification, we will use the Palmer penguins data. This cute dataset contains measurements from 333 penguins from the Palmer Archipelago in Antarctica (visualized in Figure 5.2). The dataset was collected and published by Gorman, Williams, and Fraser (2014), and the Palmer Station in Antarctica, which is part of the Long Term Ecological Research Network. The paper studies differences in appearance between male and female, among other things. That's why we'll use male/female classification based on body measurements (as the dataset creators did in their paper).



(a) The three penguin species in the data.

(b) Bill measurements.

Figure 5.2: Palmer penguins artwork by @allison\_horst.

Each row represents a penguin and contains the following information:

- Sex of the penguin (male/female), which is the classification target (`sex`).
- Species of penguin, which is one of Chinstrap, Gentoo, or Adelie (`species`).
- Body mass of the penguin, measured in grams (`body_mass_g`).
- Length of the bill (the beak), measured in millimeters (`bill_length_mm`).
- Depth of the bill, measured in millimeters (`bill_depth_mm`).
- Length of the flipper (the “tail”), measured in millimeters (`flipper_length_mm`).

11 penguins had missing data. Since the purpose of this data is to demonstrate interpretable machine learning methods and not an in-depth study of penguins, I simply dropped penguin data with missing values. The dataset is loaded using the `palmerpenguins` R package (Allison M. Horst, Hill, and Gorman 2020).

### 5.2.1 Classifying penguin sex (male / female)

For the data examples, I trained the following models, using a simple split into training 2/3 and holdout test data 1/3. To assess the performance of the models, I measured log loss and

accuracy on the test data. The results are shown in Table 5.3.

The logistic regression model is actually 3 models: I first split the data by species, trained a logistic regression model, and combined the performance results. That's also what the Gorman, Williams, and Fraser (2014) did in their paper. This is also the model that performed best. For the random forest and for the decision tree, I treated species as a feature. This didn't work out so well for the decision tree, but the performance of the random forest is close to the logistic regression models, at least in terms of accuracy.

Table 5.3: Comparison of model performance for penguin sex classification (male/female).

Model	Log_Loss	Accuracy
Logistic Regression (by Species)	0.16	0.93
Random Forest	0.19	0.92
Decision Tree	1.23	0.86

### 5.2.2 Feature dependence

Let's have a look at how the penguin body measurements are correlated (Pearson correlation).

Table 5.4: Pearson correlation between the penguin features.

Variable 1	Variable 2	Correlation
bill_depth_mm	bill_length_mm	-0.23
bill_depth_mm	flipper_length_mm	-0.58
bill_length_mm	flipper_length_mm	0.65
bill_depth_mm	body_mass_g	-0.47
bill_length_mm	body_mass_g	0.59
flipper_length_mm	body_mass_g	0.87

Table 5.4 shows that especially the body mass and the flipper length are strongly correlated. But also other features are correlated, like flipper length and bill length, or flipper length and bill depth. However, Pearson correlation only tells half of the story, since it only measures linear dependence. Let's have a look at the pair plots of the features, along with the normalized mutual information.

Figure 5.3 shows a much more nuanced picture than the Pearson correlation revealed. For example, the normalized mutual information between body mass and bill depth is similar to the NMI between body mass and flipper length. But the correlation between body mass and bill depth is much lower. The reason is that linear correlation isn't capturing the dynamics, at

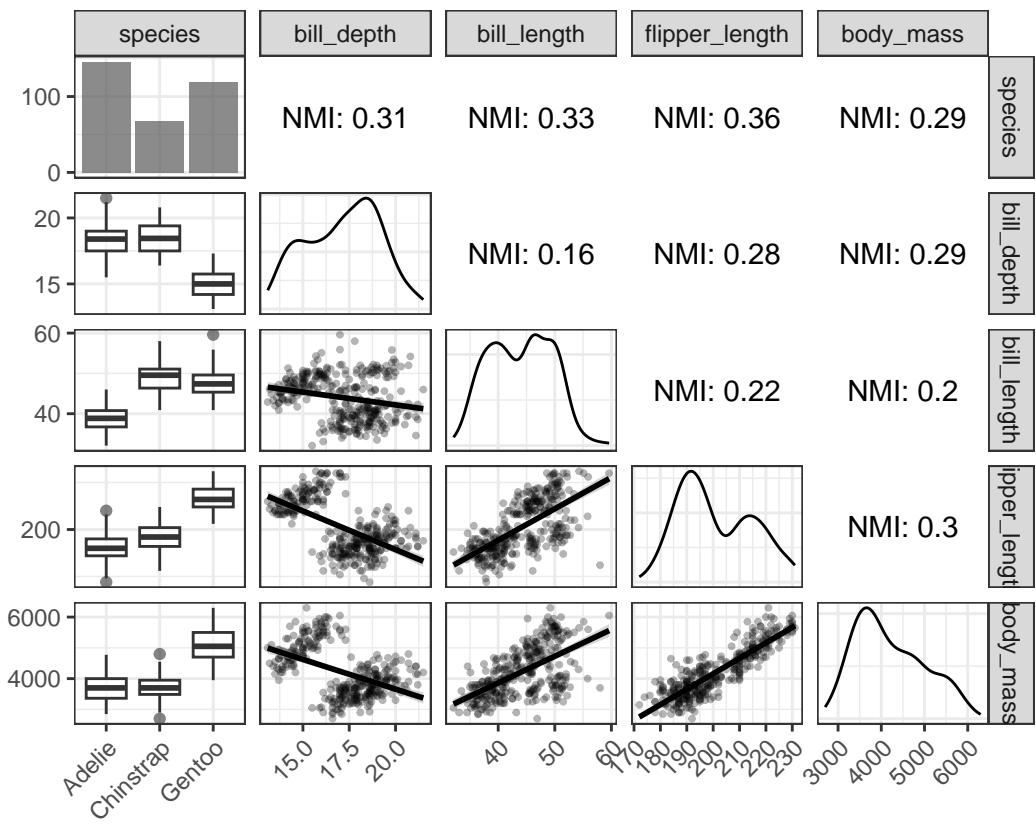


Figure 5.3: Pairwise Pearson correlation between the numerical penguin features.

least not when bundling all the penguins together. The reason why linear correlation doesn't work well here is Simpson's paradox. In the Simpson's paradox a trend appears in several groups of the data, but disappears or flips when combining all data. Combining all penguin data turns a positive correlation between bill depth and body mass into a negative one, as shown in Figure 5.4. The reason is that Gentoo penguins are heavier and have deeper bills. Mutual information overcomes this problem.

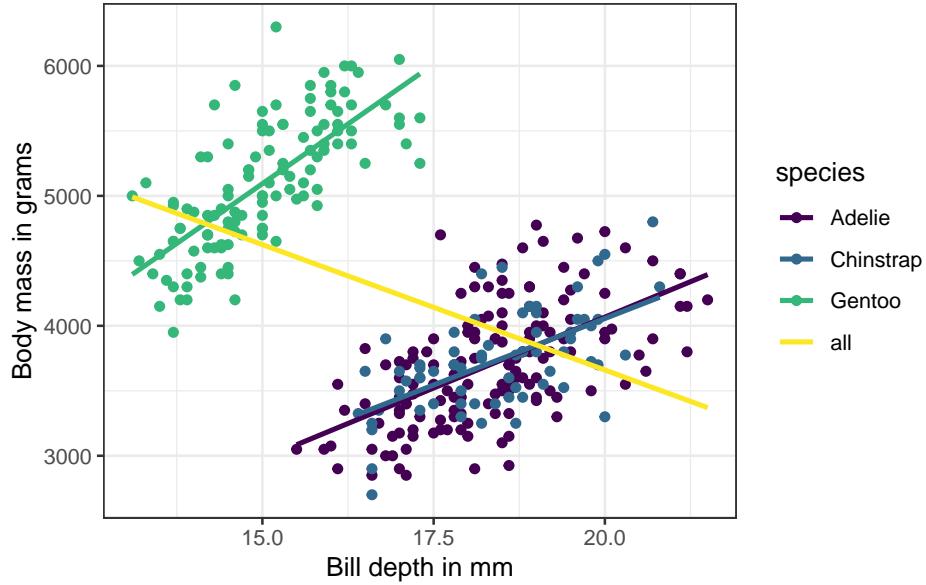


Figure 5.4: Penguin weight versus bill depth for all 3 species. The lines are regression lines for predicting bill depth from body mass for different subsets of the penguins.

A note on modeling all penguins together: Throughout the book, I'll treat the penguins as one dataset by default, using species as a feature. Arguably, it might be better to always model and analyze the penguins by species. However, this is a very common tension in machine learning: Often, data instances belong to a cluster or entity – think of forecasting sales for different stores or classifying lab samples from the same batches – and we have to decide whether to use separate models or using the entity id as a feature.

# **Part I**

# **Interpretable Models**

# 6 Linear Regression

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

Linear models can be used to model the dependence of a regression target  $y$  on features  $\mathbf{x}$ . The learned relationships can be written for a single instance  $i$  as follows:

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

The predicted outcome of an instance is a weighted sum of its  $p$  features. The betas  $\beta_j, j \in 1, \dots, p$  represent the learned feature weights or coefficients. The first weight in the sum,  $\beta_0$ , is called the intercept and is not multiplied with a feature. The epsilon  $\epsilon$  is the error we still make, i.e. the difference between the prediction and the actual outcome.<sup>1</sup> These errors are assumed to follow a Gaussian distribution, which means that we make errors in both negative and positive directions and make many small errors and few large errors.

To find the best coefficient, we typically minimize the squared differences between the actual and the estimated outcomes:

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

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

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

---

<sup>1</sup>This chapter views linear regression through a statistician’s lens. You can also define linear regression without  $\epsilon$ .

influence of the drug and at the same time take sex, age, and other features into account in an interpretable way.

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

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

 Assumptions are optional.

You only need the assumptions to get further things out of the linear model like confidence intervals.

## Linearity

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

## Normality

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

## Homoscedasticity (constant variance)

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

## Independence

It’s assumed that each instance is independent of any other instance.

If you perform repeated measurements, such as multiple blood tests per patient, the data

points are not independent. For dependent data, you need special linear regression models, such as mixed effect models or GEEs.

If you use the “normal” linear regression model, you might draw wrong conclusions from the model.

### Fixed features

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

### Absence of multicollinearity

You do not want strongly correlated features because this messes up the estimation of the weights. In a situation where two features are strongly correlated, it becomes problematic to estimate the weights because the feature effects are additive and it becomes indeterminable to which of the correlated features to attribute the effects.

## 6.1 Interpretation

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

- Numerical feature: Increasing the numerical feature by one unit changes the estimated outcome by its weight. An example of a numerical feature is the size of a house.
- Binary feature: A feature that takes one of two possible values for each instance. An example is the feature “House comes with a garden”. One of the values counts as the reference category (in some programming languages encoded with 0), such as “No garden”. Changing the feature from the reference category to the other category changes the estimated outcome by the feature’s weight.
- Categorical feature with multiple categories: A feature with a fixed number of possible values. An example is the feature “floor type,” with possible categories “carpet,” “laminate,” and “parquet.” A solution to deal with many categories is one-hot encoding, meaning that each category has its own binary column. For a categorical feature with L categories, you only need L-1 columns because the L-th column would have redundant information. For example, when columns 1 to L-1 all have value 0 for one instance, we know that the categorical feature of this instance takes on category L. The interpretation for each category is then the same as the interpretation for binary features. Some languages, such as R, allow you to encode categorical features in various ways, as [described later in this chapter](#).
- Intercept  $\beta_0$ : The intercept is the feature weight for the “constant feature,” which is always 1 for all instances. Most software packages automatically add this “1” feature

to estimate the intercept. The interpretation is: For an instance with all numerical feature values at zero and the categorical feature values at the reference categories, the model prediction is the intercept weight. The interpretation of the intercept is usually not relevant because instances with all feature values at zero often make no sense. The interpretation is only meaningful when the features have been standardized (mean of zero, standard deviation of one). Then the intercept reflects the predicted outcome of an instance where all features are at their mean value.

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

### Interpretation

- **Numerical feature:** An increase of feature  $x_j$  by one unit increases the prediction for  $y$  by  $\beta_j$  units when all other feature values remain fixed.
- **Categorical feature:** Changing feature  $x_j$  from the reference category to the other category increases the prediction for  $y$  by  $\beta_j$  when all other features remain fixed.

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

$$R^2 = 1 - \frac{SSE}{SST}$$

SSE is the squared sum of the error terms:

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

SST is the squared sum of the data variance:

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

The SSE tells you how much variance remains after fitting the linear model, which is measured by the squared differences between the predicted and actual target values. SST is the total variance of the target outcome. R-squared tells you how much of your variance can be explained by the linear model. R-squared usually ranges between 0 for models where the model does not explain the data at all and 1 for models that explain all of the variance in your data. It's also possible for R-squared to take on a negative value without violating any mathematical rules.

This happens when SSE is greater than SST, which means that a model does not capture the trend of the data and fits to the data worse than using the mean of the target as the prediction.

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

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

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

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

### Feature Importance

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

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

Let's examine what this formula tells us: The importance of a feature increases with increasing weight. This makes sense. The more variance the estimated weight has (= the less certain we are about the correct value), the less important the feature is. This also makes sense.

## 6.2 Example

In this example, we use the linear regression model to predict the [number of rented bikes](#) on a particular day, given weather and calendar information. For the interpretation, we examine the estimated regression weights. The features consist of numerical and categorical features. For each feature, Table 6.1 shows the estimated weight, the standard error of the estimate, and the absolute value of the t-statistic.

Interpretation of a numerical feature (`temp`): An increase of the temperature by 1 degree Celsius increases the predicted number of bikes by 51.0, when all other features remain fixed.

Interpretation of a categorical feature (`weather`): The estimated number of bikes is -1804.1 lower when weather is bad (raining, snowing, or stormy), compared to good weather – again assuming that all other features do not change. When the weather is misty, the predicted

Table 6.1: Coefficients, standard errors (SE), and absolute value of the t-statistic for linear model predicting bike rentals.

	Weight	SE	t
(Intercept)	2742.4	339.0	8.1
seasonSPRING	454.8	169.9	2.7
seasonSUMMER	263.3	218.7	1.2
seasonFALL	636.9	152.7	4.2
holidayY	-441.1	239.0	1.8
workdayY	242.0	101.9	2.4
weatherMISTY	-345.3	120.2	2.9
weatherBAD	-1804.1	306.0	5.9
temp	51.0	10.3	5.0
hum	-21.7	4.5	4.8
windspeed	-46.5	9.5	4.9
cnt_2d_bfr	0.6	0.0	19.3

number of bikes is -345.3 lower compared to good weather, given all other features remain the same.

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

## 6.3 Weight and effect plot

Visualizations like the weight and the effect plot make the linear regression model easy and quick to grasp for humans.

### Weight plot

The information from the weight table (weight and variance estimates) can be visualized in a weight plot. Figure 6.1 shows the results from the previous linear regression model.

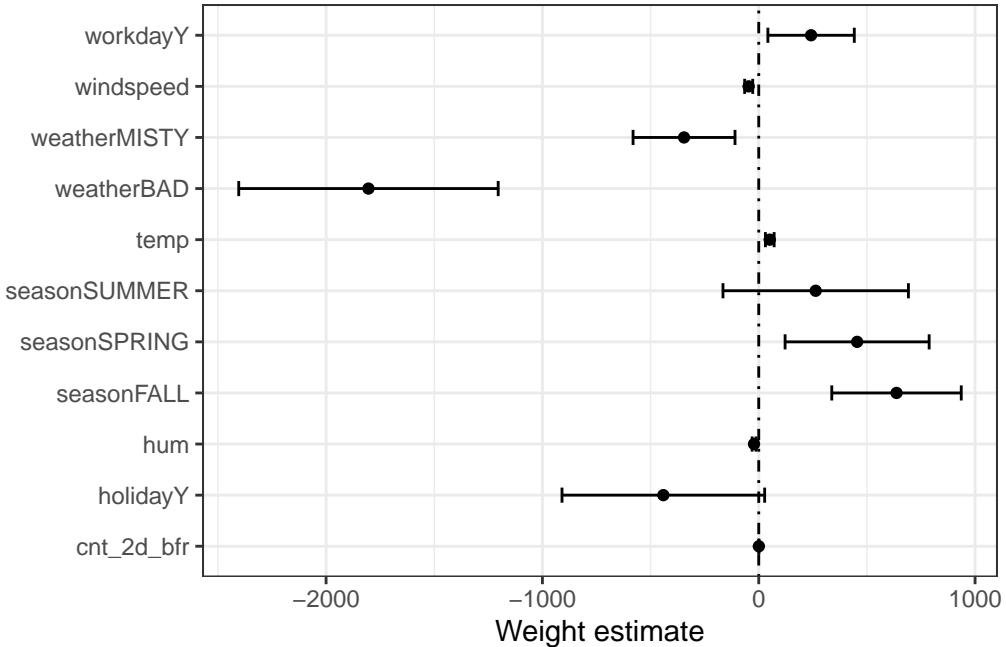


Figure 6.1: Linear regression estimates for bike rental data. Weights are displayed as points and the 95% confidence intervals as lines.

The weight plot shows that bad weather has a strong negative effect on the predicted number of bikes. The weight for summer is positive, but the 95% confidence interval contains 0, so there is no significant difference between summer and winter, given all other features. Since the model accounts for temperature, the summer coefficient is about the **additional** effect of summer. Some confidence intervals are very short, and the estimates are close to zero, yet the feature effects were statistically significant. Temperature is one such candidate. The problem with the weight plot is that the features are measured on different scales. While for the weather the estimated weight reflects the difference between good and bad weather, for temperature it only reflects an increase of 1 degree Celsius. You can make the estimated weights more comparable by scaling the features (zero mean and standard deviation of one) before fitting the linear model.

⚠ Don't compare weights of features with different scales

The units a feature is measured in affects the magnitude of coefficients/weights. For example, if you multiply a feature by 1000, like when converting from kilogram to gram, then the new coefficient will be smaller by a factor of 1/1000.

## Effect plot

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

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

The effects can be visualized with [boxplots](#), as in Figure 6.2. The box in a boxplot contains the effect range for half of the data (25% to 75% effect quantiles). The vertical line in the box is the median effect, i.e., 50% of the instances have a lower and the other half a higher effect on the prediction. The dots are outliers, defined as points that are more than  $1.5 * \text{IQR}$  (interquartile range, that is, the difference between the first and third quartiles) above the third quartile, or less than  $1.5 * \text{IQR}$  below the first quartile. The two horizontal lines, called the lower and upper whiskers, connect the points below the first quartile and above the third quartile that are not outliers. If there are no outliers, the whiskers will extend to the minimum and maximum values.

The categorical feature effects can be summarized in a single boxplot, compared to the weight plot, where each category has its own row.

The largest contributions to the expected number of rented bikes come from the temperature feature and previous bike count. The temperature has a broad range of how much it contributes to the prediction. The count feature goes from zero to large positive contributions. For effects with a negative weight, the instances with a positive effect are those that have a negative feature value. For example, days with a high negative effect of windspeed are the ones with high wind speeds.

### 6.3.1 Explain individual predictions with effect plots

How much has each feature of an instance contributed to the prediction? This can be answered by computing the effects for this instance. An interpretation of instance-specific effects only makes sense in comparison to the distribution of the effect for each feature. We want to explain the prediction of the linear model for the 6-th instance from the bike dataset. The feature values of this instance are displayed in Table 6.2.

To obtain the feature effects of this instance, we have to multiply its feature values by the corresponding weights from the linear regression model. For a temperature of -0.05 degrees Celsius, the effect is  $-0.05 \cdot 51.01 = -2.69$ . We add these individual effects as crosses to the

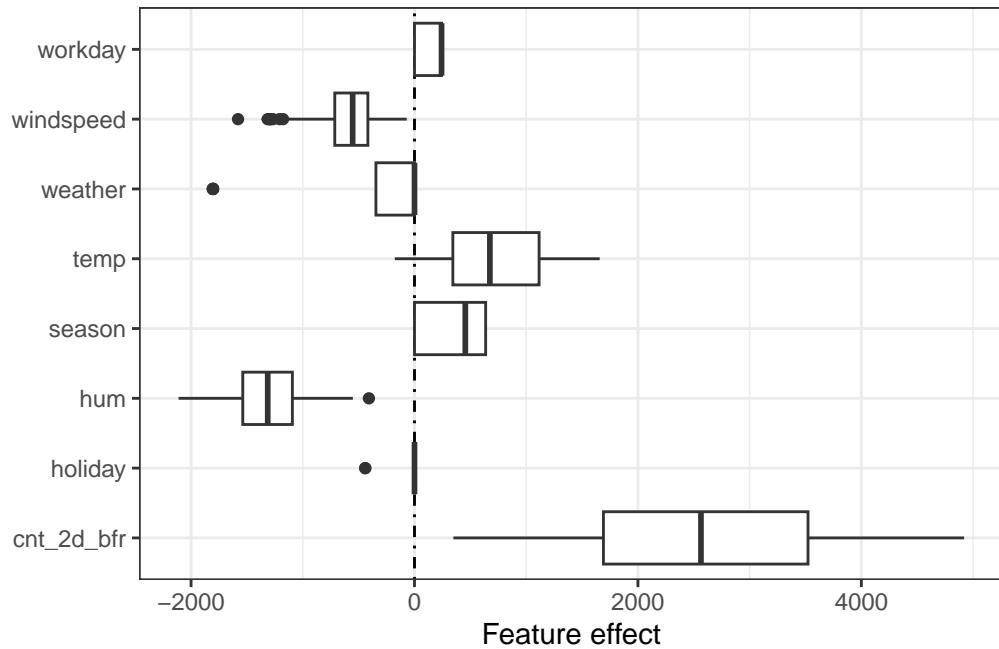


Figure 6.2: Effect plot for linear regression results for the bike rental data. The boxplots show the distribution of effects (= feature value times feature weight) across the data per feature.

Table 6.2: Feature values for instance 6

Feature	Value
season	WINTER
holiday	N
workday	Y
weather	MISTY
temp	-0.052723
hum	68.6364
windspeed	8.182844
cnt_2d_bfr	822
cnt	1263

effect plot, which shows us the distribution of the effects in the data, as visualized in Figure 6.3. This allows us to compare the individual effects with the distribution of effects in the data.

If we average the predictions for the test data instances, we get an average of 4452. In comparison, the prediction of the 6-th instance is small, since only 1238 bikes rentals are predicted. The effect plot reveals the reason why. The boxplots show the distributions of the effects for all instances of the dataset; the crosses show the effects for the 6-th instance. The 6-th instance has a low temperature effect because on this day the temperature was 0 degrees, which is low compared to most other days (and remember that the weight of the temperature feature is positive). Also, the effect of the feature `cnt_2d_bfr` is small compared to the other data instances because fewer bikes were rented early 2011.

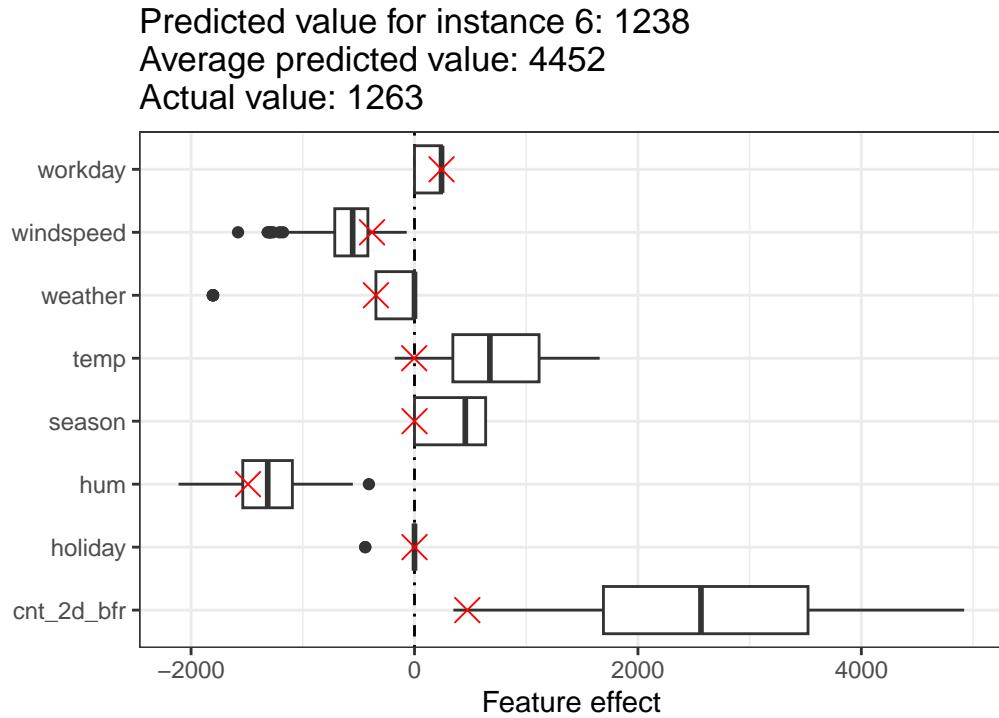


Figure 6.3: Effect plot for one linear model prediction for the bike data. The boxplots show overall effect distributions while the crosses mark the effects for the instance of interest.

## 6.4 Encoding categorical features

There are several ways to encode a categorical feature, and the choice influences the interpretation of the weights. The standard in linear regression models is treatment coding, which

is sufficient in most cases. Using different encodings boils down to creating different (design) matrices from a single column with the categorical feature. This section presents three different encodings, but there are many more. The following example has four instances and a categorical feature with three categories. For the first two instances, the feature takes category A; for instances three, category B; and for the last instance, category C. So our vector looks like this:

$$\begin{pmatrix} A \\ A \\ B \\ C \end{pmatrix}$$

### Treatment coding

In treatment coding, the weight per category is the estimated difference in the prediction between the corresponding category and the reference category. The intercept of the linear model is the mean of the reference category (when all other features remain the same). The first column of the design matrix is the intercept, which is always 1. Column two indicates whether instance i is in category B, column three indicates whether it is in category C. There's no need for a column for category A, because then the linear equation would be overspecified and no unique solution for the weights can be found. It's sufficient to know that an instance is neither in category B nor C.

$$\begin{pmatrix} A \\ A \\ B \\ C \end{pmatrix} \Rightarrow \begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix}$$

### Effect coding

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

$$\begin{pmatrix} A \\ A \\ B \\ C \end{pmatrix} \Rightarrow \begin{pmatrix} 1 & -1 & -1 \\ 1 & -1 & -1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix}$$

## Dummy coding

The  $\beta$  per category is the estimated mean value of  $\mathbf{y}$  for each category (given all other feature values are zero or the reference category). Note that the intercept has been omitted here so that a unique solution can be found for the linear model weights. Another way to mitigate this multicollinearity problem is to leave out one of the categories.

$$\begin{pmatrix} A \\ A \\ B \\ C \end{pmatrix} \Rightarrow \begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

If you want to dive a little deeper into the different encodings of categorical features, check out [this overview webpage](#), and [this blog post](#).

## 6.5 Sparse linear models

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

### 6.5.1 Lasso

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

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

Lasso adds a term to this optimization problem.

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

The term  $\|\beta\|_1$ , the L1-norm of the feature vector, leads to a penalization of large weights. Since the L1-norm is used, many of the weights receive an estimate of 0, and the others are shrunk. The parameter  $\lambda$  controls the strength of the regularizing effect and is usually tuned by cross-validation. Especially when  $\lambda$  is large, many weights become 0. The feature weights can be visualized as a function of the penalty term  $\lambda$ . Each feature weight is represented by a curve in Figure 6.4. With increasing penalty of the weights, fewer and fewer features receive a non-zero weight estimate. These curves are also called regularization paths.

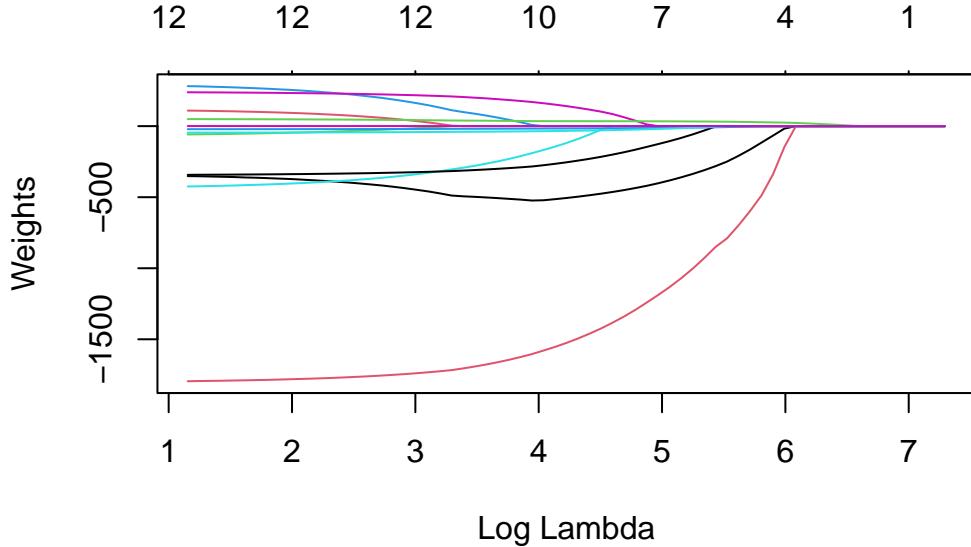


Figure 6.4: Effect of penalization term  $\lambda$  on weight estimates in a linear model. The number above the plot is the number of non-zero weights.

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

### Example with Lasso

We'll predict bike rentals using Lasso. We set the number of features we want to have in the model beforehand. Let's first set the number to 2 features so we get results in Table 6.3.

The first two features with non-zero weights in the Lasso path are temperature (`temp`) and the previous count (`cnt_2d_bfr`). Now, let's select 5 features. Results are in Table 6.4. Note that the weights for "temp" and "cnt\_2d\_bfr" differ from the model with two features. The

Table 6.3: Weight estimates when using Lasso and setting the number of features to two.

	Weight
seasonWINTER	0.00
seasonSPRING	0.00
seasonSUMMER	0.00
seasonFALL	0.00
holidayY	0.00
workdayY	0.00
weatherMISTY	0.00
weatherBAD	0.00
temp	22.45
hum	0.00
windspeed	0.00
cnt_2d_bfr	0.48

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

### Other methods for sparsity in linear models

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

Pre-processing methods:

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

Step-wise methods:

- Forward selection: Fit the linear model with one feature. Do this with each feature. Select the model that works best (e.g., highest R-squared). Now again, for the remaining features, fit different versions of your model by adding each feature to your current best

Table 6.4: Weight estimates when using Lasso and setting the number of features to five.

	Weight
seasonWINTER	-249.37
seasonSPRING	0.00
seasonSUMMER	0.00
seasonFALL	0.00
holidayY	0.00
workdayY	0.00
weatherMISTY	0.00
weatherBAD	-789.74
temp	31.14
hum	-6.47
windspeed	0.00
cnt_2d_bfr	0.53

model. Select the one that performs best. Continue until some criterion is reached, such as the maximum number of features in the model.

- Backward selection: Similar to forward selection. But instead of adding features, start with the model that contains all features and try out which feature you have to remove to get the highest performance increase. Repeat this until some stopping criterion is reached.

I recommend using Lasso because it can be automated, considers all features simultaneously, and can be controlled via  $\lambda$ . It also works for [logistic regression](#) for classification.

## 6.6 Strengths

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

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

...

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

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

Linear models **can create truthful explanations**, as long as the linear equation is an appropriate model for the relationship between features and outcome. The more non-linearities and interactions there are, the less accurate the linear model will be, and the less truthful the explanations become.

## 6.7 Limitations

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

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

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

Judging by the attributes that constitute a good explanation, as presented [in the Human-Friendly Explanations chapter](#), **linear models do not create the best explanations**. They are contrastive, but the reference instance is a data point where all numerical features are zero, and the categorical features are at their reference categories. This is usually an artificial, meaningless instance that is unlikely to occur in your data or reality. There is an exception: If all numerical features are mean-centered (feature minus mean of feature) and all categorical features are effect coded, the reference instance is the data point where all the features take on the mean feature value. This might also be a non-existent data point, but it might at least be more likely or more meaningful. In this case, the weights times the feature values (feature effects) explain the contribution to the predicted outcome contrastive to the “mean-instance”.

By default, **linear models do not create selective explanations**.

# 7 Logistic Regression

Logistic regression models the probabilities for classification problems with two possible outcomes. It's an extension of the linear regression model for class outcomes.<sup>1</sup>

## 7.1 Don't use linear regression for classification

The linear regression model can work well for regression, but fails for classification. Why is that? In the case of two classes, you could label one of the classes with 0 and the other with 1 and use linear regression. Technically, it works, and most linear model programs will spit out weights for you. But there are a few problems with this approach: A linear model does not output probabilities, but it treats the classes as numbers (0 and 1) and fits the best hyperplane (for a single feature, it's a line) that minimizes the distances between the points and the hyperplane. So it simply interpolates between the points, and you cannot interpret it as probabilities.

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

Since the predicted outcome is not a probability, but a linear interpolation between points, there is no meaningful threshold at which you can distinguish one class from the other, as illustrated in Figure 7.1. A good explanation of this issue has been given on [Stackoverflow](#).

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

---

<sup>1</sup>To be accurate, logistic regression is a regression model, since the output is continuous. But together with a decision threshold, like 0.5, it can be used for classification as well.

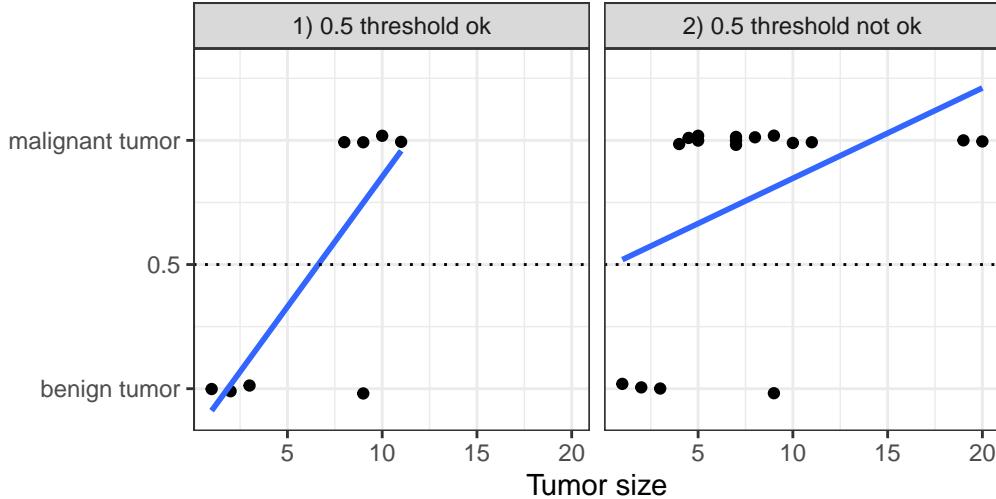


Figure 7.1: Estimating a linear model to predict (simulated) tumor type based on tumor size. Points are slightly jittered to reduce over-plotting. Left: Using a cutoff at 0.5 would yield an acceptable classifier. Right) Adding just two more data points completely changes the regression estimates and makes a 0.5 cutoff meaningless. The model also produces predictions  $> 1$

## 7.2 Theory

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

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

The logistic function is visualized in Figure 7.2.

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

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

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

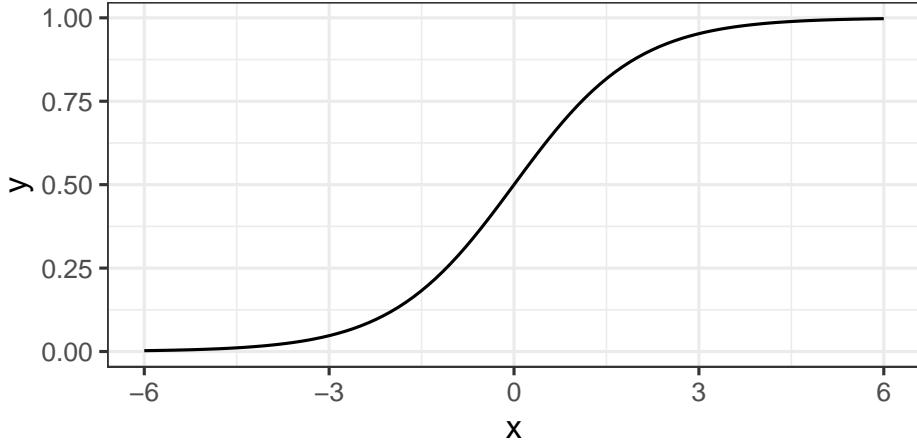


Figure 7.2: The logistic function outputs numbers between 0 and 1. At input 0, it outputs 0.5.

$$\mathbb{P}(Y^{(i)} = 1) = \text{logistic}(\mathbf{x}^{(i)T} \boldsymbol{\beta}) = \frac{1}{1 + \exp(-(\beta_0 + \beta_1 x_1^{(i)} + \dots + \beta_p x_p^{(i)}))}$$

Let's revisit the tumor size example again. But instead of the linear regression model, we use the logistic regression model and get a much better fitting curve, see Figure 7.3.

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

### 7.3 Interpretation

The interpretation of the weights in logistic regression differs from the interpretation of the weights in linear regression since the outcome in logistic regression is a value between 0 and 1. The weights don't influence the probability linearly any longer. To interpret the weights, we need to reformulate the equation for the interpretation so that only the linear term is on the right side of the formula.

$$\ln \left( \frac{\mathbb{P}(Y = 1)}{1 - \mathbb{P}(Y = 1)} \right) = \ln \left( \frac{\mathbb{P}(Y = 1)}{\mathbb{P}(Y = 0)} \right) = \beta_0 + \beta_1 x_1 + \dots + \beta_p x_p$$

We call the term in the  $\ln()$  function “odds” (probability of event divided by probability of no event), and wrapped in the logarithm, it is called log odds.

This formula shows that the logistic regression model is a linear model for the log odds. Great! That doesn't sound helpful!

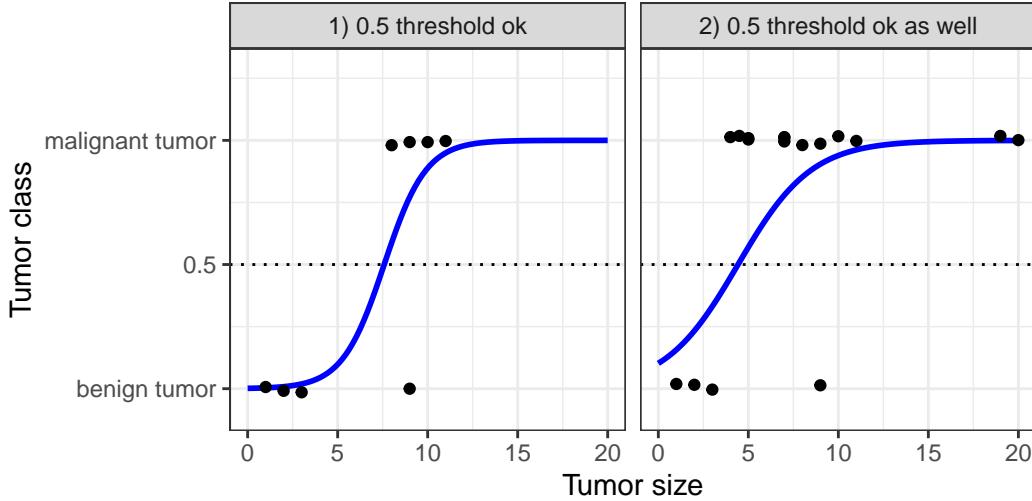


Figure 7.3: Left: Logistic regression model fitted to predict tumor class from tumor size based on simulated tumor data. Right: Logistic regression is even robust when adding two outliers.

**⚠ Logistic regression is multiplicative**

On the level of probabilities, logistic regression is not linear in the features. Meaning an increase by one unit in the features doesn't increase the probability by  $\beta_j$ , but rather changes the probability multiplicatively.

With a little shuffling of the terms, you can figure out how the prediction changes when one of the features  $X_j$  is changed by 1 unit. To do this, we can first apply the exp function to both sides of the equation:

$$\frac{\mathbb{P}(Y = 1)}{1 - \mathbb{P}(Y = 1)} = \text{odds} = \exp(\beta_0 + \beta_1 x_1 + \dots + \beta_p x_p)$$

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

$$\frac{\text{odds}_{x_j+1}}{\text{odds}_{x_j}} = \frac{\exp(\beta_0 + \beta_1 x_1 + \dots + \beta_j(x_j + 1) + \dots + \beta_p x_p)}{\exp(\beta_0 + \beta_1 x_1 + \dots + \beta_j x_j + \dots + \beta_p x_p)}$$

We apply the following rule:

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

And we remove many terms:

$$\frac{\text{odds}_{x_j+1}}{\text{odds}_{x_j}} = \exp(\beta_j(x_j + 1) - \beta_j x_j) = \exp(\beta_j)$$

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

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

- Numerical feature: If you increase the value of  $x_j^{(i)}$  by one unit, the estimated odds change by a factor of  $\exp(\beta_j)$ .
- Binary categorical feature: One of the two values of the feature is the reference category (in some languages, the one encoded in 0). Changing  $x_j^{(i)}$  from the reference category to the other category changes the estimated odds by a factor of  $\exp(\beta_j)$ .
- Categorical feature with more than two categories: One solution to deal with multiple categories is one-hot-encoding, meaning that each category has its own column. You only need L-1 columns for a categorical feature with L categories, otherwise it is over-parameterized. The L-th category is then the reference category. You can use any other encoding that can be used in linear regression. The interpretation for each category then is equivalent to the interpretation of binary features.
- Intercept  $\beta_0$ : When all numerical features are zero and the categorical features are at the reference category, the estimated odds are  $\exp(\beta_0)$ . The interpretation of the intercept weight is usually not relevant.

#### 💡 Enhance interpretation with model-agnostic methods

If you want to interpret the outcome on the level of probabilities, then you have to use model-agnostic methods, such as the [partial dependence plot](#).

Table 7.1: Logistic regression results for predicting penguin P(female): Weight, standard errors and odds ratios.

	Weight	Std. Error	Odds ratio
(Intercept)	95.86	29.29	4.28931e+41
bill_depth_mm	-2.06	0.83	0.13
bill_length_mm	-0.53	0.19	0.59
flipper_length_mm	-0.16	0.11	0.85
chonkinessRegular_Penguin	0.65	1.25	1.92
chonkinessAbsolute_Unit	-0.84	1.66	0.43

## 7.4 Example

We use logistic regression to predict whether a penguin is female for Chinstrap penguins based on body measurements. The feature `chonkiness` is a discretization of the feature `body_mass_g`: Light penguins (0% to 25% quantile) are categorized as “Smol\_Penguin”, most penguins are “Regular\_Penguin”, and those with the highest body mass (75% to 100%) are categorized as “Absolute\_Unit”. Normally, I don’t recommend discretizing continuous features because you lose information. In this case, I used binarization to illustrate the interpretation of a categorical feature with logistic regression, and I also love the word “chonky” and wanted to use it in a textbook.

Table 7.1 shows the estimated weights, associated odds ratios, and standard errors of the estimates.

Here are two examples of interpretation:

**Numerical feature:** An increase in a penguin’s bill length increases the odds of being female vs. male by a factor of 0.59 , when all other features remain the same.

**Categorical feature:** For the penguins of regular chonkiness, the odds of being female versus male are increased by a factor of 1.92 compared to lightweight penguins, given all other features remain the same. For the chonkiest penguins, the odds of being female versus male are decreased by a factor of 0.43 compared to lightweight penguins, given all other features remain the same. Again, given all other features remain the same.

The intercept is a very large value. This is because it is to be interpreted as odds of being female when all numerical features are set to zero and all categorical features are set to the reference level. This would be a non-chonky penguin with flipper length, bill length, and depth all at zero – not a realistic penguin. You could standardize the features, then the intercept would be more meaningful, but you can also just ignore the intercept.

## 7.5 Strengths

Many of the pros and cons of the [linear regression model](#) also apply to the logistic regression model.

On the good side, the logistic regression model is not only a classification model, but also gives you probabilities. This is a big advantage over models that can only provide the final classification. Knowing that an instance has a 99% probability for a class compared to 51% makes a big difference. However, you should check whether the probabilities are calibrated, meaning whether 60% really means 60%.

Logistic regression can also be extended from binary classification to multi-class classification.

## 7.6 Limitations

Logistic regression has been widely used by many different people, but it struggles with its restrictive expressiveness (e.g., interactions must be added manually), and other models may have better predictive performance.

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

Logistic regression can suffer from **complete separation**. If there is a feature that would perfectly separate the two classes, the logistic regression model can no longer be trained. This is because the weight for that feature would not converge, because the optimal weight would be infinite. This is really a bit unfortunate because such a feature is really useful. But you do not need machine learning if you have a simple rule that separates both classes. The problem of complete separation can be solved by introducing penalization of the weights or defining a prior probability distribution of weights.

## 7.7 Software

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

## 8 GLM, GAM and more

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

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

Let's remember the formula of a linear regression model:

$$\hat{f}(\mathbf{x}) = \beta_0 + \beta_1 x_1 + \dots + \beta_p x_p + \epsilon$$

The linear regression model assumes that the prediction of an instance can be expressed by a weighted sum of its  $p$  features with a random variable  $\epsilon^{(i)}$  that follows a Gaussian distribution. By forcing the data into this corset of a formula, we obtain a lot of model interpretability. The feature effects are additive, meaning no interactions, and the relationship is linear, which allows us to compress the relationship between a feature and the expected outcome into a single number, namely the estimated weight.

But a simple weighted sum is too restrictive for many real-world prediction problems. In this chapter, we will learn about three problems of the classical linear regression model and how to solve them. There are many more problems with possibly violated assumptions, but we will focus on the three shown in Figure 8.1.

**Problem:** The target outcome  $y$  given the features does not follow a Gaussian distribution.  
**Example:** Suppose I want to predict how many minutes I will ride my bike on a given day. As features, I have the type of day, the weather, and so on. If I use a linear model, it could predict negative minutes because it assumes a Gaussian distribution which does not stop at 0

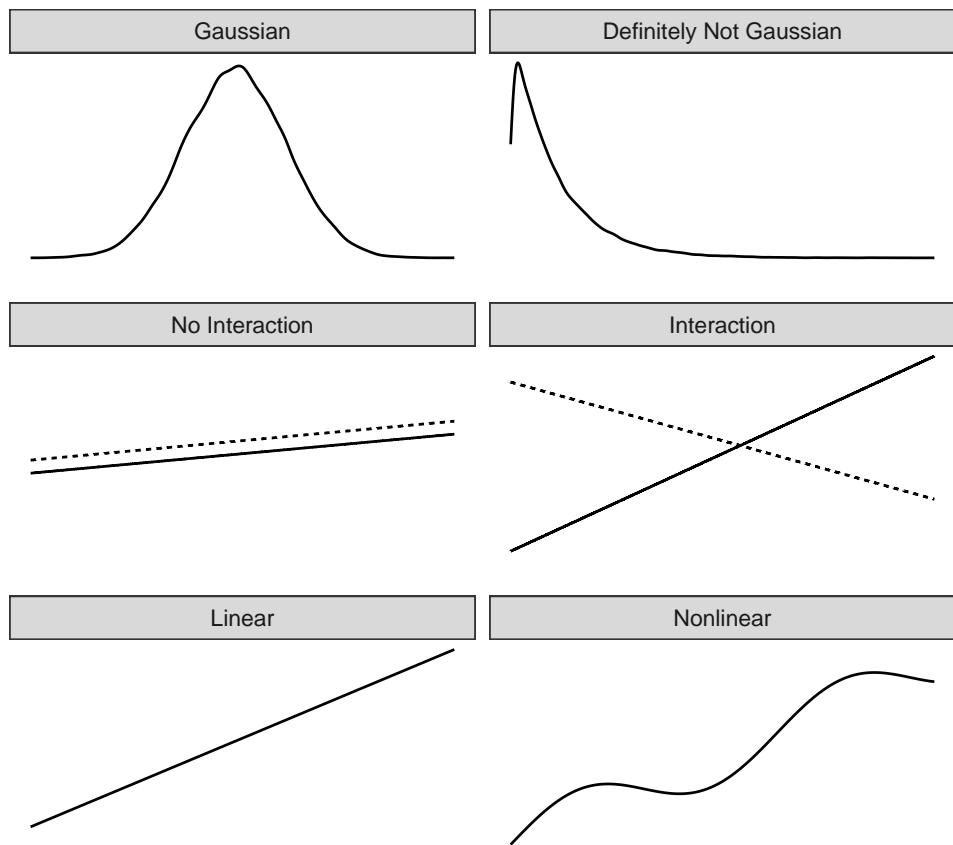


Figure 8.1: Three assumptions of the linear model (left side): Gaussian distribution of the outcome given the features, additivity (= no interactions) and linear relationship. Reality usually does not adhere to those assumptions (right side): Outcomes might have non-Gaussian distributions, features might interact and the relationship might be nonlinear.

minutes. Also, if I would predict probabilities with a linear model, I can get probabilities that are negative or greater than 1.

**Solution:** [Generalized Linear Models \(GLMs\)](#).

**Problem:** The features interact.

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

**Solution:** [Adding interactions](#).

**Problem:** The true relationship between the features and  $y$  is not linear.

**Example:** Between 0 and 25 degrees Celsius, the influence of the temperature on my desire to ride a bike could be linear, which means that an increase from 0 to 1 degree causes the same increase in cycling desire as an increase from 20 to 21. But at higher temperatures my motivation to cycle levels off and even decreases - I do not like to bike when it's too hot.

**Solutions:** [Generalized Additive Models \(GAMs\)](#); [transformation of features](#).

#### 💡 Check for linear model violations

You can empirically check for these problems, like feature interactions, by comparing the solutions with plain linear regression: Does your linear model perform better on validation data when an interaction term is used?

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

## 8.1 Non-Gaussian outcomes - GLMs

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

the logistic regression model assumes a Bernoulli distribution for the outcome and links the expected mean and the weighted sum using the logistic function.

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

$$g(\mathbb{E}[Y|\mathbf{x}]) = \beta_0 + \beta_1 x_1 + \dots + \beta_p x_p = \mathbf{x}^T \boldsymbol{\beta}$$

GLMs consist of three components: The link function  $g$ , the weighted sum  $\mathbf{x}^T \boldsymbol{\beta}$  (sometimes called linear predictor) and a probability distribution from the exponential family that defines  $\mathbb{E}_Y$ .

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

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

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

$$\ln(\mathbb{E}[Y|\mathbf{x}]) = \mathbf{x}^T \boldsymbol{\beta}$$

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

$$\mathbf{x}^T \boldsymbol{\beta} = \ln \left( \frac{\mathbb{E}[Y|\mathbf{x}]}{1 - \mathbb{E}[Y|\mathbf{x}]} \right) = \ln \left( \frac{\mathbb{P}(Y = 1|\mathbf{x})}{1 - \mathbb{P}(Y = 1|\mathbf{x})} \right)$$

And if we solve this equation to have  $\mathbb{P}(Y = 1)$  on one side, we get the logistic regression formula:

$$\mathbb{P}(Y = 1) = \frac{1}{1 + \exp(-\mathbf{x}^T \boldsymbol{\beta})}$$

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

### Examples

I have simulated a dataset on coffee drinking behavior to highlight the need for GLMs. Suppose you have collected data about your daily coffee drinking behavior. If you do not like coffee, pretend it's about tea or something. Along with the number of cups, you record your current stress level on a scale of 1 to 10, how well you slept the night before on a scale of 1 to 10, and whether you had to work on that day. The goal is to predict the number of coffees given the features stress, sleep, and work. I simulated data for 200 days. Stress and sleep were drawn uniformly between 1 and 10 and work yes/no was drawn with a 50/50 chance (what a life!). For each day, the number of coffees was then drawn from a Poisson distribution, modeling the intensity  $\lambda$  (which is also the expected value of the Poisson distribution) as a function of the features sleep, stress and work. You can guess where this story will lead: *“Hey, let’s model this data with a linear model ... Oh it does not work ... Let’s try a GLM with Poisson distribution ... SURPRISE! Now it works!”*. I hope I did not spoil the story too much for you.

Figure 8.2 shows the distribution of the target variable, the number of coffees on a given day. On 84 of the 200 days, you had no coffee at all, and on the most extreme day, you had 10. Let’s naively use a linear model to predict the number of coffees using sleep level, stress level, and work yes/no as features. What can go wrong when we falsely assume a Gaussian distribution? A wrong assumption can invalidate the estimates, especially the confidence intervals of the weights. A more obvious problem is that the predictions do not match the “allowed” domain of the true outcome, as the following Figure 8.3 shows. The linear model does not make sense because it predicts a negative number of coffees. It might also simply have bad performance on test data.

The problem of non-matching distributions can be solved with Generalized Linear Models (GLMs). We can change the link function and the assumed distribution. One possibility is to keep the Gaussian distribution and use a link function that always leads to positive predictions,

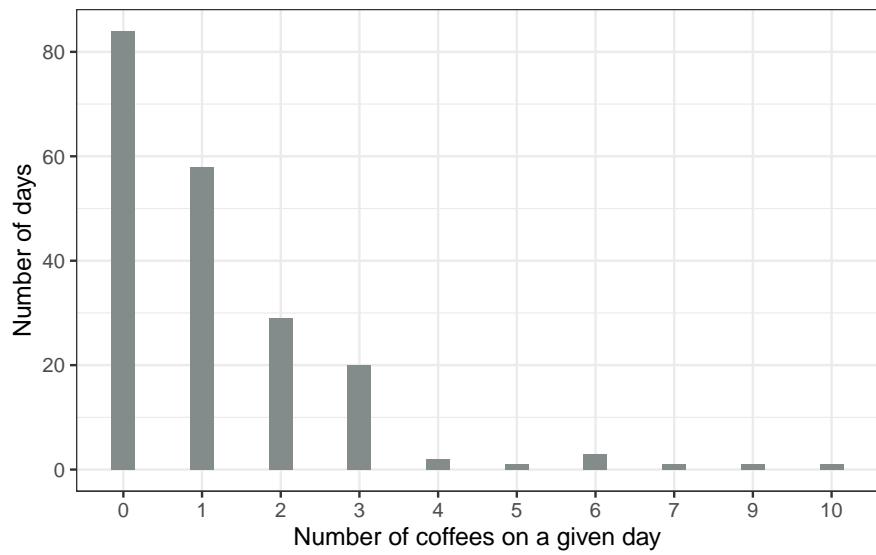


Figure 8.2: Simulated distribution of number of daily coffees for 200 days.

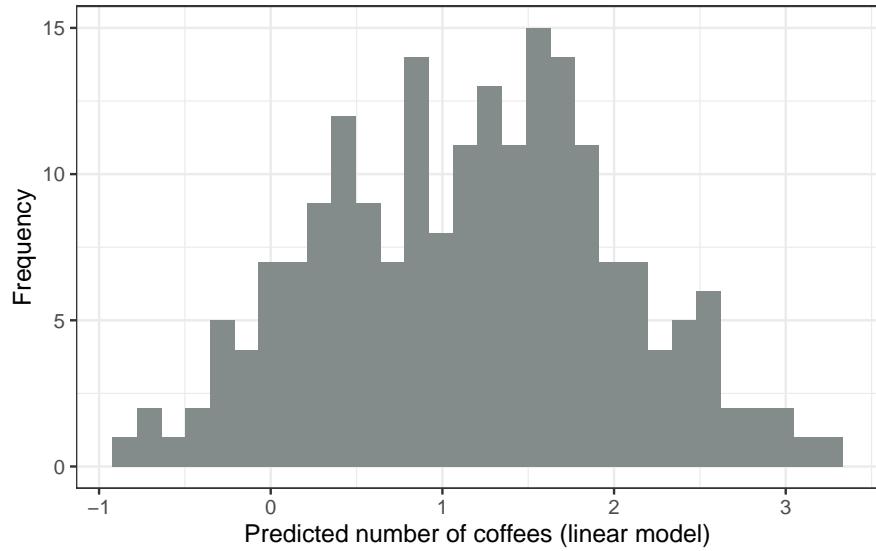


Figure 8.3: Predicted number of coffees dependent on stress, sleep and work. The linear model predicts negative values.

such as the log-link (the inverse is the exp-function), instead of the identity function. Even better: We choose a distribution that corresponds to the data-generating process, and an appropriate link function. Since the outcome is a count, the Poisson distribution is a natural choice, along with the logarithm as the link function. In this case, the data was even generated with the Poisson distribution, so the Poisson GLM is the perfect choice. The fitted Poisson GLM leads to the distribution of predicted values shown in Figure 8.4.

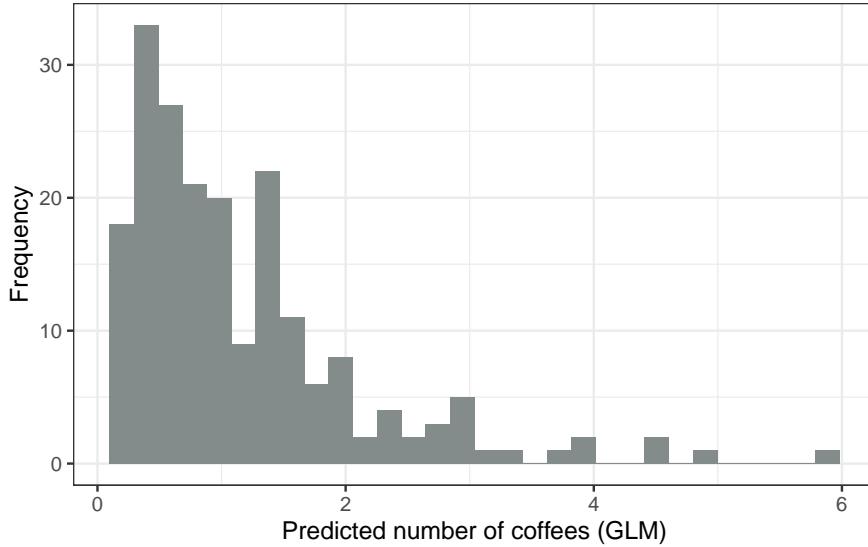


Figure 8.4: Predicted number of coffees dependent on stress, sleep and work. The GLM with Poisson assumption and log link is an appropriate model for this dataset.

No negative amounts of coffees, looks much better now.

### Interpretation of GLM weights

The assumed distribution together with the link function determines how the estimated feature weights are interpreted. In the coffee count example, I used a GLM with Poisson distribution and log link, which implies the following relationship between the expected outcome and the features str (stress), slp (sleep), and wrk (work). Allow me to be a bit casual with math notation in the following example:

$$\ln(\mathbb{E}[\text{coffee}|\text{str}, \text{slp}, \text{wrk}]) = \beta_0 + \beta_{\text{str}}x_{\text{str}} + \beta_{\text{slp}}x_{\text{slp}} + \beta_{\text{wrk}}x_{\text{wrk}}$$

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

$$\mathbb{E}[\text{coffee}|\text{str}, \text{slp}, \text{wrk}] = \exp(\beta_0 + \beta_{\text{str}}x_{\text{str}} + \beta_{\text{slp}}x_{\text{slp}} + \beta_{\text{wrk}}x_{\text{wrk}})$$

Table 8.1: Results of the Poisson model: weight,  $\exp(\text{weight})$ , and 95% confidence intervals.

	weight	$\exp(\text{weight})$ [2.5%, 97.5%]
(Intercept)	0.03	1.03 [0.65, 1.59]
stress	0.11	1.12 [1.06, 1.18]
sleep	-0.23	0.80 [0.76, 0.84]
workYES	0.98	2.66 [2.03, 3.53]

Since all the weights are in the exponential function, the effect interpretation is not additive, but multiplicative, because  $\exp(a + b)$  is  $\exp(a)$  times  $\exp(b)$ . The last ingredient for the interpretation is the actual weights of the toy example. Table 8.1 lists the estimated weights and  $\exp(\text{weights})$  together with the 95% confidence interval:

Increasing the stress level by one point multiplies the expected number of coffees by the factor 1.12. Increasing the sleep quality by one point multiplies the expected number of coffees by the factor 0.8. The predicted number of coffees on a workday is on average 2.66 times the number of coffees on a day off. In summary, the more stress, the less sleep, and the more work, the more coffee is consumed.

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

## 8.2 Interactions

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

How can we get the linear model to include interactions? Before you fit the linear model, add a column to the feature matrix  $\mathbf{X}$  that represents the interaction between the features and fit the model as usual. The solution is elegant in a way, since it does not require any change of the linear model, only additional columns in the data. In the work day and temperature example, we would add a new feature that has zeros for no-work days; otherwise, it has the value of the temperature feature, assuming that work day is the reference category. Suppose our data looks like in Table 8.2.

Table 8.2: Workday and temperature features for 4 days.

work	temp
Y	25
N	12
N	30
Y	5

Table 8.3: The corresponding data for Table 8.2 for the linear regression model.

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

The data matrix used by the linear model looks slightly different. Table 8.3 shows what the data prepared for the model looks like if we do not specify any interactions. Normally, this transformation is performed automatically by any statistical software.

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

If we want the linear model to consider the interaction between temperature and the workday feature, we have to add a column for the interaction as in Table 8.4.

The new column “workY.temp” captures the interaction between the features work day (work) and temperature (temp). This new feature column is zero for an instance if the work feature is at the reference category (“N” for no work day); otherwise, it assumes the values of the instance’s temperature feature. With this type of encoding, the linear model can learn a different linear effect of temperature for both types of days. This is the interaction effect between the two features. Without an interaction term, the combined effect of a categorical

Table 8.4: Model matrix for the linear regression model when including an interaction between temperature and workday feature.

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

Table 8.5: Data example for bike data with features workday and weather for 4 days.

work	wthr
Y	B
N	G
N	M
Y	B

Table 8.6: Corresponding model matrix for Table 8.5.

Intercept	workY	wthrG	wthrM	workY.wthrG	workY.wthrM
1	1	0	0	0	0
1	0	1	0	0	0
1	0	0	1	0	0
1	1	0	0	0	0

and a numerical feature can be described by a line that is vertically shifted for the different categories. If we include the interaction, we allow the effect of the numerical features (the slope) to have a different value in each category.

The interaction of two categorical features works similarly. We create additional features which represent combinations of categories. Table 8.5 shows some artificial data containing work day (work) and a categorical weather feature (wthr). I'm shortening the weather categories here to narrow the table a bit: G=GOOD, B=BAD, and M=misty.

Next, we include interaction terms to get Table 8.6. The first column serves to estimate the intercept. The second column is the encoded work feature. Columns three and four are for the weather feature, which requires two columns because you need two weights to capture the effect for three categories, one of which is the reference category. The rest of the columns capture the interactions. For each category of both features (except for the reference categories), we create a new feature column that is 1 if both features have a certain category, otherwise 0.

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

There are approaches to automatically detect and add interaction terms. One of them can be found in the [RuleFit chapter](#). The RuleFit algorithm first mines interaction terms and then estimates a linear regression model including interactions. Another option is GA2M (Caruana et al. 2015).

### Example

Let's return to the [bike rental prediction task](#) which we have already modeled in the [linear model chapter](#). This time, we additionally consider an interaction between the temperature

Table 8.7: Linear regression estimates for bike data including interaction between workday and temperature.

	Weight	Std. Error	2.5%	97.5%
(Intercept)	2385.1	355.4	1686.8	3083.5
seasonSPRING	433.2	168.5	102.1	764.2
seasonSUMMER	239.3	216.9	-186.9	665.5
seasonFALL	618.0	151.5	320.4	915.6
holidayY	-434.7	236.9	-900.1	30.8
workdayY	776.7	200.6	382.5	1171.0
weatherMISTY	-374.9	119.5	-609.8	-140.0
weatherBAD	-1802.3	303.2	-2398.2	-1206.4
temp	74.5	12.7	49.5	99.5
hum	-21.7	4.5	-30.6	-12.9
windspeed	-45.5	9.4	-64.0	-27.0
cnt_2d_bfr	0.6	0.0	0.5	0.6
workdayY:temp	-34.4	11.2	-56.4	-12.5

and the work day feature. This results in `tbl-example-lm-interaction` estimated weights and confidence intervals, shown in Table 8.7.

The additional interaction effect is negative (-34.4) and differs significantly from zero, as shown by the 95% confidence interval, which does not include zero. By the way, it's debatable whether the data is IID, because days that are close to each other are not independent of each other. Conditioning on previous counts should alleviate the problem somewhat, at least though. Confidence intervals might be misleading; just take it with a grain of salt. The interaction term changes the interpretation of the weights of the involved features. Does the temperature have a negative effect given it's a workday? The answer is no, even if the table suggests it to an untrained user.

### 💡 Include main effects

Make sure to include both main effects and the interaction effect in a model. This helps prevent misleading interpretations.

We cannot interpret the “workdayY:temp” interaction weight in isolation, since the interpretation would be: “While leaving all other feature values unchanged, increasing the interaction effect of temperature for workday decreases the predicted number of bikes.” But the interaction effect only adds to the main effect of the temperature. Suppose it is a workday and we want to know what would happen if the temperature were 1 degree warmer today. Then we need to sum both the weights for “temp” and “workdayY:temp” to determine how much the

estimate increases.

It's easier to understand the interaction visually, see Figure 8.5. By introducing an interaction term between a categorical and a numerical feature, we get two slopes for the temperature instead of one. The temperature slope for days on which people do not have to work ('N') can be read directly from the table (74.5). The temperature slope for days on which people have to work ('Y') is the sum of both temperature weights (74.5 -34.4 = 40.1). The intercept of the 'N'-line at temperature = 0 is determined by the intercept term of the linear model (2385.1). The intercept of the 'Y'-line at temperature = 0 is determined by the intercept term + the effect of work day (2385.1 + 776.7 = 3161.8).

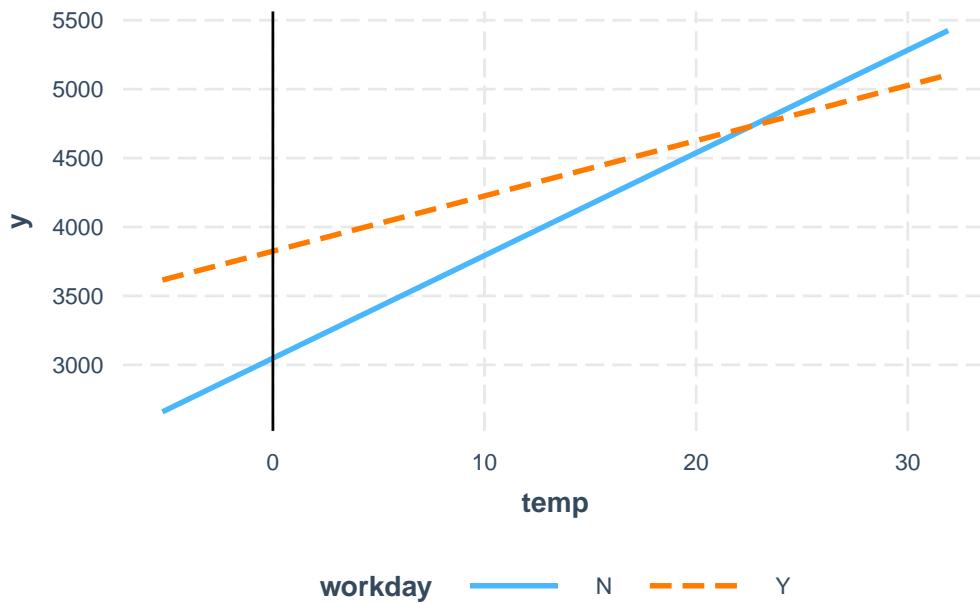


Figure 8.5: Linear model for predicting bike rentals based on temperature, workday, and their interaction. We get two slopes for the temperature, one for each category of the work day feature. The effect of temperature is “flatter” on work days.

### 8.3 Nonlinear effects - GAMs

**The world is not linear.** Linearity in linear models means that no matter what value an instance has in a particular feature, increasing the value by one unit always has the same effect on the predicted outcome. Is it reasonable to assume that increasing the temperature by one degree at 10 degrees Celsius has the same effect on the number of rental bikes as increasing the temperature when it already has 30 degrees? Intuitively, one expects that increasing the temperature from 10 to 11 degrees Celsius has a positive effect on bike rentals and from 30 to

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

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

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

Before I go into the details of each method, let's start with an example that illustrates all three of them. I took the [bike rental dataset](#) and trained a linear model with only the temperature feature to predict the number of rental bikes. Figure 8.6 shows the estimated slope with: the standard linear model, a linear model with transformed temperature (logarithm), a linear model with temperature treated as a categorical feature, and using regression splines (GAM). The linear model (top left) does not fit the data well. One solution is to transform the feature with e.g. the logarithm (top right), categorize it (bottom left), which is usually a bad decision, or use Generalized Additive Models that can automatically fit a smooth curve for temperature (bottom right).

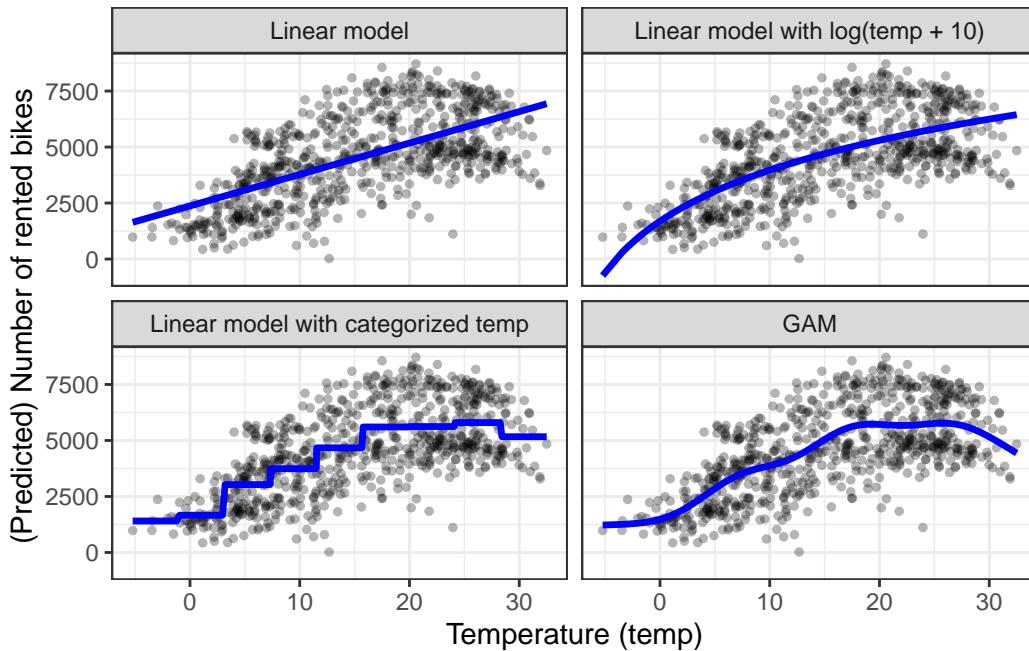


Figure 8.6: Four models predicting bike rentals using only temperature. Each dot is a data point, and the curves represent the model predictions.

## Feature transformation

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

## Feature categorization

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

## Generalized Additive Models (GAMs)

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

$$g[\mathbb{E}(Y|X = \mathbf{x})] = \beta_0 + f_1(x_1) + f_2(x_2) + \dots + f_p(x_p)$$

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

Table 8.8: Feature matrix when using splines.

(Intercept)	s(temp).1	s(temp).2	s(temp).3	s(temp).4
1	1.33	-0.70	-0.39	-1.64
1	1.33	-0.69	-0.37	-1.62
1	1.30	-0.57	-0.25	-1.47
1	1.32	-0.67	-0.35	-1.59
1	1.33	-0.70	-0.39	-1.64
1	1.35	-0.82	-0.53	-1.81

Table 8.9: Weights estimated for the spline features when training the model.

(Intercept)	s(temp).1	s(temp).2	s(temp).3	s(temp).4
4519.6	-922.04	-740.59	2333.45	611.39

The big question is how to learn nonlinear functions. The answer is called “splines” or “spline functions”. Splines are functions that are constructed from simpler basis functions. Splines can be used to approximate other, more complex functions. A bit like stacking Lego bricks to build something more complex. There’s a confusing number of ways to define these spline basis functions. If you are interested in learning more about all the ways to define basis functions, I wish you good luck on your journey. I’m not going to go into details here; I’m just going to build an intuition. What personally helped me the most for understanding splines was to visualize the individual basis functions and to look into how the data matrix is modified. For example, to model the temperature with splines, we remove the temperature feature from the data and replace it with, say, 4 columns, each representing a spline basis function. Usually, you would have more spline basis functions; I only reduced the number for illustration purposes. The value for each instance of these new spline basis features depends on the instances’ temperature values. Together with all linear effects, the GAM then also estimates these spline weights. GAMs also introduce a penalty term for the weights to keep them close to zero. This effectively reduces the flexibility of the splines and reduces overfitting. A smoothness parameter that is commonly used to control the flexibility of the curve is then tuned via cross-validation. Ignoring the optimization with the penalty term, nonlinear modeling with splines looks like fancy feature engineering.

In the example where we are predicting the number of bikes with a GAM using only the temperature, the model feature matrix looks like in Table 8.8. Each row represents an individual instance from the data (one day). Each spline basis column contains the value of the spline basis function at the particular temperature values.

Figure 8.7 shows what these spline basis functions look like.

The GAM assigns weights to each temperature spline basis feature:

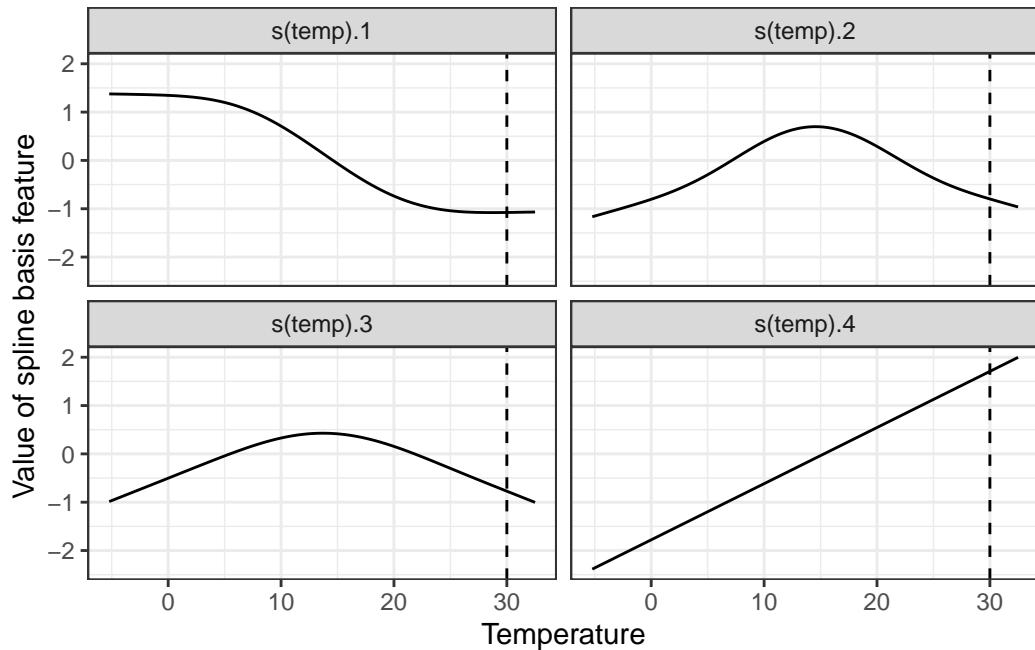


Figure 8.7: Four spline functions for temperature. Each temperature value is mapped to (here) 4 spline basis values. If an instance has a temperature of 30 °C, the value for the first spline basis feature is -1, for the second -0.7, for the third -0.8 and for the fourth 1.7.

And the actual spline curve, which results from the sum of the spline basis functions weighted with the estimated weights, looks like in Figure 8.8. The interpretation of smooth effects requires a visual check of the fitted curve. Splines are usually centered around the mean prediction, so a point on the curve is the difference to the mean prediction. For example, at 0 degrees Celsius, the predicted number of bikes is 3,000 lower than the average prediction.

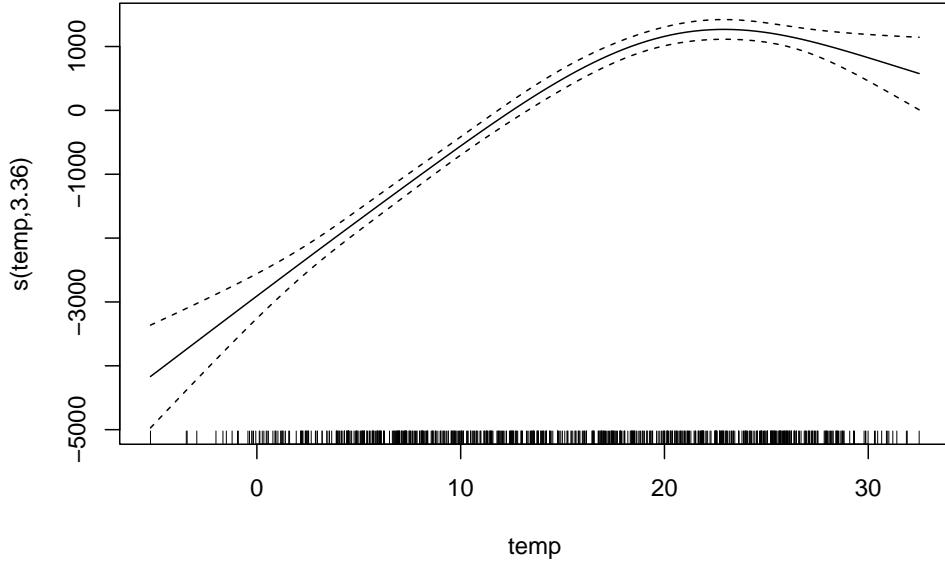


Figure 8.8: GAM feature effect of the temperature for predicting the number of rented bikes (temperature used as the only feature).

## 8.4 Strengths

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

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

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

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

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

## 8.5 Limitations

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

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

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

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

### 💡 Use model-agnostic methods

The more you move away from the pure linear regression by using transformations, interactions, and smooth effects, the more you may need model-agnostic tools like the [partial dependence plot](#) to analyze the model.

## 8.6 Software

All examples in this chapter were created using the R language. For GAMs, the `gam` package was used, but there are many others. R has an incredible number of packages to extend linear regression models. Unsurpassed by any other analytics language, R is home to every

conceivable extension of the linear regression model. You'll find implementations of e.g. GAMs in Python (such as [pyGAM](#)), but these implementations are not as mature. The [Python PiML](#) package also implements various GAM versions.

## 8.7 Further extensions

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

- My data violates the assumption of being independent and identically distributed (iid). For example, repeated measurements on the same patient. Search for **mixed models** or **generalized estimating equations**.
- My model has heteroscedastic errors. For example, when predicting the value of a house, the model errors are usually higher in expensive houses, which violates the homoscedasticity of the linear model. Search for **robust regression**.
- I have outliers that strongly influence my model. Search for **robust regression**.
- I want to predict the time until an event occurs. Time-to-event data usually comes with censored measurements, which means that for some instances there was not enough time to observe the event. For example, a company wants to predict the failure of its ice machines, but only has data for two years. Some machines are still intact after two years, but might fail later. Search for **parametric survival models**, **cox regression**, **survival analysis**.
- My outcome to predict is a category. If the outcome has two categories use a [logistic regression model](#), which models the probability for the categories. If you have more categories, search for **multinomial regression**. Logistic regression and multinomial regression are both GLMs.
- I want to predict ordered categories.  
For example, school grades. Search for **proportional odds model**.
- My outcome is a count (like the number of children in a family). Search for **Poisson regression**. The Poisson model is also a GLM. You might also have the problem that the count value of 0 is very frequent. Search for **zero-inflated Poisson regression**, **hurdle model**.
- I'm not sure what features need to be included in the model to draw correct causal conclusions. For example, I want to know the effect of a drug on the blood pressure. The drug has a direct effect on some blood value and this blood value affects the outcome. Should I include the blood value into the regression model? Search for **causal inference**, **mediation analysis**.
- I have missing data. Search for **multiple imputation**.
- I want to integrate prior knowledge into my models. Search for **Bayesian inference**.
- I'm feeling a bit down lately. Search for “**Amazon Alexa Gone Wild!!! Full version from beginning to end**”.

# 9 Decision Tree

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

There are various algorithms that can grow a tree. They differ in the possible structure of the tree (e.g., number of splits per node), the criteria for how to find the splits, when to stop splitting, and how to estimate the simple models within the leaf nodes. The classification and regression trees (CART) algorithm is probably the most popular algorithm for tree induction. An example is visualized in Figure 9.1. We'll focus on CART, but the interpretation is similar for most other tree types. I recommend the book ‘The Elements of Statistical Learning’ (Hastie 2009) for a more detailed introduction to CART.

The following formula describes the relationship between the outcome  $y$  and features  $\mathbf{x}$ .

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

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

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

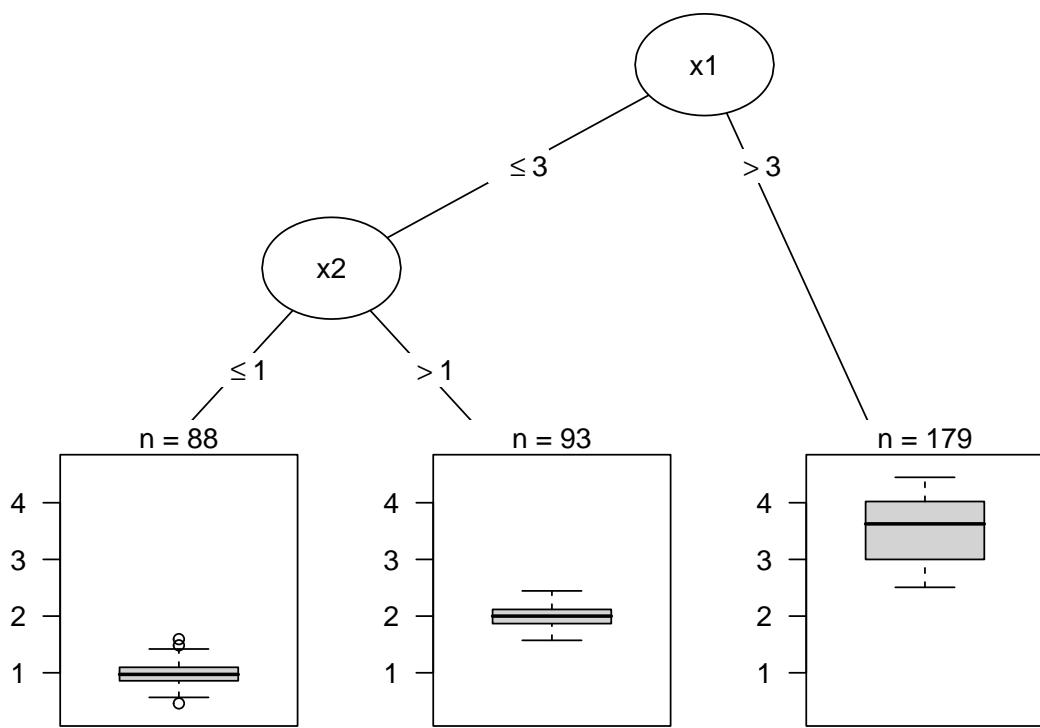


Figure 9.1: Decision tree with artificial data. Instances with a value greater than 3 for feature  $x_1$  end up in node 5. All other instances are assigned to node 3 or node 4, depending on whether values of feature  $x_2$  exceed 1.

that would result in the best partition in terms of the variance or Gini index, and adds this split to the tree. The algorithm continues this search-and-split recursively in both new nodes until a stop criterion is reached. Possible criteria are: A minimum number of instances that have to be in a node before the split, or the minimum number of instances that have to be in a terminal node.

## 9.1 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’.

### Interpretation Template

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

### Feature importance

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

### Tree decomposition

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

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

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

The prediction of an individual instance is the mean of the target outcome plus the sum of all contributions of the  $D$  splits that occur between the root node and the terminal node where the instance ends up. We are not interested in the split contributions, though, but in the

feature contributions. A feature might be used for more than one split or not at all. We can add the contributions for each of the  $p$  features and get an interpretation of how much each feature has contributed to a prediction.

## 9.2 Example

Let's have another look at the [bike rental data](#). We want to predict the number of rented bikes on a certain day with a decision tree. The learned tree is shown in Figure 9.2. I set the maximum allowed depth for the tree to 2. The recent rental count (`cnt_2d_bfr`) and the temperature (`temp`) have been selected for the splits.

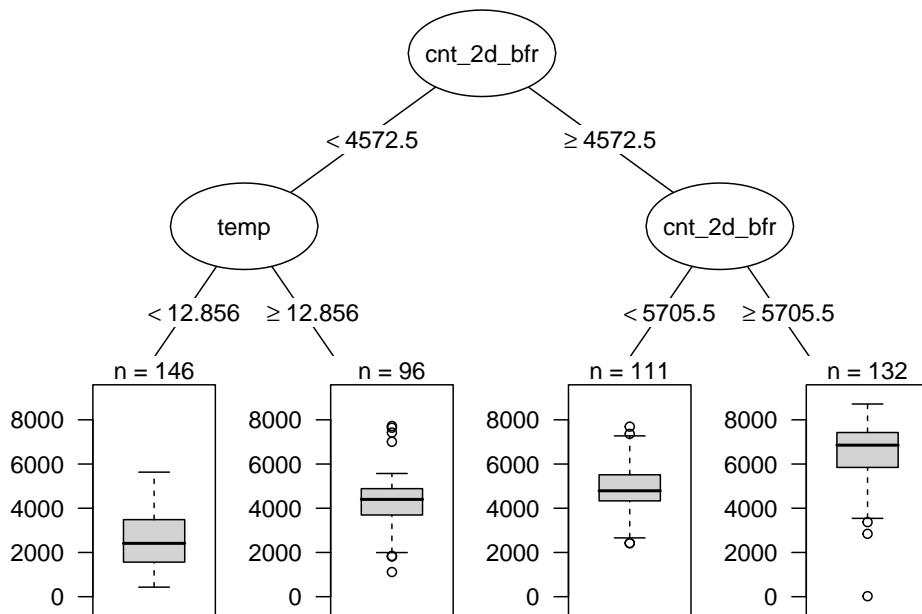


Figure 9.2: Regression tree fitted on the bike rental data. The boxplots show the distribution of bike counts in the terminal node.

Decrease depth and number of nodes

Trees with lower depths and fewer nodes are easier to interpret. Many hyperparameters directly or indirectly control tree depth and/or number of nodes. Some of them interact with the data size, so make sure to pick hyperparameters that control the number of nodes and tree depth directly if you have specific interpretability requirements.

The first split and one of the second level splits were performed with the previous count feature. Basically, the larger the number of recent bike rentals, the more are predicted. If recently not so many bikes were rented, then the temperature decides whether to predict more or fewer rentals. The visualized tree shows that both temperature and time trend were used for the splits, but does not quantify which feature was more important. Figure 9.3 shows the feature importance if we allow the tree to grow deeper. The most important feature, according to Gini importance, was the previous bike count (`cnt_2d_bfr`), followed by temperature and season. Weather was the least important.

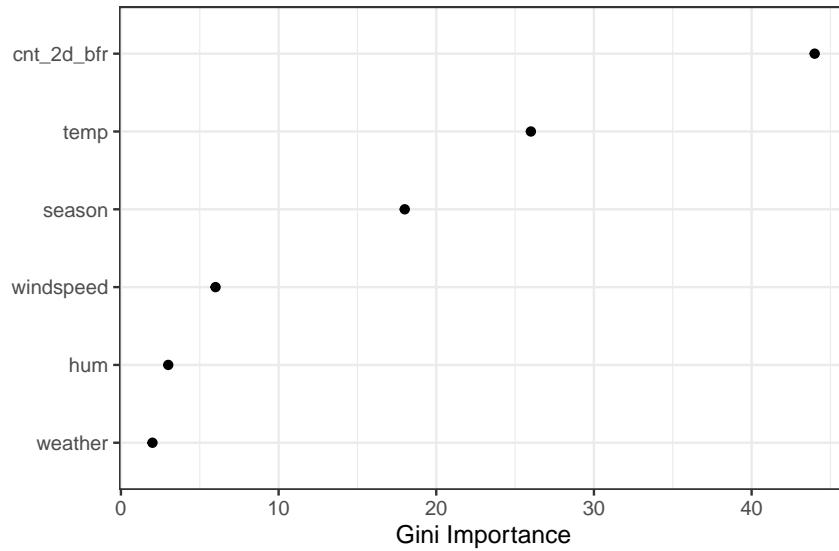


Figure 9.3: Importance of the bike features measured by how much the node purity is improved on average.

### ⚠️ Warning

Gini importance has a bias that it gives higher importance to numerical features and categorical features with many categories (Strobl et al. 2008). Better use [permutation feature importance](#).

## 9.3 Strengths

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

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

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

Trees **create good explanations** as defined in the [chapter on “Human-Friendly Explanations”](#). The tree structure automatically invites to think about predicted values for individual instances as counterfactuals: “If a feature had been greater / smaller than the split point, the prediction would have been  $y_1$  instead of  $y_2$ .” The tree explanations are contrastive, since you can always compare the prediction of an instance with relevant “what if”-scenarios (as defined by the tree) that are simply the other leaf nodes of the tree. If the tree is short, like one to three splits deep, the resulting explanations are selective. A tree with a depth of three requires a maximum of three features and split points to create the explanation for the prediction of an individual instance. The truthfulness of the prediction depends on the predictive performance of the tree. The explanations for short trees are very simple and general because for each split the instance falls into either one or the other leaf, and binary decisions are easy to understand.

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

 Feature scale doesn't matter

CART and other tree algorithms are invariant to the scale and monotonic transformations of the feature. For example, changing a feature from kilograms to grams (multiplying by 1000) will change the split value, but the overall tree structure will be unchanged.

## 9.4 Limitations

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

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

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

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

## 9.5 Software

For the examples in this chapter, I used the `rpart` R package that implements CART (classification and regression trees). CART is implemented in many programming languages, including [Python](#). Arguably, CART is a pretty old and somewhat outdated algorithm, and there are some interesting new algorithms for fitting trees. You can find an overview of some R packages for decision trees in the [Machine Learning and Statistical Learning CRAN Task View](#) under the keyword “Recursive Partitioning”. In Python, the [imodels package](#) provides various algorithms for growing decision trees (e.g., greedy vs. optimal fitting), pruning trees, and regularizing trees.

# 10 Decision Rules

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

Decision rules follow a general structure: IF the conditions are met THEN make a certain prediction. Decision rules are probably the most interpretable prediction models: Their IF-THEN structure semantically resembles natural language and the way we think, provided that the condition is built from intelligible features, the length of the condition is short (small number of `feature=value` pairs combined with an AND) and there are not too many rules. In programming, it'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`, `ok` or `high`). One decision rule learned by this model could be: If a house is bigger than 100 square meters and has a garden, then its value is high. More formally: IF `size>100 AND garden=1` THEN `value=high`.

Let'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 `value=high`.

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

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

**Support or coverage of a rule:** The percentage of instances to which the condition of a rule applies is called the support. Take for example the rule `size=big AND location=good THEN value=high` for predicting house values. Suppose 100 of 1,000 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 AND location=good THEN value=high` applies, 85 have `value=high`, 14 have `value=ok` and 1 has `value=low`, then the accuracy of the rule is 85%.

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

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

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

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

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

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

Both decision lists and sets can suffer from the problem that no rule applies to an instance. This can be resolved by introducing a default rule. The default rule is the rule that applies when no other rule applies. The prediction of the default rule is often the most frequent class of the data points that 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.

## 10.1 Learn rules from a single feature (OneR)

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

Despite the name OneR, which stands for “One Rule,” the algorithm generates more than one rule: It’s actually one rule per unique feature value of the selected best feature. A more fitting name would be OneFeatureRule.

The algorithm is simple and fast:

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

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

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

 Use OneR as baseline

OneR makes for a great baseline to compare more complex models against.

Let's look at an example of how the best feature is chosen by OneR. Table 10.1 shows an artificial dataset about houses with information about their value, location, size, and whether pets are allowed. We are interested in learning a simple model to predict the value of a house.

Table 10.1: Artificial dataset for house value.

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

Table 10.2: Cross tables between each feature and the target.

	value=low	value=ok	value=high
size=big	0	0	2
size=medium	1	3	0
size=small	2	1	1
pets=no	1	1	2
pets=only cats	0	2	0
pets=yes	2	1	1
location=bad	3	2	0
location=good	0	2	3

OneR constructs rules based on the cross-tables between each feature and the outcome. Table 10.2 shows the three cross tables.

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 an `ok` value. The predicted value of houses in good locations is `high`, and again we make two mistakes, because two houses have an `ok` value. The error we make by using the location feature is  $\frac{4}{10}$ , for the size feature it is  $\frac{3}{10}$ , and for the pet feature it is  $\frac{4}{10}$ . The size feature produces the rules with the lowest error and will be used for the final OneR model:

Table 10.3: Rule learned by OneR for penguin classification.

bill_depth_mm	prediction
(13.1,14.7]	female
(14.7,16.3]	male
(16.3,18]	female
(18,19.6]	male
(19.6,21.2]	male

```

IF size=small THEN value=low
IF size=medium THEN value=ok
IF size=big THEN value=high

```

If not properly validated, OneR may overfit on features with many possibly levels. 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.

### ⚠️ Use validation data

Always validate your chosen model with a separate validation set to ensure that it performs well on unseen data, even if it's a simple “one-rule” model.

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

### Example

Let's try OneR with real data. We use the [Palmer penguins data](#) to test the OneR algorithm. All continuous input features were discretized into 5 quantiles. The rules created by the OneR learning algorithms are shown in Table 10.3.

The `bill_depth_mm` feature was chosen by OneR as the best predictive feature. To get a better feel for how this one rule performs, let's look into the cross table between the (discretized) `bill_depth_mm` feature and the penguin species, see Table 10.4.

We can see that this feature is particularly useful for classifying penguins with deep bills. Penguins with bills with a depth of over 1.98 cm are most likely to be male. Very short bills make it likely that we are dealing with a female penguin. Caveat: This model ignores

Table 10.4: Confusion matrix for penguin sex classification rule found by OneR.

bill_depth_mm	female	male
[13.1,14.7)	86.7% (13)	13.3% (2)
[14.7,16.3)	38.5% (10)	61.5% (16)
[16.3,18)	72.0% (18)	28.0% (7)
[18,19.6)	34.4% (11)	65.6% (21)
[19.6,21.2)	0% (0)	100.0% (12)

Table 10.5: OneR classification rule for the bike data.

cnt_2d_bfr	prediction
(13.3,1.76e+03]	[22,3193]
(1.76e+03,3.5e+03]	[22,3193]
(3.5e+03,5.24e+03]	(4551,5978.5]
(5.24e+03,6.98e+03]	(5978.5,8714]
(6.98e+03,8.72e+03]	(5978.5,8714]

the species, or rather bundles them all together. To fix this, we could train one model per species.

OneR doesn't inherently support regression tasks. But we can turn a regression task into a classification task by cutting the continuous outcome into intervals. We use this trick to predict the number of [rented bikes](#) with OneR by cutting the number of bikes into its four quartiles (0-25%, 25-50%, 50-75%, and 75-100%). Table 10.5 shows the selected feature after fitting the OneR model: The selected feature is the previous bike rental count.

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.

## 10.2 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 part of the sequential covering family. 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 that are covered by the rule. A data point is covered when the conditions apply, regardless of whether the points are classified correctly or not. Repeat the rule-learning and removal of covered points with the remaining points until no more points are left, or another

stop condition is met. The result is a decision list. This approach of repeated rule-learning and removal of covered data points is called “separate-and-conquer.”

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

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

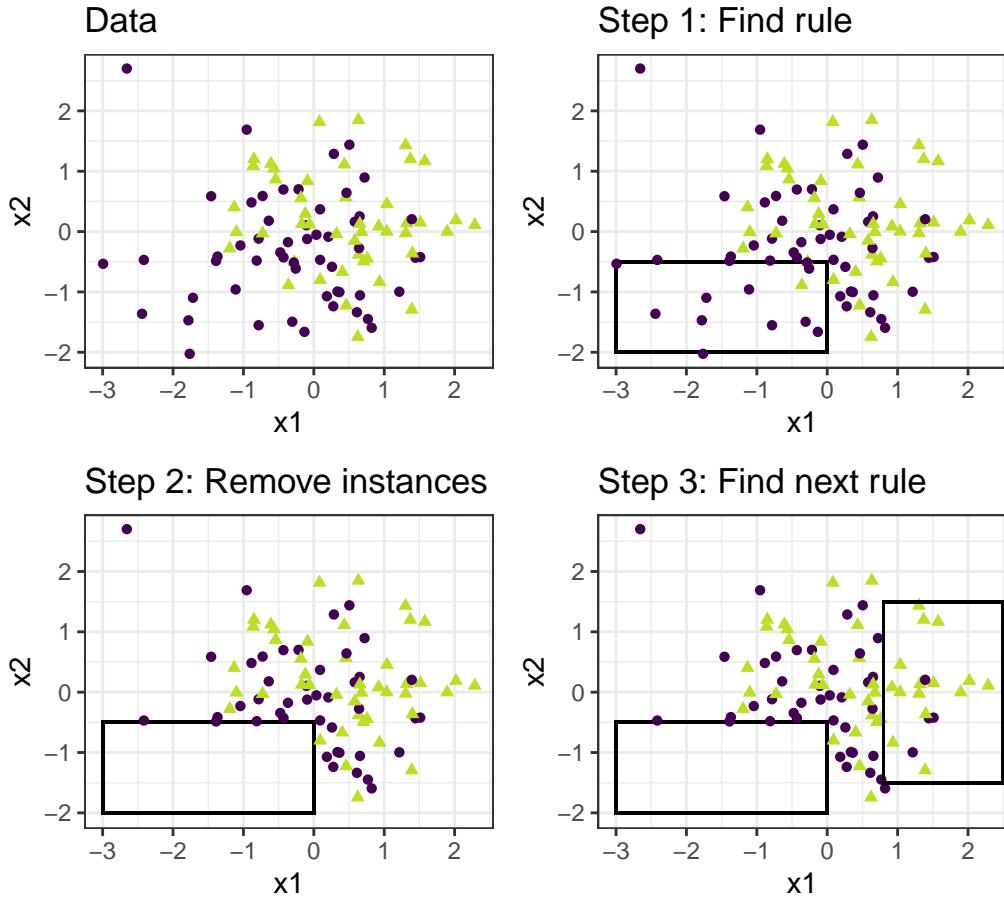


Figure 10.1: Steps of the covering algorithm.

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=ok`. Note that this rule is learned on data without big houses in good locations, leaving only medium and small houses in good locations.

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

How do we learn a single rule? The OneR algorithm wouldn't work here since it would always cover the whole feature space. But there are many other possibilities. One possibility is to learn a single rule from a decision tree with beam search:

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

Figure 10.2 illustrates the beam search in a tree:

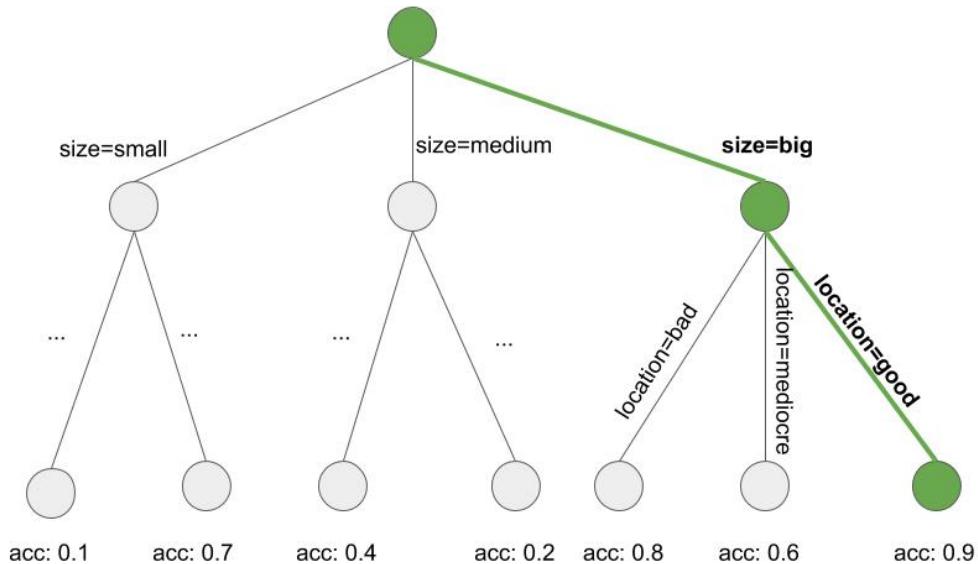


Figure 10.2: Rule learned by searching a path through a decision tree. We end up with: If `location=good` and `size=big`, then `value=high`.

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

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

Table 10.6: JRip rules learned for the penguin classification task.

rules
(body_mass_g <= 3700) and (bill_depth_mm <= 18.5) => sex=F
(bill_depth_mm <= 14.8) and (body_mass_g <= 5200) => sex=F
(body_mass_g <= 3850) and (bill_length_mm <= 36.9) => sex=F
=> sex=M

### Examples

First, we extract the rules for the [penguin sex classification task](#). The rules plus the default rule are printed in Table 10.6. The interpretation is simple: To predict a new instance, start at the top of the list and check whether a rule applies. If the conditions apply, predict the species on the right-hand side of the rule. If it doesn't apply, go to the next rule until you have a classification. The default rule ensures that there is always a prediction.

RIPPER has a specific way of resolving conflicts between rules: it applies them sequentially. This is just one of many general strategies, ranging from rules with the most-specific if-clause to using the rule with the highest precision.

When we use RIPPER on the regression task to predict [bike counts](#), some rules are found. Since RIPPER only works for classification, the bike counts must be turned into a categorical outcome. I achieved this by cutting the bike counts into the quartiles. For example, the interval ([4548, 5956]) covers predicted bike counts between 4548 and 5956. Table 10.7 shows the decision list of learned rules. Interpretation is again: If the conditions apply, we predict the interval on the right-hand side for the number of bikes. I don't like the decision rules for this example. The nature of the `cnt_2d_bfr` feature is to adjust for a general trend of bike rentals. `cnt` and `cnt_2d_bfr` have a linear relationships which decision rules have a hard time to reflect.

## 10.3 Bayesian Rule Lists

In this section, I'll 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) 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).

Table 10.7: JRip rules for the bike task.

rules
(cnt_2d_bfr <= 5729) and (cnt_2d_bfr >= 4780) and (temp <= 26) => cnt=(4551,5978.5]
(temp >= 14) and (cnt_2d_bfr <= 5515) and (hum <= 63) and (cnt_2d_bfr >= 3574) and (temp <= 27) => cnt=(4551,5978.5]
(cnt_2d_bfr >= 3544) and (cnt_2d_bfr <= 3915) and (temp >= 17) and (temp <= 28) => cnt=(4551,5978.5]
(cnt_2d_bfr <= 5336) and (cnt_2d_bfr >= 2914) and (weather = GOOD) and (temp >= 7) => cnt=(3193,4551]
(cnt_2d_bfr <= 4833) and (cnt_2d_bfr >= 2169) and (hum <= 72) and (workday = Y) => cnt=(3193,4551]
(cnt_2d_bfr <= 4097) and (season = WINTER) => cnt=[22,3193]
(cnt_2d_bfr <= 4570) and (temp <= 13) => cnt=[22,3193]
(hum >= 88) and (season = FALL) => cnt=[22,3193]
(cnt_2d_bfr <= 3351) and (cnt_2d_bfr >= 2710) => cnt=[22,3193]
=> cnt=(5978.5,8714]

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

### Pre-mining of frequent patterns

A frequent pattern is the frequent (co-)occurrence of feature values. As a pre-processing step for the BRL algorithm, we use the features (we don't need the target outcome in this step) and extract frequently occurring patterns from them. A pattern can be a single feature value such as `size=medium` or a combination of feature values such as `size=medium AND location=bad`.

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

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

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

There are many algorithms to find such frequent patterns, for example, Apriori or FP-Growth. Which you use 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=small` as patterns. This does not mean that the houses are removed from the data; it just means that `size=big` is not returned as a frequent pattern. Based on frequent patterns with a single feature value, the Apriori algorithm iteratively tries to find combinations of feature values of increasingly higher order. Patterns are constructed by combining `feature=value` statements with a logical AND, e.g., `size=medium AND location=bad`. Generated patterns with a support below the minimum support are removed. In the end, we have all the frequent patterns. Any subset of a frequent pattern's clauses is frequent again, which is called the Apriori property. It makes sense intuitively: By removing a condition from a pattern, the reduced pattern can only cover more or the same number of data points, but not less. For example, if 20% of the houses are `size=medium AND location=good`, then the support of houses that are only `size=medium` is 20% or greater. The Apriori property is used to reduce the number of patterns to be inspected. Only in the case of frequent patterns do we have to check patterns of higher order.

Now we are done with pre-mining conditions for the Bayesian Rule List algorithm. But before we move on to the second step of BRL, I would like to hint at another way for rule-learning based on pre-mined patterns. Other approaches suggest including the outcome of interest in 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 posterior probability distribution of lists makes it possible to say how likely a decision list is, given assumptions of shortness and how well the list fits the data. Our goal is to find the list that maximizes this posterior probability. Since it is not possible to find the exact best list directly from the distributions of lists, BRL suggests the following recipe: 1) Generate an initial decision list, which is randomly drawn from the prior 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 posterior 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. (That's Bayesian statistics.) If you are unfamiliar with Bayesian statistics, do not 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 fitting to the data. In the case of decision lists, the Bayesian approach makes sense, since the prior assumptions nudge the decision lists to be short with short rules.

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

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

where  $d$  is a decision list,  $\mathbf{x}$  are the features,  $\mathbf{y}$  is the target,  $A$  the set of pre-mined conditions,  $\lambda$  the prior expected length of the decision lists,  $\eta$  the prior expected number of conditions in a rule,  $\alpha$  the prior pseudo-count for the positive and negative classes which is best fixed at (1,1).

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

quantifies how probable a decision list is, given the observed data and the prior 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.

$$p(\mathbf{y}|\mathbf{x}, d, \alpha)$$

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  $\lambda$ ) for the number of rules in the list and a truncated Poisson distribution (parameter  $\eta$ ) for the number of feature values in the conditions of the rules.

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

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

- Pre-mine patterns with FP-Growth.
- Sample the list length parameter  $m$  from a truncated Poisson distribution.
- For the default rule: Sample the Dirichlet-Multinomial distribution parameter  $\theta_0$  of the target value (i.e., the rule that applies when nothing else applies).
- For decision list rule  $j = 1, \dots, m$ , do:
  - Sample the rule length parameter  $l$  (number of conditions) for rule  $j$ .
  - Sample a condition of length  $l_j$  from the pre-mined conditions.
  - Sample the Dirichlet-Multinomial distribution parameter for the THEN-part (i.e., for the distribution of the target outcome given the rule).
- For each observation in the dataset:
  - Find the rule from the decision list that applies first (top to bottom).
  - Draw the predicted outcome from the probability distribution (Binomial) suggested by the rule that applies.

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

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

## Examples

That'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) (H. Yang, Rudin, and Seltzer 2017). We use the SBRL algorithm to predict the [species of the penguins](#). I had to discretize all input

Table 10.8: Rules from SBRL model.

rules
If {body_mass_g=[5.1e+03,6.3e+03]} then p = 0.083
else if {species=Gentoo} then p = 0.873
else if {body_mass_g=[3.9e+03,5.1e+03]} then p = 0.066
else if {bill_depth_mm=[15.9,18.7)} then p = 0.876
else (default rule) then p = 0.333

Table 10.9: Premined rules for the penguin classification task.

pre-mined conditions
species=Gentoo, bill_depth_mm=[13.1, 15.9)
body_mass_g=[2.7e+03, 3.9e+03)
bill_depth_mm=[13.1, 15.9), bill_length_mm=[41.3, 50.4)
bill_length_mm=[41.3, 50.4)
flipper_length_mm=[172, 192), body_mass_g=[3.9e+03, 5.1e+03)
bill_depth_mm=[15.9, 18.7), flipper_length_mm=[211, 231]
species=Adelie
species=Adelie, bill_depth_mm=[15.9, 18.7)
bill_depth_mm=[18.7, 21.5], flipper_length_mm=[172, 192)
species=Adelie, bill_length_mm=[41.3, 50.4)

features for the SBRL algorithm to work. To do this, I binned the continuous features based on the frequency of the values by quantiles. We get the rules displayed in Table 10.8.

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. Table 10.9 shows a sample of ten patterns.

 Sparsity is king

Fewer and shorter rules make for better model interpretation. But there is a trade-off with complexity and therefore with predictive performance.

## 10.4 Strengths

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

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

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

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

Decision rules are **robust** against monotonic 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 a baseline** for more complex algorithms.

## 10.5 Limitations

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

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

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

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

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

## 10.6 Software and alternatives

OneR is implemented in the [R package OneR](#), which was used for the examples in this book. OneR is also implemented in the [Weka machine learning library](#) and, as such, is 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](#). SBRL is available as an [R package](#) (which I used for the examples), in [Python](#), or as a [C implementation](#). Additionally, I recommend the [imodels package](#), which implements rule-based models such as Bayesian rule lists, CORELS, OneR, greedy rule lists, and more in a Python package with a unified scikit-learn interface.

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

# 11 RuleFit

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

The linear regression model doesn't account for interactions between features. Wouldn't it be convenient to have a model that is as simple and interpretable as linear models, but also integrates feature interactions? RuleFit fills this gap. RuleFit learns a sparse linear model with the original features and also a number of new features that are decision rules. These new features capture interactions between the original features. RuleFit automatically generates these features from decision trees. Each path through a tree can be transformed into a decision rule by combining the split decisions into a rule, see Figure 11.1. The node predictions are discarded and only the splits are used in the decision rules:

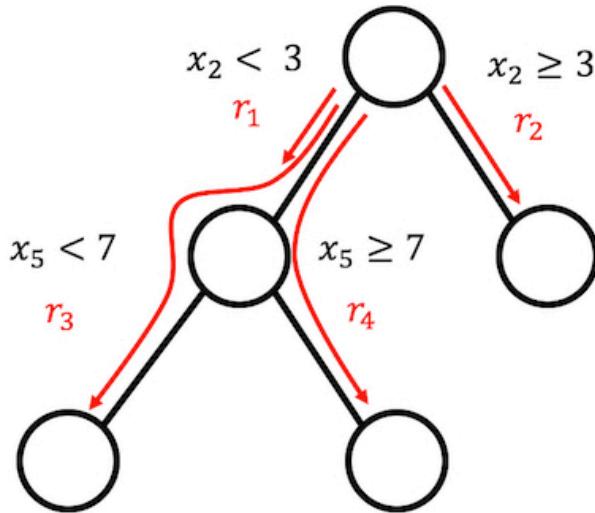


Figure 11.1: 4 rules can be generated from a tree with 3 terminal nodes.

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

Table 11.1: Rules generated by RuleFit along with their weights in the linear model.

Description	Weight	Importance
temp > 12 & hum <= 86	626	311
cnt_2d_bfr > 5382 & hum <= 80	661	299
cnt_2d_bfr > 2703 & weather in ("GOOD")	426	213
cnt_2d_bfr > 5047 & temp > 9	376	178
4 <= windspeed <= 24	-32	161

The RuleFit paper uses the Boston housing data to illustrate this: The goal is to predict the median house value of a Boston neighborhood. One of the rules generated by RuleFit is: IF `number of rooms > 6.64 AND concentration of nitric oxide < 0.67 THEN 1 ELSE 0.`

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

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

## 11.1 Interpretation and example

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

In this example, we use RuleFit to predict the number of [rented bikes](#) on a given day. Table 11.1 shows five of the rules that were generated by RuleFit, along with their Lasso weights and importances. The calculation is explained later in the chapter. The most important rule was: “temp > 12 & hum <= 86,” and the corresponding weight is 626. The interpretation is: If temp > 12 & hum <= 86, then the predicted number of bikes increases by 626, when all other feature values remain fixed. In total, 368 such rules were created from the original 8 features. Quite a lot! But thanks to Lasso, only 33 of the 368 have a weight different from 0.

Computing the global feature importances reveals that temperature and time trend are the most important features, as visualized in Figure 11.2. The feature importance measurement

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

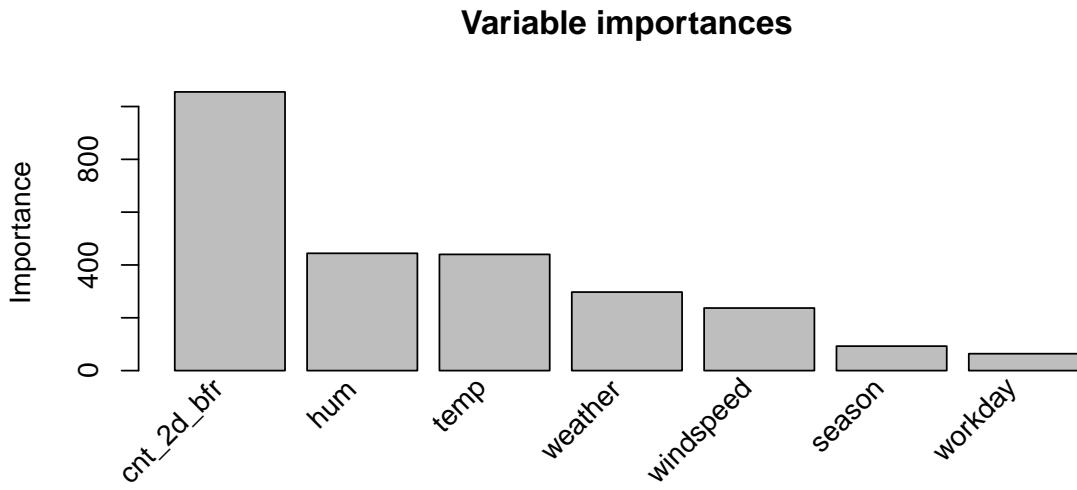


Figure 11.2: Feature importance measures for a RuleFit model predicting bike counts. The most important features for the predictions were the previous count, humidity and temperature.

### Interpretation template

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

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

#### ! Beware of extrapolation

“Given all other features remain the same” is a bit unreasonable here because a feature may both have a linear component and appear in multiple rules. Changing the original feature would then change multiple components, so in fact, not all model features remain the same.

## 11.2 Theory

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

### Step 1: Rule generation

What does a rule look like? The rules generated by the algorithm have a simple form. For example: IF  $x_2 < 3$  AND  $x_5 < 7$  THEN 1 ELSE 0. The rules are constructed by decomposing decision trees: Any path to a node in a tree can be converted to a decision rule. The trees used for the rules are fitted to predict the target outcome. Therefore, the splits and resulting rules are optimized to predict the outcome you are interested in. You simply chain the binary decisions that lead to a certain node with "AND," and voilà, you have a rule. It's 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 outcomes  $\mathbf{y}$  with your original features  $\mathbf{X}$ . Each resulting tree is converted into multiple rules. Not only boosted trees, but any tree ensemble algorithm can be used to generate the trees for RuleFit. A tree ensemble can be described with this general formula:

$$\hat{f}(\mathbf{x}) = a_0 + \sum_{m=1}^M a_m \hat{f}_m(\mathbf{x})$$

$M$  is the number of trees, and  $\hat{f}_m(\mathbf{x})$  is the prediction function of the  $m$ -th tree. The  $a$ 's are the weights. Bagged ensembles, random forest, AdaBoost, and MART produce tree ensembles and can be used for RuleFit.

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

$$r_m(\mathbf{x}) = \prod_{j \in T_m} I(x_j \in s_{jm})$$

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

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

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

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

A made-up example for the bike rental dataset:

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

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

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

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

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

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

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

 Fewer conditions, better interpretability

Keep the number of conditions within each rule to between 1 and 3 for better interpretability.

## Step 2: Sparse linear model

You get MANY rules in step 1. Since the first step can be seen as only a feature transformation, you are still not done with fitting a model. Also, you want to reduce the number of rules. In addition to the rules, all your “raw” features from your original dataset will also be used in the sparse linear model. Every rule and every original feature becomes a feature in the linear

model and gets a weight estimate. The original raw features are added because trees fail at representing simple linear relationships between  $Y$  and  $X_j$ . Before we train a sparse linear model, we winsorize the original features so that they are more robust against outliers:

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

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

$$l_j(x_j) = 0.4 \cdot \frac{l_j^*(x_j)}{\text{std}(l_j^*(x_j))}$$

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

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

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

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

$$\begin{aligned} (\{\hat{\alpha}\}_1^K, \{\hat{\beta}\}_0^p) = & \arg \min_{\{\hat{\alpha}\}_1^K, \{\hat{\beta}\}_0^p} \sum_{i=1}^n L(y^{(i)}, \hat{f}(\mathbf{x}^{(i)})) \\ & + \lambda \cdot \left( \sum_{k=1}^K |\hat{\alpha}_k| + \sum_{j=1}^p |\hat{\beta}_j| \right) \end{aligned}$$

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

### Step 3 (optional): Feature importance

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

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

where  $\hat{\beta}_j$  is the weight from the Lasso model, and  $\text{std}(l_j(x_j))$  is the standard deviation of the linear term over the data.

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

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

where  $\hat{\alpha}_k$  is the associated Lasso weight of the decision rule, and  $s_k$  is the support of the feature in the data, which is the percentage of data points to which the decision rule applies (where  $r_k(\mathbf{x}) = 1$ ):

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

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

$$J_j(x_j) = I_j(x_j) + \sum_{k|x_j \in r_k} \frac{I_k(r_k)}{m_k}$$

where  $I_j$  is the importance of the linear term and  $I_k$  the importance of the decision rules in which  $X_j$  appears, and  $m_k$  is the number of features constituting the rule  $r_k$ . Adding the feature importance from all instances gives us the global feature importance:

$$J_j(\mathbf{x}_j) = \sum_{i=1}^n J_j(x_j^{(i)})$$

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

## 11.3 Strengths

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

RuleFit can handle both classification and regression tasks.

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

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

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

## 11.4 Limitations

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

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

The end product of the RuleFit procedure is a linear model with additional fancy features (the decision rules). But since it is a linear model, the **weight interpretation is still unintuitive**. It comes with the same “footnote” as a usual linear regression model: “... given all features are fixed.” It gets a bit more tricky when you have overlapping rules. For example, one decision rule (feature) for the bike prediction could be: “temp > 10” and another rule could be “temp > 15 & weather=‘GOOD’”. If the weather is good and the temperature is above 15 degrees, the temperature is automatically greater than 10. In the cases where the second rule applies, the first rule applies as well. The interpretation of the estimated weight for the second rule is: “Assuming all other features remain fixed, the predicted number of bikes increases by  $\beta_2$  when the weather is good and temperature above 15 degrees.”. But, now it becomes really clear that

the ‘all other feature fixed’ is problematic, because if rule 2 applies, also rule 1 applies and the interpretation is nonsensical.

## 11.5 Software and alternatives

The RuleFit algorithm is [implemented in R](#) by Fokkema (2020a), and you can find a [Python version on GitHub](#).

A very similar framework is [skope-rules](#), a Python module that also extracts rules from ensembles. It differs in the way it learns the final rules: First, skope-rules remove low-performing rules, based on recall and precision thresholds. Then, duplicate and similar rules are removed by performing a selection based on the diversity of logical terms (variable + larger/smaller operator) and performance (F1-score) of the rules. This final step does not rely on using Lasso but considers only the out-of-bag F1-score and the logical terms that form the rules.

The [imodels package](#) also contains implementations of other rule sets, such as Bayesian rule sets, Boosted rule sets, and SLIPPER rule sets, as a Python package with a unified scikit-learn interface. Then there is model-based boosting (Bühlmann and Hothorn 2007), which also allows mixing linear components and rule components, implemented in the R package [mboost](#).

## **Part II**

# **Local Model-Agnostic Methods**

## 12 Ceteris Paribus Plots

Ceteris paribus (CP) plots (Kuźba, Baranowska, and Biecek 2019) visualize how changes in a single feature change the prediction of a data point.

Ceteris paribus plots are one of the simplest analysis one can do, despite the complex-sounding Latin name, which stands for “other things equal” and means changing one feature but keeping the others untouched.<sup>1</sup> It’s so simple since it only looks at one feature at a time, systematically changes its values, and plots how the prediction changes across the range of the feature. But ceteris paribus is the perfect model-agnostic method to start the book with, since it teaches the basic principles of model-agnostic interpretation. Also, don’t be deceived by the simplicity: By creatively combining multiple ceteris paribus curves<sup>2</sup>, you can compare models, features, and study multi-class classification models. CP curves are building blocks for [Individual Conditional Expectation curves](#) and [Partial Dependence Plots](#), as visualized in Figure 12.1.

- ICE plots are CP plots containing all CP curves for an entire dataset.
- A partial dependence plot (PDP) is the average of all CP curves of one dataset.

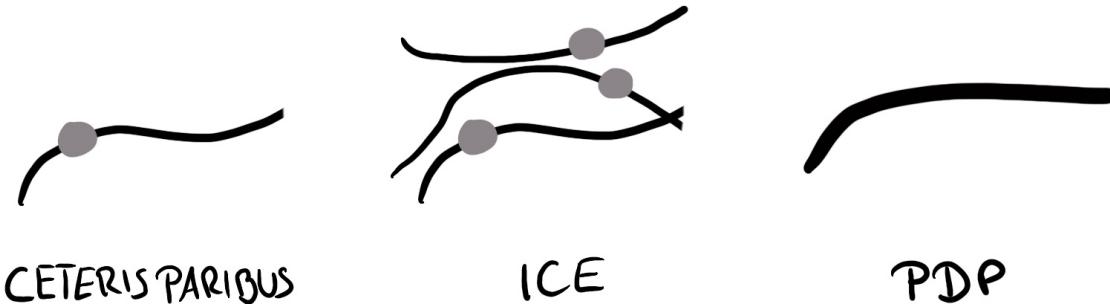


Figure 12.1: Ceteris Paribus, ICE, and PDP.

<sup>1</sup>I’m not a fan of the Latin name, because in school they promised that Latin helps with learning other Latin-based languages, like Italian. But you know what even helps better when your goal is to learn Italian? That’s right. Just learn Italian.

<sup>2</sup>By “CP curve” I mean a single line, and by “CP plot” I mean a graph showing one or more CP curves.

## 12.1 Algorithm

Let's get started with the ceteris paribus algorithm. This is also a little showcase of how things may seem more complex than they are when you use math. The following algorithm is for numerical features:

Input: Data point  $\mathbf{x}^{(i)}$  to explain and feature  $j$

1. Create an equidistant value grid:  $z_1, \dots, z_K$ , where typically  $z_1 = \min(\mathbf{x}_j)$  and  $z_K = \max(\mathbf{x}_j)$ .
2. For each grid value  $z_k \in \{z_1, \dots, z_K\}$ :
  1. Create new data point  $\mathbf{x}_{x_j:=z_k}^{(i)}$
  2. Get prediction  $\hat{f}(\mathbf{x}_{x_j:=z_k}^{(i)})$
3. Visualize the CP curves:
  - Plot line for data points  $\left\{z_l, \hat{f}(\mathbf{x}_{x_j:=z_k}^{(i)})\right\}_{k=1}^K$
  - Plot dot for original data point  $(\mathbf{x}_k^{(i)}, \hat{f}(\mathbf{x}^{(i)}))$

The more grid values, the more fine-grained the CP curve, but the more calls you have to make to the predict function of the model. And for a categorical feature:

1. Create list of unique categories  $z_1, \dots, z_K$
2. For each category  $z_k \in \{z_1, \dots, z_K\}$ :
  1. Create new data point  $\mathbf{x}_{x_j:=z_k}^{(i)}$
  2. Get prediction  $\hat{f}(\mathbf{x}_{x_j:=z_k}^{(i)})$
3. Create bar plot or dot plot with categories on x-axis and predictions on y-axis.

But that sounded more complex than necessary. Let's make CP plots more concrete with a few examples.

## 12.2 Examples

For our first example, we look at the random forest predicting the probability of a penguin being female. We look at the first penguin in the test dataset, a Adelie penguin with a bill depth of 20.6 millimeters (ground truth male). Figure 12.2 shows that decreasing the bill depth of this penguin first slightly increases the predicted P(female), but then greatly decreases P(female).

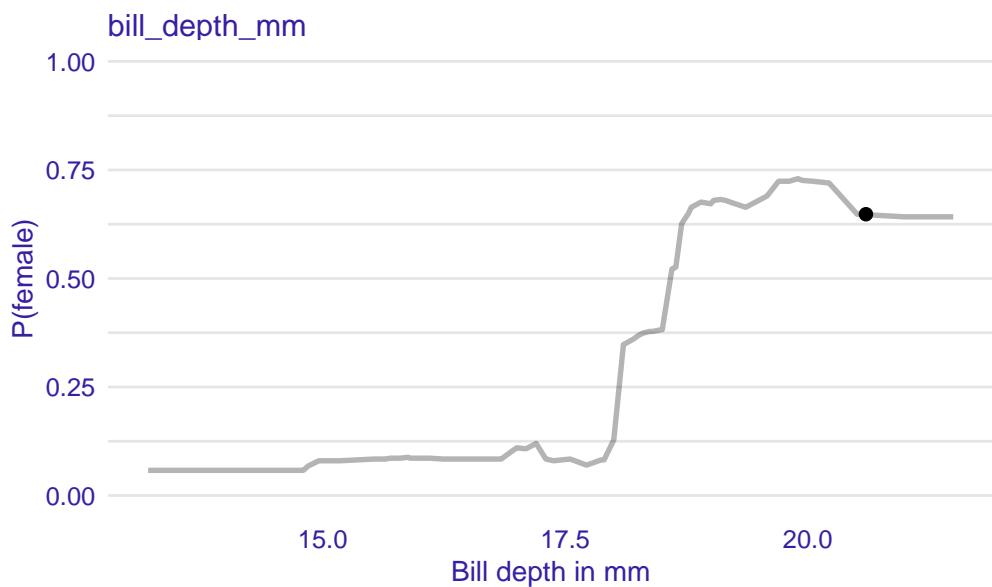


Figure 12.2: Ceteris paribus plot for bill depth and one penguin. The line shows the predicted value for a particular penguin when changing the bill depth. The penguin's actual bill depth is marked with a dot.

Since it's a binary classification task, visualizing  $\mathbb{P}(Y = \text{male})$  is redundant – it would just be the inverted  $\mathbb{P}(Y = \text{female})$  plot. But if we had more than two classes, we could plot the ceteris paribus curves for all the classes in one plot.

Again, we have to keep in mind that changing a feature can break the dependence with other features. Looking at correlation and other dependence measures can be helpful. Bill depth is correlated with body mass and flipper length. So when looking at Figure 12.2, we should keep in mind not to over-interpret strong reductions in bill depth in this ceteris paribus plot.

### ⚠ Look beyond pairwise dependencies

Figure 12.3 shows an example where we artificially change the bill depth feature of the lightest Gentoo penguin. The data point is realistic when we only look at the combination of body mass and bill depth. It's also realistic when we only look at bill depth and species. However, it's unrealistic when considering the new bill depth, body mass, and species together.

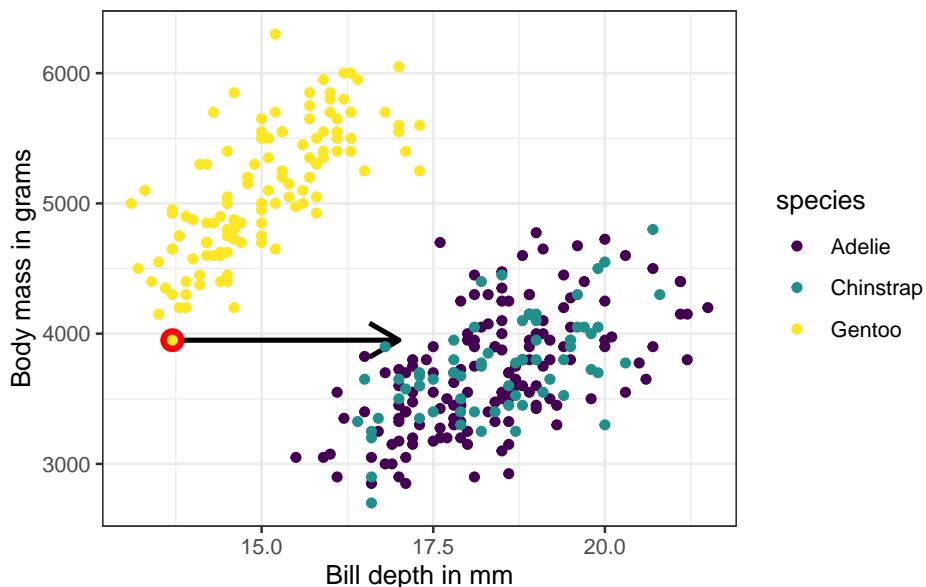


Figure 12.3: Scatter plot of bill depth and body mass. Manipulating a penguins bill depth can create an unrealistic data point.

Next, we study the SVM predicting the number of rented bikes based on weather and seasonal information. We pick a winter day with good weather and see how changing the features would change the prediction. But this time we visualize it for all features, see Figure 12.4. Changes in the number of rented bikes 2 days before would change the prediction the most. Also, a higher temperature would have been better for more bike rentals. Were the prediction for a non-workday, the SVM would predict fewer bike rentals.

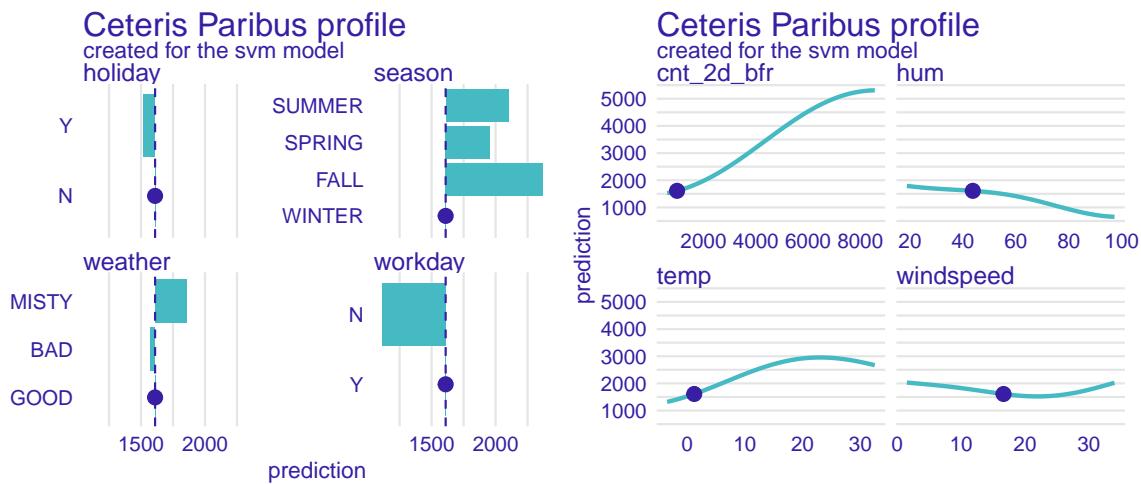
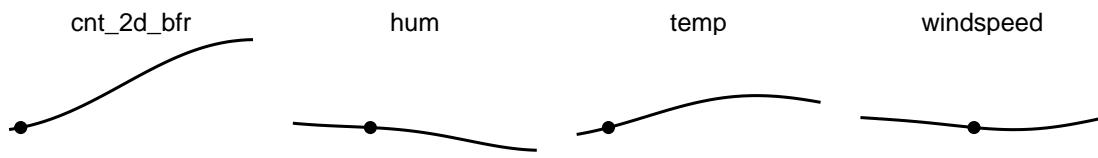


Figure 12.4: How changing individual features changes the predicted number of bike rentals.

#### Minimal version with sparklines

CP plots can be packaged into sparklines, a minimalist line-plot popularized by Tufte and Graves-Morris (1983).



In general, the CP plots show how feature changes affect the prediction, from small changes to large changes in all directions. By comparing all the features side-by-side with a shared y-axis, we can see which feature has more influence on this data point's prediction. However, correlation between features is a concern, especially when interpreting the CP curve far away from the original value (marked with a dot). For example, increasing the temperature to 30 degrees Celsius, but keeping the season the same (winter) would be quite unrealistic.

Ceteris paribus plots are super flexible. We can compute CP curves for different models to better understand how they treat features differently. In Figure 12.5, we compare the ceteris paribus plots for different models.

Here we can see how very different the models behave: The linear model does what linear models do and models the relation linearly between temperature and predicted bike rentals. The tree shows one jump. The random forest and the SVM model show a smoother increase, which flattens at high temperature, and, in the case of the SVM, slightly decreases for very

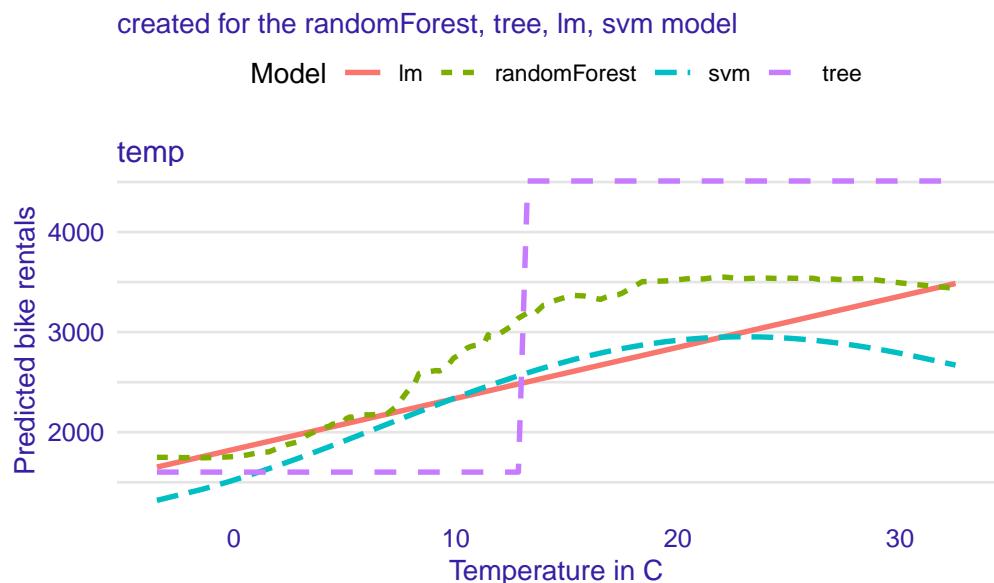


Figure 12.5: Ceteris paribus curves for the bike prediction task for different models: linear model, random forest, SVM, and decision tree.

high temperatures.

 Be creative with comparisons

Ceteris paribus plots are simple, yet surprisingly insightful when you combine multiple CP curves:

- Compare features.
- Compare models from different machine learning algorithms or with different hyperparameter settings.
- Compare class probabilities.
- Compare different data points (see also [ICE curves](#)).
- Subset the data (e.g., by a binary feature) and compare CP curves.

## 12.3 Strengths

**Ceteris paribus plots are super simple to implement and understand.** This makes them a great entry point for beginners, but also for communicating model-agnostic explainability to others, especially non-experts.

**CP plots can fix limitations of attribution methods.** Attribution-based methods like SHAP or LIME don't show how sensitive the prediction function is to local changes. Ceteris paribus plots can complement attribution-based techniques and provide a complete picture when it comes to explaining individual predictions.

**Ceteris paribus plots are flexible building blocks.** They are building blocks for other interpretation methods, but you can also get creative in combining these lines across models, classes, hyperparameter settings, and features to create nuanced insights into the model predictions.

## 12.4 Limitations

**Ceteris paribus plots only show us one feature change at a time.** This means we don't see how two features interact. Of course, you can change two features, especially if one is continuous and the other binary, and plot them in the same CP plot. But it's a more manual process.

**Interpretation suffers when features are correlated.** When features are correlated, not all parts of the curve are likely or might even be completely unrealistic. This can be alleviated by e.g. restricting the range of the ceteris paribus plots to shorter ranges, at least for correlated features. But this would also mean we need a model or procedure to tell us what these ranges are.

In general, you must **be careful with causal interpretation**; or if you want one, make sure the model itself is causal. This is a problem with all interpretation methods, but the risk of wrongful causal interpretation may be higher with CP plots since there is a lower barrier to showing the plots to non-experts.

## 12.5 Software and alternatives

I created all plots in this chapter with the [ceterisParibus R package](#). It also has a [Python implementation](#). You can further create CP plots with any tool that can produce ICE plots, like ICEBox and iml, by simply providing a “dataset” that only contains the one data point you are interested in. However, the ceterisParibus package is better suited because it makes it simpler to compare ceteris paribus curves.

# 13 Individual Conditional Expectation (ICE)

Individual Conditional Expectation (ICE) plots display one line per instance that shows how the instance's prediction changes when a feature changes. An ICE plot (Goldstein et al. 2015) visualizes the dependence of the prediction on a feature for *each* instance separately, resulting in one line per instance of a dataset. The values for a line (and one instance) can be computed by keeping all other features the same, creating variants of this instance by replacing the feature's value with values from a grid, and making predictions with the black box model for these newly created instances. The result is a set of points for an instance with the feature value from the grid and the respective predictions. In other words, ICE plots are all the [ceteris paribus curves](#) for a dataset in one plot.

## 13.1 Examples

Figure 13.1 shows ICE plots for the [bike rental prediction](#). The underlying prediction model is a random forest. All curves seem to follow the same course, so there are no obvious interactions.

But we can also explore possible interactions by modifying the ICE plot. Figure 13.2 shows again the ICE plot for humidity, with the difference that the lines are now colored by the season. This shows a couple of things: First – and that's not surprising – different seasons have different “intercepts”. Meaning that, for example, winter days have a lower prediction and summer the highest ones, independent of the humidity. But Figure 13.2 also shows that the effect of the humidity differs for the seasons: In winter, an increase in humidity only slightly reduces the predicted number of bike rentals. For summer, the predicted bike rentals stay more or less flat between 20% and 60% relative humidity and above 60% they drop by quite a bit. Humidity effects for spring and fall seem to be a mix of the “winter flatness” and the “summer jump”. However, as indicated by the boxplots in Figure 13.2, we shouldn't over-interpret very low humidity effects for summer and fall.

 Use transparency and color

If lines overlap heavily in a boxplot you can try to make them slightly transparent. If that doesn't help, you may be better off with a [partial dependence plot](#). By coloring the lines based on another feature's value, you can study interactions.

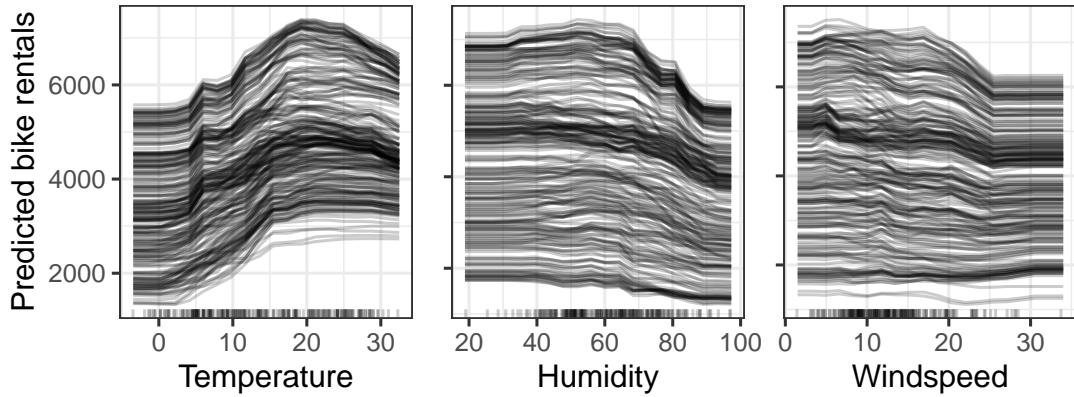


Figure 13.1: ICE plots of predicted bike rentals by temperature, humidity, and windspeed.

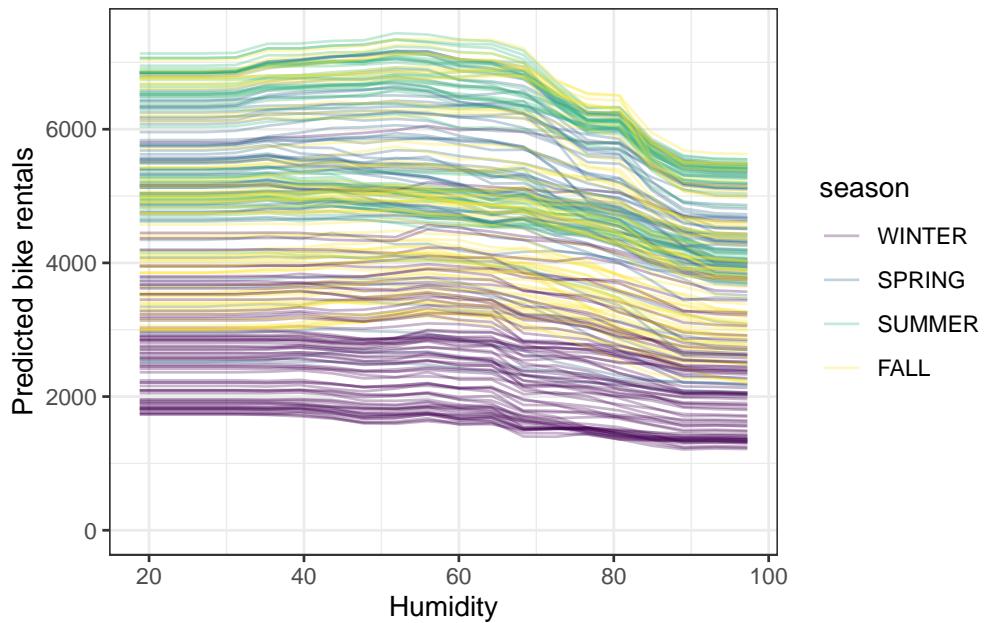


Figure 13.2: ICE curves for the random forest predicting bike rentals. Lines are colored by the season. Above the ICE plots are boxplots showing the distributions of humidity per season.

Let's go back to the [penguin classification task](#) and see how the prediction of each instance is related to the feature `bill_length_mm`. We'll analyze a random forest that predicts the probability of a penguin being female given body measurements. Figure 13.3 is a rather ugly ICE plot. But sometimes that's the reality. The reason is that the model is rather sure for most penguins and jumps between 0 and 1.

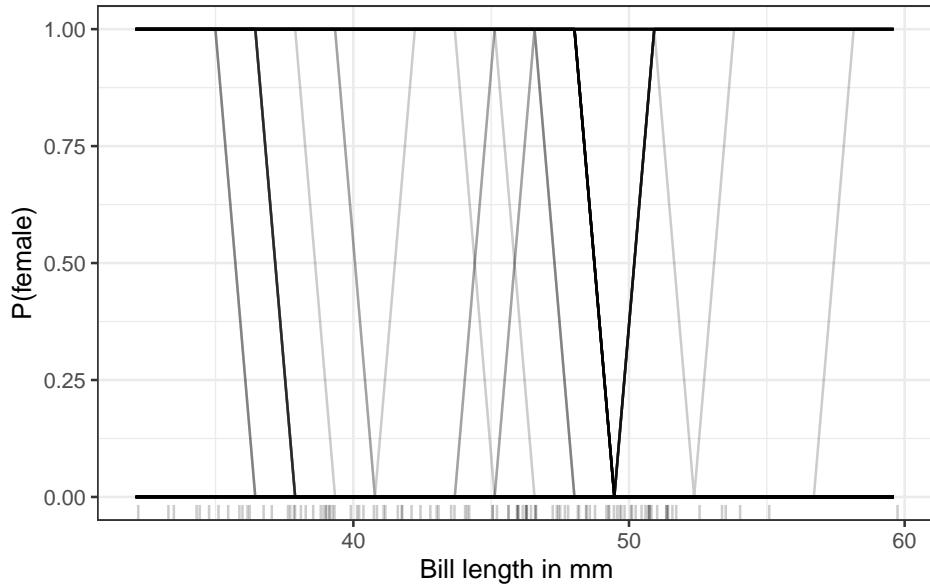


Figure 13.3: ICE plot of  $P(\text{Adelie})$  by bill length. Each line represents a penguin.

## 13.2 Centered ICE plot

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

$$ICE_j^{(i)}(x_j) = \hat{f}(x_j, \mathbf{x}_{-j}^{(i)}) - \hat{f}(a, \mathbf{x}_{-j}^{(i)})$$

where  $\hat{f}$  is the fitted model, and  $a$  is the anchor point.

Let's have a look at a centered ICE plot for temperature for the bike rental prediction:

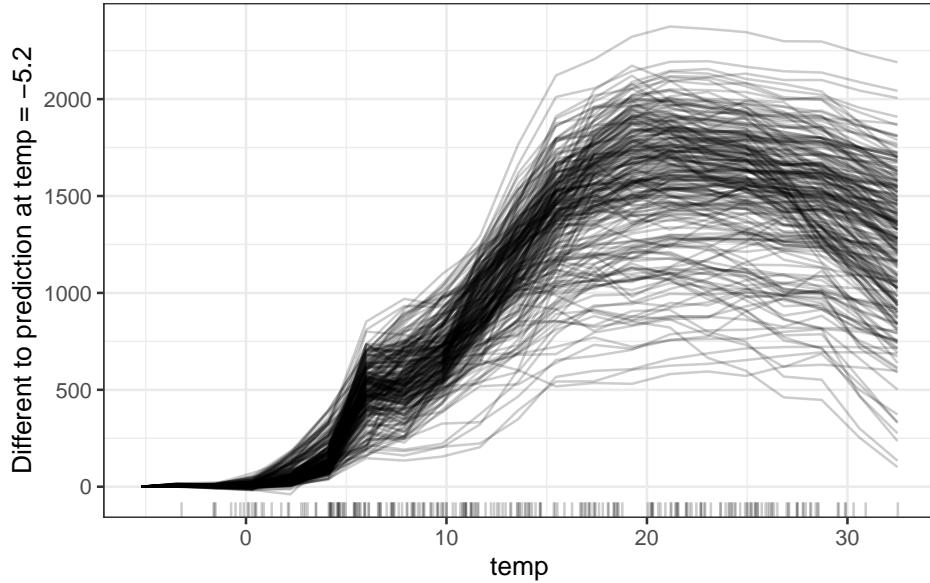


Figure 13.4: Centered ICE plots of predicted number of bikes by temperature. The lines show the difference in prediction compared to the prediction with the temperature fixed at its observed minimum.

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

### 13.3 Derivative ICE plot

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

$$\hat{f}(\mathbf{x}) = \hat{f}(x_j, \mathbf{x}_C) = g(x_j) + h(\mathbf{x}_{-j}), \quad \text{with} \quad \frac{\partial \hat{f}(\mathbf{x})}{\partial x_j} = g'(x_j)$$

Without interactions, the individual partial derivatives should be the same for all instances. If they differ, it's due to interactions, and it becomes visible in the d-ICE plot. In addition

to displaying the individual curves for the derivative of the prediction function with respect to the feature in  $j$ , showing the standard deviation of the derivative helps to highlight regions in feature  $j$  with heterogeneity in the estimated derivatives. The derivative ICE plot takes a long time to compute and is rather impractical.

## 13.4 Strengths

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

ICE curves can **uncover heterogeneous relationships**.

## 13.5 Limitations

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

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

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

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

## 13.6 Software and alternatives

ICE plots are implemented in the R packages `iml` (Molnar, Casalicchio, and Bischl 2018) (used for these examples), `ICEbox`, and `pdp`. Another R package that does something very similar to ICE is `condvis`. In Python, you can use [PiML](#) (Sudjianto et al. 2023).

## 14 LIME

Local surrogate models are interpretable models that are used to explain individual predictions of black box machine learning models. Local interpretable model-agnostic explanations (LIME), proposed by Ribeiro, Singh, and Guestrin (2016b), is an approach for fitting surrogate models. Surrogate models are trained to approximate the predictions of the underlying black box model.

The idea is quite intuitive. First, forget about the training data and imagine you only have the black box model where you can input data points and get the predictions of the model. You can probe the box as often as you want. Your goal is to understand why the machine learning model made a certain prediction. LIME tests what happens to the predictions when you give variations of your data into the machine learning model. LIME generates a new dataset consisting of perturbed samples and the corresponding predictions of the black box model. On this new dataset, LIME then trains an interpretable model, which is weighted by the proximity of the sampled instances to the instance of interest. The interpretable model can be anything from [Lasso](#) to a [decision tree](#). The learned model should be a good approximation of the machine learning model predictions locally, but it does not have to be a good global approximation. This kind of accuracy is also called local fidelity.

Mathematically, local surrogate models with interpretability constraint can be expressed as follows:

$$\text{explanation}(\mathbf{x}) = \arg \min_{g \in G} L(\hat{f}, g, \pi_{\mathbf{x}}) + \Omega(g)$$

The explanation model for instance  $\mathbf{x}$  is the model  $g$  (e.g., linear regression model) that minimizes loss  $L$  (e.g., mean squared error), which measures how close the explanation is to the prediction of the original model  $\hat{f}$  (e.g., an xgboost model), while the model complexity  $\Omega(g)$  is kept low (e.g., prefer fewer features).  $G$  is the family of possible explanations, for example, all possible linear regression models. The proximity measure  $\pi_{\mathbf{x}}$  defines how large the neighborhood around instance  $\mathbf{x}$  is that we consider for the explanation. In practice, LIME only optimizes the loss part. The user has to determine the complexity, e.g., by selecting the maximum number of features that the linear regression model may use.

The recipe for training local surrogate models:

- Select your instance of interest for which you want to have an explanation of its black box prediction.

- Perturb your dataset and get the black box predictions for these new points.
- Weight the new samples according to their proximity to the instance of interest.
- Train a weighted, interpretable model on the dataset with the variations.
- Explain the prediction by interpreting the local model.

In the current implementations in [R](#) and [Python](#), for example, linear regression can be chosen as an interpretable surrogate model. In advance, you have to select  $K$ , the number of features you want to have in your interpretable model. The lower  $K$ , the easier it is to interpret the model. A higher  $K$  potentially produces models with higher fidelity. There are several methods for training models with exactly  $K$  features. A good choice is [Lasso](#). A Lasso model with a high regularization parameter  $\lambda$  yields a model without any feature. By retraining the Lasso models with slowly decreasing  $\lambda$ , one after the other, the features get weight estimates that differ from zero. If there are  $K$  features in the model, you have reached the desired number of features. Other strategies are forward or backward selection of features. This means you either start with the full model (= containing all features) or with a model with only the intercept and then test which feature would bring the biggest improvement when added or removed, until a model with  $K$  features is reached.

How do you get the variations of the data? This depends on the type of data, which can be either text, image, or tabular data. For text and images, the solution is to turn single words or super-pixels on or off. In the case of tabular data, LIME creates new samples by perturbing each feature individually, drawing from a normal distribution with mean and standard deviation taken from the feature.

## 14.1 LIME for tabular data

Tabular data is data that comes in tables, with each row representing an instance and each column a feature. LIME samples are not taken around the instance of interest, but from the training data's mass center, which is problematic. But it increases the probability that the result for some of the sample points' predictions differs from the data point of interest, and that LIME can learn at least some explanation. Figure 14.1 visually explains how sampling and local model training works:

As always, the devil is in the detail. Defining a meaningful neighborhood around a point is difficult. LIME currently uses an exponential smoothing kernel to define the neighborhood. A smoothing kernel is a function that takes two data instances and returns a proximity measure. The kernel width determines how large the neighborhood is: A small kernel width means that an instance must be very close to influence the local model; a larger kernel width means that instances that are farther away also influence the model. If you look at [LIME's Python implementation \(file lime/lime\\_tabular.py\)](#), you will see that it uses an exponential smoothing kernel (on the normalized data), and the kernel width is 0.75 times the square root of the number of columns of the training data. It looks like an innocent line of code, but it's the

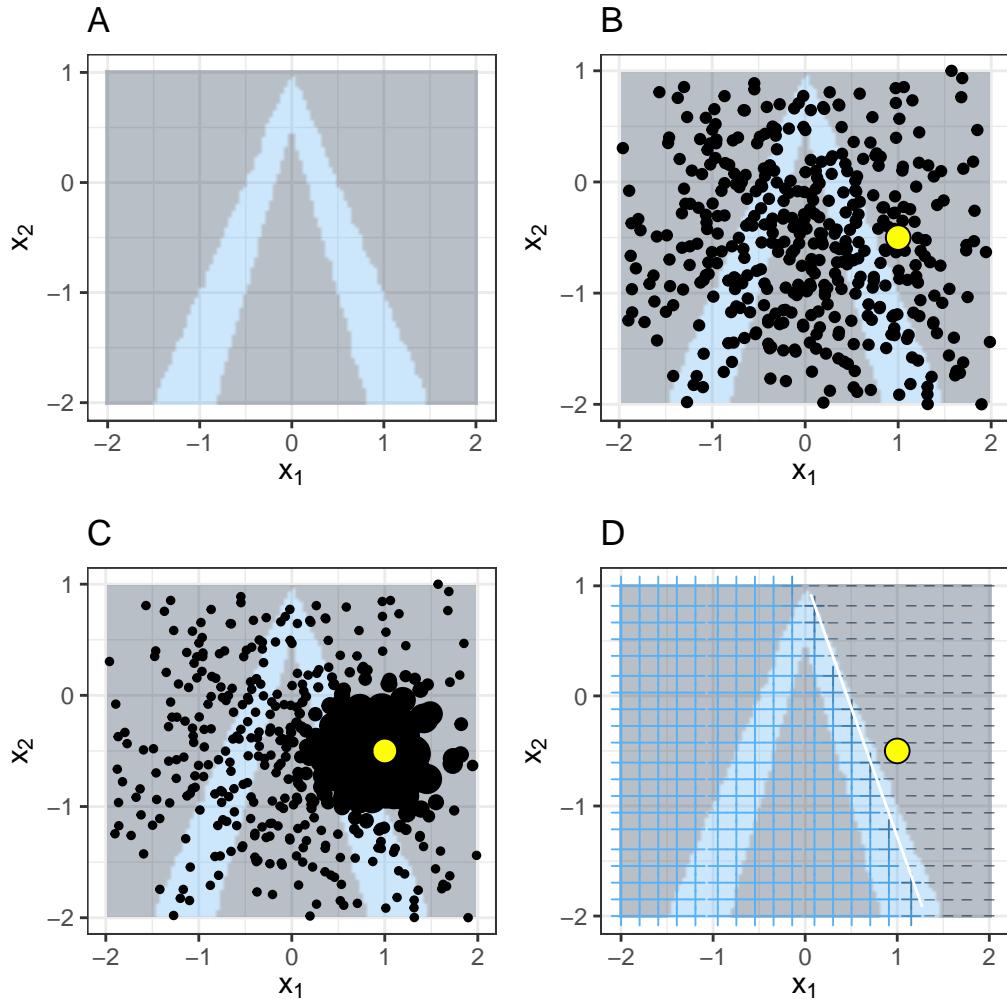


Figure 14.1: LIME algorithm for tabular data. A) Prediction surface given features  $x_1$  and  $x_2$ . Predicted classes: 1 (dark) or 0 (light). B) Instance of interest (big dot) and sampled data (small dots). C) Assign weights based on distance to instance. D) Signs (+/-) show the classifications of the locally learned model from the weighted samples. The white line marks the decision boundary ( $P(c=1) = 0.5$ ).

elephant sitting in your living room next to the good porcelain you got from your grandparents. The big problem is that we don't have a good way to find the best kernel or width. And where does the 0.75 even come from? In certain scenarios, you can easily turn your explanation around by changing the kernel width, as shown in Figure 14.2: The resulting linear regression model depends on the kernel width. Should the "true" local feature effect be negative, positive, or no effect for  $x = 1.6$ ?

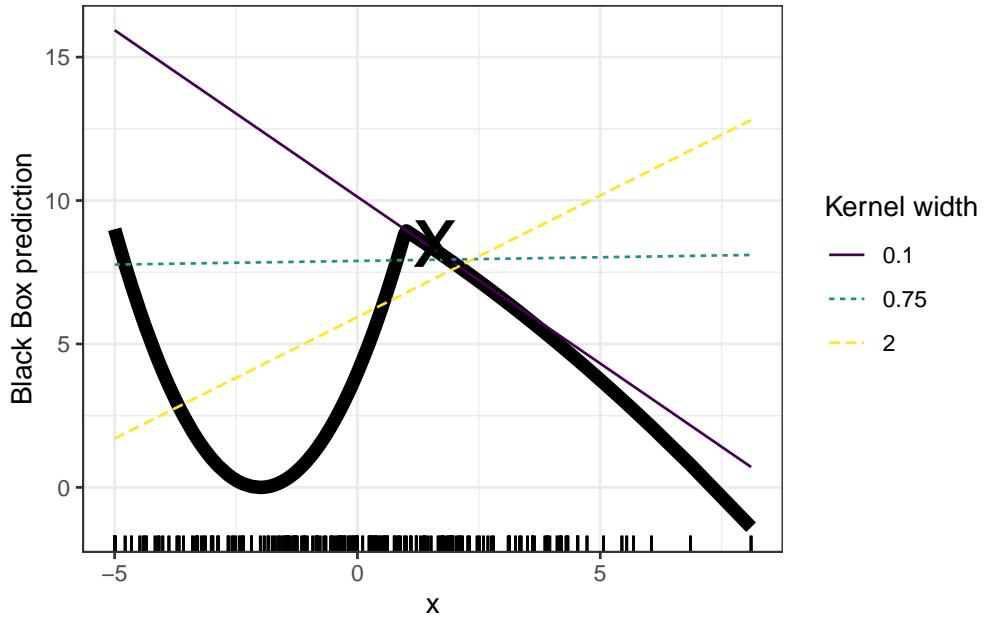


Figure 14.2: Explanation of the prediction of instance  $x = 1.6$  with different kernel widths. The model predictions are shown as a thick line and the distribution of the data is shown with rugs. Three local surrogate models with different kernel widths are computed.

The example shows only one feature. It gets worse in high-dimensional feature spaces. It's also very unclear whether the distance measure should treat all features equally. Is a distance unit for feature  $X_1$  identical to one unit for feature  $X_2$ ? Distance measures are quite arbitrary and distances in different dimensions (aka features) might not be comparable at all.

Let's look at a concrete example. We go back to the [penguins data](#), which is about classifying a penguin as female or male based on body measurements. We analyze the random forest trained to do that. The explanations are created with 2 features. The results of the sparse local linear models trained for two instances with different predicted classes are shown in Figure 14.3. Here, higher body mass and longer bills are associated with lower  $P(\text{female})$ .

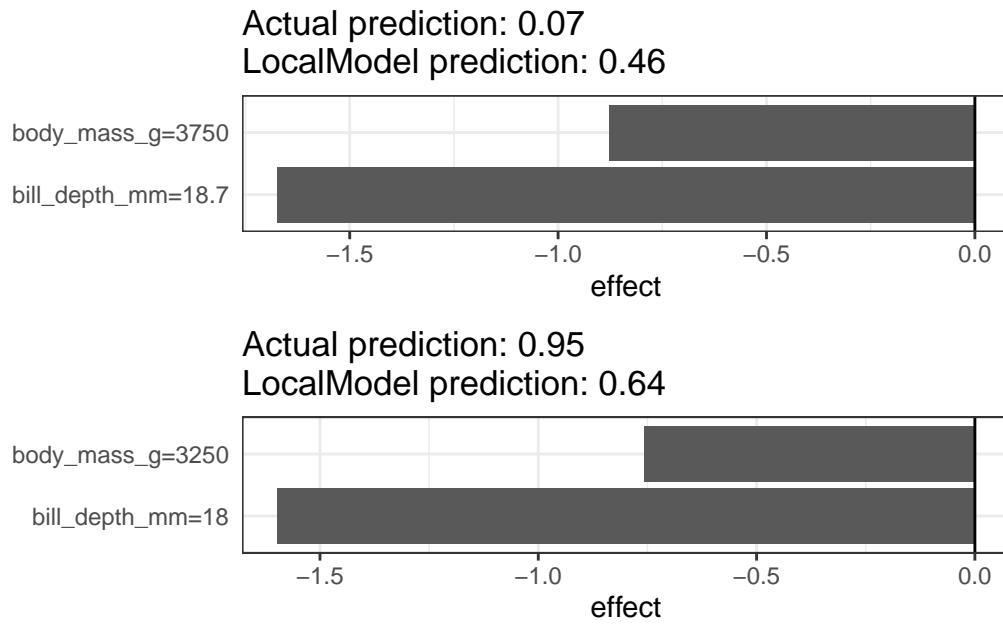


Figure 14.3: LIME explanations for two instances of the penguin dataset. The x-axis shows the feature effect, which is the weight times the actual feature value.

## 14.2 LIME for text data

LIME for text differs from LIME for tabular data. Variations of the data are generated differently: Starting from the original text, new texts are created by randomly removing words from the original text. The dataset is represented with binary features for each word. A feature is 1 if the corresponding word is included and 0 if it has been removed.

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

The comments were collected via the YouTube API from five of the ten most viewed videos on YouTube in the first half of 2015. All 5 are music videos. One of them is “Gangnam Style” by Korean artist Psy. The other artists are Katy Perry, LMFAO, Eminem, and Shakira. Check out two comments in Table 14.1, one spam (class=1), one legit (class=0). The comments were manually labeled as spam or legitimate. Spam was coded with a “1” and legitimate comments with a “0”. You can also go to YouTube and take a look at the comment section. But please do not get caught in YouTube hell, and end up watching videos of monkeys stealing and drinking cocktails from tourists on the beach. The Google Spam detector has also probably changed a lot since 2015. [Watch the view-record-breaking video “Gangnam Style” here](#).

Table 14.1: Two examples of comments.

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

Table 14.2: Variations of one of the YouTube comments along with their weights (based on distance) and predicted probability of being spam.

For	Christmas	Song	visit	my	channel!	;) prob	weight
1	0	1	1	0	0	1	0.17
0	1	1	1	1	0	1	0.17
1	0	0	1	1	1	1	0.99
1	0	1	1	1	1	1	0.99
0	1	1	1	0	0	1	0.17

The black box model is a deep decision tree trained on the document word matrix. Each comment is one document (= one row), and each column is the number of occurrences of a given word. Short decision trees are easy to understand, but in this case, the tree is very deep. Also, in place of this tree, there could have been a recurrent neural network or a support vector machine trained on word embeddings (abstract vectors).

The next step is to create some variations of the datasets used in a local model. For example, some variations of one of the comments, see Table 14.2. Each column corresponds to one word in the sentence. Each row is a variation; 1 means that the word is part of this variation, and 0 means that the word has been removed. The corresponding sentence for one of the variations is “Christmas Song visit my ;)”. The “prob” column shows the predicted probability of spam for each of the sentence variations. The “weight” column shows the proximity of the variation to the original sentence, calculated as 1 minus the proportion of words that were removed, for example if 1 out of 7 words was removed, the proximity is  $1 - 1/7 = 0.86$ .

And finally Table 14.3 shows the two sentences (one spam, one no spam) with their estimated local weights found by the LIME algorithm: The word “channel” indicates a high probability of spam. For the non-spam comment, no non-zero weight was estimated, because no matter which word is removed, the predicted class remains the same.

## 14.3 LIME for image data

*This section was written by Verena Haunschmid.*

Table 14.3: LIME explanations for text classification.

case	label_prob	feature	feature_weight
1	0.1701170	is	0.000000
1	0.1701170	good	0.000000
1	0.1701170	a	0.000000
2	0.9939024	channel!	6.180747
2	0.9939024	;)	0.000000
2	0.9939024	visit	0.000000

LIME for images works differently than LIME for tabular data and text. Intuitively, it would not make much sense to perturb individual pixels, since many more than one pixel contribute to one class. Randomly changing individual pixels would probably not change the predictions by much. Therefore, variations of the images are created by segmenting the image into “superpixels” and turning superpixels off or on. Superpixels are interconnected pixels with similar colors and can be turned off by replacing each pixel with a user-defined color, such as gray. The user can also specify a probability for turning off a superpixel in each permutation.

In this example, we look at a classification made by the Inception V3 neural network. The image used shows some bread I baked that is in a bowl (see Figure 14.4). Since we can have several predicted labels per image (sorted by probability), we can explain the top labels. The top prediction is “Bagel” with a probability of  $P(Y = \text{Bagel}) = 0.77$ , followed by “Strawberry” with a probability of  $P(Y = \text{Strawberry}) = 0.04$ . The following images show for “Bagel” and “Strawberry” the LIME explanations. The explanations can be displayed directly on the image samples. Green means that this part of the image increases the probability for the label, and red means a decrease.

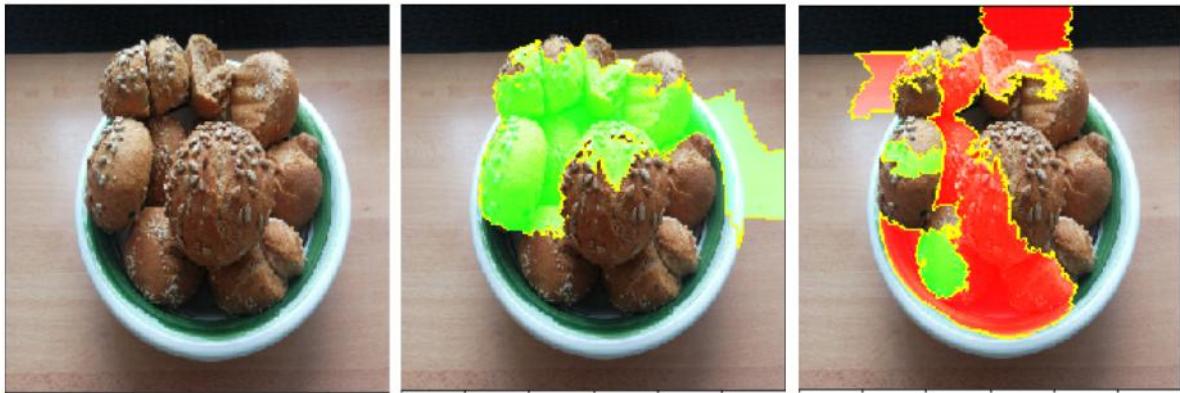


Figure 14.4: Left: Image of a bowl of bread. Middle and right: LIME explanations for the top 2 classes (Bagel, Strawberry) for image classification made by Google’s Inception V3 neural network.

The prediction and explanation for “Bagel” are very reasonable, even if the prediction is wrong – these are clearly no bagels since the hole in the middle is missing.

## 14.4 Strengths

Even if you **replace the underlying machine learning model**, you can still use the same local, interpretable model for explanation. Suppose the people looking at the explanations understand decision trees best. Because you use local surrogate models, you use decision trees as explanations without actually having to use a decision tree to make the predictions. For example, you can use an SVM. And if it turns out that an xgboost model works better, you can replace the SVM and still use a decision tree to explain the predictions.

Local surrogate models benefit from the literature and experience of training and interpreting interpretable models.

When using Lasso or short trees, the resulting **explanations are short (= selective) and possibly contrastive**. Therefore, they make **human-friendly explanations**.

LIME is one of the few methods that **works for tabular data, text, and images**.

The **fidelity measure** (how well the interpretable model approximates the black box predictions) gives us a good idea of how reliable the interpretable model is in explaining the black box predictions in the neighborhood of the data instance of interest.

LIME is implemented in Python ([lime library](#)) and R ([lime package](#) and [iml package](#)) and is **very easy to use**.

The explanations created with local surrogate models **can use other (interpretable) features than the original model was trained on**. Of course, these interpretable features must be derived from the data instances. A text classifier can rely on abstract word embeddings as features, but the explanation can be based on the presence or absence of words in a sentence. A regression model can rely on a non-interpretable transformation of some attributes, but the explanations can be created with the original attributes. For example, the regression model could be trained on components of a principal component analysis (PCA) of answers to a survey, but LIME might be trained on the original survey questions. Using interpretable features for LIME can be a big advantage over other methods, especially when the model was trained with non-interpretable features.

## 14.5 Limitations

The **choice of neighborhood is an unsolved problem** when using LIME with tabular data. In my opinion, it's the biggest problem with LIME and the reason why I would recommend using LIME only with great care. For each application, you have to try different kernel settings

and see for yourself if the explanations make sense. Unfortunately, this is the best advice I can give to find good kernel widths.

Sampling could be improved in the current implementation of LIME. Data points are sampled from a Gaussian distribution, ignoring the correlation between features. This **can lead to unlikely data points**, which can then be used to learn local explanation models.

The **complexity of the explanation model has to be defined in advance**. This is just a small complaint because, in the end, the user always has to define the compromise between fidelity and sparsity.

Another really big problem is the **instability of the explanations**. In an article (Alvarez-Melis and Jaakkola 2018), the authors showed that the explanations of two very close points varied greatly in a simulated setting. Also, in my experience, if you repeat the sampling process, then the explanations that come out can be different. Instability means that it is difficult to trust the explanations, and you should be very critical.

LIME explanations can be manipulated by the data scientist to hide biases (Slack et al. 2020). The possibility of manipulation makes it more difficult to trust explanations generated with LIME.

Conclusion: Local surrogate models, with LIME as a concrete implementation, are very promising. But the method is still in the development phase and many problems need to be solved before it can be safely applied.

## 14.6 Software

The original Python implementation is in the [lime package](#). Other Python implementations can be found in [PiML](#), and [eli5](#). In R, you can use [iml](#) or [DALEX](#).

# 15 Counterfactual Explanations

Authors: Susanne Dandl & Christoph Molnar

A counterfactual explanation describes a causal situation in the form: “If X had not occurred, Y would not have occurred.” For example: “If I hadn’t taken a sip of this hot coffee, I wouldn’t have burned my tongue.” Event Y is that I burned my tongue; cause X is that I had a hot coffee. Thinking in counterfactuals requires imagining a hypothetical reality that contradicts the observed facts (for example, a world in which I’ve not drunk the hot coffee), hence the name “counterfactual.” The ability to think in counterfactuals makes us humans so smart compared to other animals.

In interpretable machine learning, counterfactual explanations can be used to explain predictions of individual instances.<sup>1</sup> Displayed as a graph (Figure 15.1), the relationship between the inputs and the prediction is very simple: The feature values cause the prediction. 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.

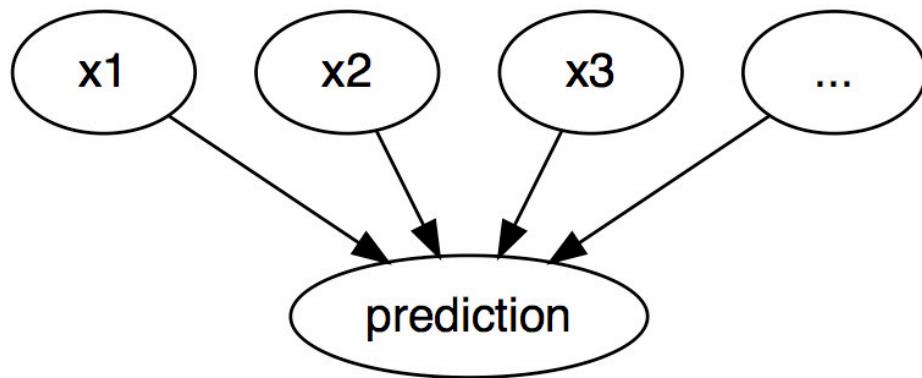


Figure 15.1: The causal relationships between inputs of a machine learning model and the predictions, when the model is merely seen as a black box. The inputs cause the prediction (not necessarily reflecting the real causal relation of the data).

---

<sup>1</sup>“Counterfactuals” is an overloaded term. In causal inference, they have a different meaning and are connected to hypothetical, causal interventions.

Given this simple graph, it's easy to see how we can simulate counterfactuals for predictions of machine learning models: We simply change the feature values of an instance before making the predictions and we analyze how the prediction changes. We're interested in scenarios in which the prediction changes in a relevant way, like a flip in predicted class (for example, credit application accepted or rejected), or in which the prediction reaches a certain threshold (for example, 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.**

There are both model-agnostic and model-specific counterfactual explanation methods, but in this chapter we focus on model-agnostic methods that only work with the model inputs and outputs (and not the internal structure of specific models).

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’s 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 more per year, he would get the loan. Or if Peter had fewer credit cards and had not defaulted on a loan five years ago, he would get the loan. Peter will never know the reasons for the rejection, as the bank has no interest in transparency, but that is another story.

In our second example, we want to explain a model that predicts a continuous outcome with counterfactual explanations. Anna wants to rent out her apartment, but she is not sure how much to charge for it, so she decides to train a machine learning model to predict the rent. Of course, since Anna is a data scientist, that is how she solves her problems. After entering all the details about size, location, whether pets are allowed, and so on, the model tells her that she can charge 900 EUR. She expected 1000 EUR 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 EUR if it were 15 m<sup>2</sup> larger. Interesting, but non-actionable knowledge, because she cannot enlarge her apartment. Finally, by tweaking only the feature values under her control (built-in kitchen yes/no, pets allowed yes/no, type of floor, etc.), she finds out that if she allows pets and installs windows with better insulation, she can charge 1000 EUR. Anna has intuitively worked with counterfactuals to change the outcome. Note that Anna worked with the rent prediction model and wasn't necessarily interested in whether these factors are truly causal for higher rent in the “real world.”

### Warning

By default, counterfactuals (the IML method) alone don't support **causal** claims about the real world. This would require a causal model.

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). An obvious first requirement is that **a counterfactual instance produces the predefined prediction as closely as possible**. It's not always possible to find a counterfactual with the predefined prediction. For example, in a classification setting with two classes, a rare class and a frequent class, the model might always classify an instance as the frequent class. Changing the feature values so that the predicted label would flip from the frequent class to the rare class might be impossible. We therefore want to relax the requirement that the prediction of the counterfactual must match the predefined outcome exactly. 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 minimal changes in the features so that the predicted probability changes from 2% to 10% (or close to 10%)?

### Use probabilities

For classification tasks, it's better to define the counterfactual in terms of predicted probabilities than class outcomes.

Another quality criterion is that **a counterfactual should be as similar as possible to the instance regarding feature values**. The distance between two instances can be measured, for example, with the Manhattan distance or the Gower distance if we have both discrete and continuous features. The counterfactual should not only be close to the original instance, but should also **change as few features as possible**. To measure how good a counterfactual explanation is in this metric, we can simply count the number of changed features or, in fancy mathematical terms, measure the  $L_0$  norm between the counterfactual and actual instance.

Third, it is often desirable to generate **multiple diverse counterfactual explanations** so that the decision subject gets access to multiple viable ways of generating a different outcome. For instance, continuing our loan example, one counterfactual explanation might suggest only doubling the income to get a loan, while another counterfactual might suggest shifting to a nearby city and increasing the income by a small amount to get a loan. It could be noted that while the first counterfactual might be possible for some, the latter might be more actionable for others. Thus, besides providing a decision subject with different ways to get the desired outcome, diversity also enables “diverse” individuals to alter the features that are convenient for them.

The last requirement is that **a counterfactual instance should have feature values that are likely**. It would not make sense to generate a counterfactual explanation for the rent example where the size of an apartment is negative or the number of rooms is set to 200. It’s even better when the counterfactual is likely according to the joint distribution of the data; for example, an apartment with 10 rooms and 20 m<sup>2</sup> should not be regarded as a counterfactual explanation. Ideally, if the number of square meters is increased, an increase in the number of rooms should also be proposed.

## 15.1 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 based on the criteria mentioned above. This loss takes as input the instance of interest, a counterfactual, and the desired (counterfactual) outcome. Then, we can find the counterfactual explanation that minimizes this loss using an optimization algorithm. Many methods proceed in this way, but differ in their definition of the loss function and optimization method.

In the following, we focus on two of them: first, the one by Wachter, Mittelstadt, and Russell (2018), who introduced counterfactual explanation as an interpretation method and, second, the one by Dandl et al. (2020) that takes into account all four criteria mentioned above.

### 15.1.1 Method by Wachter et al.

Wachter et al. suggest minimizing the following loss:

$$L(\mathbf{x}, \mathbf{x}', y', \lambda) = \lambda \cdot (\hat{f}(\mathbf{x}') - y')^2 + d(\mathbf{x}, \mathbf{x}')$$

The first term is the quadratic distance between the model prediction for the counterfactual  $\mathbf{x}'$  and the desired outcome  $y'$ , which the user must define in advance. The second term is the distance  $d$  between the instance  $\mathbf{x}$  to be explained and the counterfactual  $\mathbf{x}'$ . The loss measures how far the predicted outcome of the counterfactual is from the predefined outcome and how far the counterfactual is from the instance of interest. The distance function  $d$  is defined as the Manhattan distance weighted with the inverse median absolute deviation (MAD) of each feature.

$$d(\mathbf{x}, \mathbf{x}') = \sum_{j=1}^p \frac{|x_j - x'_j|}{MAD_j}$$

The total distance is the sum of all  $p$  feature-wise distances, that is, the absolute differences of feature values between instance  $\mathbf{x}$  and counterfactual  $\mathbf{x}'$ . The feature-wise distances are scaled by the inverse of the median absolute deviation of feature  $j$  over the dataset defined as:

$$MAD_j = \text{median}_{i \in \{1, \dots, n\}} (|x_j^{(i)} - \text{median}_{l \in \{1, \dots, n\}} (x_j^{(l)})|)$$

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 is more robust to outliers. Scaling with the MAD is necessary to bring all the features to the same scale – it should not matter whether you measure the size of an apartment in square meters or square feet.

The parameter  $\lambda$  balances the distance in prediction (first term) against the distance in feature values (second term). The loss is solved for a given  $\lambda$  and returns a counterfactual  $\mathbf{x}'$ . A higher value of  $\lambda$  means that we prefer counterfactuals with predictions close to the desired outcome  $y'$ , while a lower value means that we prefer counterfactuals  $\mathbf{x}'$  that are very similar to  $\mathbf{x}$  in the feature values. If  $\lambda$  is very large, the instance with the prediction closest to  $y'$  will be selected, regardless of how far it is from  $\mathbf{x}$ . 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  $\mathbf{x}$ . The authors of the method suggest instead of selecting a value for  $\lambda$ , to select a tolerance  $\epsilon$  for how far away from  $y'$  the prediction of the counterfactual instance is allowed to be. This constraint can be written as:

$$|\hat{f}(\mathbf{x}') - y'| \leq \epsilon$$

To minimize this loss function, any suitable optimization algorithm can be used, such as Nelder-Mead. If you have access to the gradients of the machine learning model, you can use gradient-based methods like ADAM. The instance  $\mathbf{x}$  to be explained, the desired output  $y'$ , and the tolerance parameter  $\epsilon$  must be set in advance. The loss function is minimized for  $\mathbf{x}'$

and the (locally) optimal counterfactual  $\mathbf{x}'$  returned while increasing  $\lambda$  until a sufficiently close solution is found (= within the tolerance parameter):

$$\arg \min_{\mathbf{x}'} \max_{\lambda} L(\mathbf{x}, \mathbf{x}', y', \lambda).$$

Overall, the recipe for producing the counterfactuals is simple:

1. Select an instance  $\mathbf{x}$  to be explained, the desired outcome  $y'$ , a tolerance  $\epsilon$ , and a (low) initial value for  $\lambda$ .
2. Sample a random instance as initial counterfactual.
3. Optimize the loss with the initially sampled counterfactual as the starting point.
4. While  $|\hat{f}(\mathbf{x}') - y'| > \epsilon$ :
  - Increase  $\lambda$ .
  - Optimize the loss with the current counterfactual as the starting point.
  - Return the counterfactual that minimizes the loss.
5. Repeat steps 2-4 and return the list of counterfactuals or the one that minimizes the loss.

The proposed method has some disadvantages. It **only takes the first and second criteria into account**, not the last two (“produce counterfactuals with only a few feature changes and likely feature values”).  $d$  does not prefer sparse solutions since increasing 10 features by 1 will give the same distance to  $\mathbf{x}$  as increasing one feature by 10. Unrealistic feature combinations are not penalized.

The method does **not handle categorical features** with many different levels well. The authors of the method suggested running the method separately for each combination of feature values of the categorical features, but this will lead to a combinatorial explosion if you have multiple categorical features with many values. For example, six categorical features with ten unique levels would mean one million runs.

Let's now have a look at another approach overcoming these issues.

### 15.1.2 Method by Dandl et al.

Dandl et al. suggest simultaneously minimizing a four-objective loss:

$$L(\mathbf{x}, \mathbf{x}', y', \mathbf{X}^{\text{obs}}) = (o_1(\hat{f}(\mathbf{x}'), y'), o_2(\mathbf{x}, \mathbf{x}'), o_3(\mathbf{x}, \mathbf{x}'), o_4(\mathbf{x}', \mathbf{X}^{\text{obs}}))$$

Each of the four objectives  $o_1$  to  $o_4$  corresponds to one of the four criteria mentioned above. The first objective  $o_1$  reflects that the prediction of our counterfactual  $\mathbf{x}'$  should be as close

as possible to our desired prediction  $y'$ . We therefore want to minimize the distance between  $\hat{f}(\mathbf{x}')$  and  $y'$ , here calculated by the Manhattan metric ( $L_1$  norm):

$$o_1(\hat{f}(\mathbf{x}'), y') = \begin{cases} 0 & \text{if } \hat{f}(\mathbf{x}') \in y' \\ \inf_{y' \in y'} |\hat{f}(\mathbf{x}') - y'| & \text{else} \end{cases}$$

The second objective  $o_2$  reflects that our counterfactual should be as similar as possible to our instance  $\mathbf{x}$ . It quantifies the distance between  $\mathbf{x}'$  and  $\mathbf{x}$  as the Gower distance:

$$o_2(\mathbf{x}, \mathbf{x}') = \frac{1}{p} \sum_{j=1}^p \delta_G(x_j, x'_j)$$

with  $p$  being the number of features. The value of  $\delta_G$  depends on the feature type of  $x_j$ :

$$\delta_G(x_j, x'_j) = \begin{cases} \frac{1}{\widehat{R}_j} |x_j - x'_j| & \text{if } x_j \text{ numerical} \\ \mathbb{I}_{x_j \neq x'_j} & \text{if } x_j \text{ categorical} \end{cases}$$

Dividing the distance of a numeric feature  $j$  by  $\widehat{R}_j$ , the observed value range, scales  $\delta_G$  for all features between 0 and 1.

The Gower distance can handle both numerical and categorical features, but does not count how many features were changed. Therefore, we count the number of features in a third objective  $o_3$  using the  $L_0$  norm:

$$o_3(\mathbf{x}, \mathbf{x}') = \|\mathbf{x} - \mathbf{x}'\|_0 = \sum_{j=1}^p I_{x'_j \neq x_j}.$$

By minimizing  $o_3$  we aim for our third criterion – sparse feature changes.

The fourth objective  $o_4$  reflects that our counterfactuals should have likely combinations of feature values. We can infer how “likely” a data point is using the training data or another dataset. We denote this dataset as  $\mathbf{X}^{\text{obs}}$ . As an approximation for the likelihood,  $o_4$  measures the average Gower distance between  $\mathbf{x}'$  and the nearest observed data point  $x^{[1]} \in \mathbf{X}^{\text{obs}}$ :

$$o_4(\mathbf{x}', \mathbf{X}^{\text{obs}}) = \frac{1}{p} \sum_{j=1}^p \delta_G(x'_j, x_j^{[1]})$$

Compared to Wachter et al.,  $L(\mathbf{x}, \mathbf{x}', y', \mathbf{X}^{\text{obs}})$  has no balancing/weighting terms like  $\lambda$ . We do not want to collapse the four objectives  $o_1$ ,  $o_2$ ,  $o_3$ , and  $o_4$  into a single objective by summing them up and weighting them, but we want to optimize all four terms simultaneously.

How can we do that? We use the **Nondominated Sorting Genetic Algorithm** (Deb et al. 2002) or short NSGA-II. NSGA-II is a nature-inspired algorithm that applies Darwin's law of the "survival of the fittest". We denote the fitness of a counterfactual by its vector of objective values ( $o_1, o_2, o_3, o_4$ ). The lower the values of the objectives for a counterfactual, the "fitter" it is.

The algorithm consists of four steps that are repeated until a stopping criterion is met, for example, a maximum number of iterations/generations. Figure 15.2 visualizes the four steps of one generation.

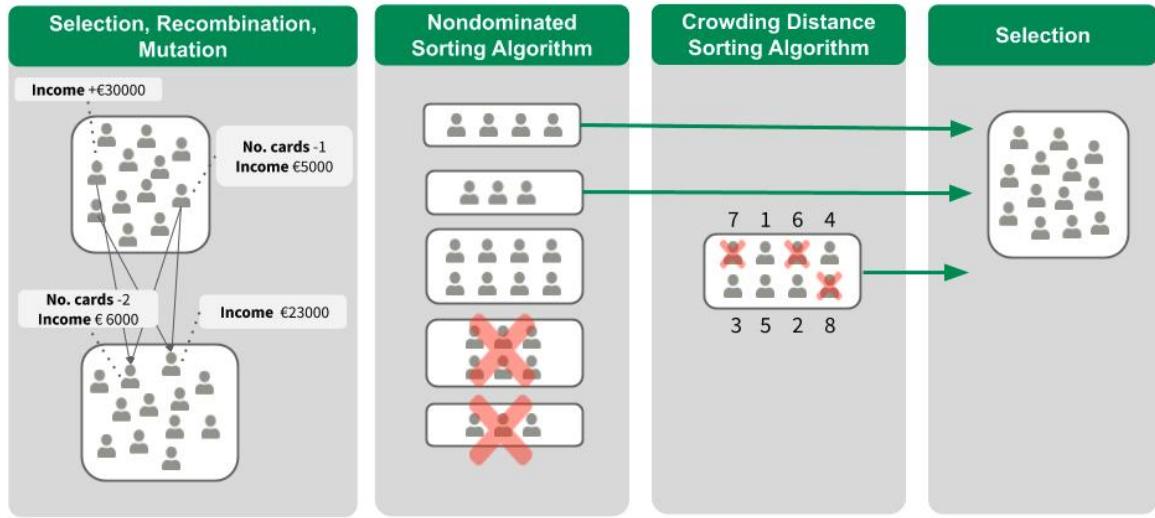


Figure 15.2: Visualization of one generation of the NSGA-II algorithm.

In the first generation, a group of counterfactual candidates is initialized by randomly changing some of the features compared to our instance  $\mathbf{x}$  to be explained. Sticking with the above credit example, one counterfactual could suggest increasing the income by €30,000 while another one proposes to have no default in the last five years and a reduction in age by ten. All other feature values are equal to the values of  $\mathbf{x}$ . Each candidate is then evaluated using the four objective functions above. Among them, we randomly select some candidates, where fitter candidates are more likely to be selected. The candidates are pairwise recombined to produce children that are similar to them by averaging their numerical feature values or by crossing over their categorical features. In addition, we slightly mutate the feature values of the children to explore the whole feature space.

From the two resulting groups, one with parents and one with children, we only want the best half using two sorting algorithms. The nondominated sorting algorithm sorts the candidates according to their objective values. If candidates are equally good, the crowding distance sorting algorithm sorts the candidates according to their diversity.

Given the ranking of the two sorting algorithms, we select the most promising and/or most diverse half of the candidates. We use this set for the next generation and start again with the selection, recombination, and mutation process. By repeating the steps over and over, we hopefully approach a diverse set of promising candidates with low objective values. From this set, we can choose those with which we are most satisfied, or we can give a summary of all counterfactuals by highlighting which and how often features have been changed.

## 15.2 Example

The following example is based on the credit dataset example in Dandl et al. (2020). The German Credit Risk dataset can be found on the machine learning challenges platform [kaggle.com](https://www.kaggle.com). The authors trained a support vector machine (with radial basis kernel) to predict the probability that a customer has a good credit risk. The corresponding dataset has 522 complete observations and nine features containing credit and customer information.

The goal is to find counterfactual explanations for a customer with feature values in Table 15.1.

Table 15.1: Feature values of a particular customer

age	sex	job	housing	savings	amount	dur.	purpose
58	f	unskilled	free	little	6143	48	car

The SVM predicts that the probability that the person has a good credit risk is 24.2%. The counterfactuals should answer how the input features need to be changed to get a predicted probability larger than 50%. Table 15.2 shows the ten best counterfactuals. The first five columns contain the proposed feature changes (only altered features are displayed), the next three columns show the objective values ( $o_1$  equals 0 in all cases), and the last column displays the predicted probability.

Table 15.2: The ten best counterfactuals found for the customer

age	sex	job	amount	dur.	$o_2$	$o_3$	$o_4$	$f(x')$
		skilled		-20	0.108	2	0.036	0.501
		skilled		-24	0.114	2	0.029	0.525
		skilled		-22	0.111	2	0.033	0.513
-6		skilled		-24	0.126	3	0.018	0.505
-3		skilled		-24	0.120	3	0.024	0.515
-1		skilled		-24	0.116	3	0.027	0.522
-3	m			-24	0.195	3	0.012	0.501
-6	m			-25	0.202	3	0.011	0.501

age	sex	job	amount	dur.	$o_2$	$o_3$	$o_4$	$f(x')$
-30	m	skilled		-24	0.285	4	0.005	0.590
-4	m		-1254	-24	0.204	4	0.002	0.506

All counterfactuals have predicted probabilities greater than 50% and do not dominate each other. Nondominated means that none of the counterfactuals has smaller values in all objectives than the other counterfactuals. We can think of our counterfactuals as a set of trade-off solutions.

They all suggest a reduction of the duration from 48 months to a minimum of 23 months, some of them propose that the woman should become skilled instead of unskilled. Some counterfactuals even suggest changing the gender from female to male, which shows a gender bias of the model. This change is always accompanied by a reduction in age between one and 30 years. We can also see that, although some counterfactuals suggest changes to four features, these counterfactuals are the ones that are closest to the training data.

## 15.3 Strengths

**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 counterfactuals are not as dangerous as methods like [LIME](#), where it is unclear how far we can extrapolate the local model for the interpretation.

The counterfactual method creates a new instance, but we can also summarize a counterfactual by reporting which feature values have changed. This gives us **two options for reporting our results**. You can either report the counterfactual instance or highlight which features have been changed between the instance of interest and the counterfactual instance.

The **counterfactual method does not require access to the data or the model**. It only requires access to the model's prediction function, which would also work via a web API, for example. This is attractive for companies which are audited by third parties or which are offering explanations for users without disclosing the model or data. A company has an interest in protecting the model and data because of trade secrets or data protection reasons. Counterfactual explanations offer a balance between explaining model predictions and protecting the interests of the model owner.

The method **works also with systems that do not use machine learning**. We can create counterfactuals for any system that receives inputs and returns outputs. The system that predicts apartment rents could also consist of handwritten rules, and counterfactual explanations would still work.

The counterfactual explanation method is relatively easy to implement, since it's essentially a loss function (with a single or many objectives) 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).

Counterfactuals are useful for the goal of justification, especially recourse, since they are truthful and simple. Compared to other interpretation methods, they are not just estimates of something, like Shapley values. But counterfactuals are just data newly created data instances for which we can report what the model predicts.

## 15.4 Limitations

For each instance, you will usually find multiple counterfactual explanations (**Rashomon effect**). This is inconvenient – most people prefer simple explanations over the complexity of the real world. It's 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 humans then can select the ones that correspond to their previous knowledge.

Counterfactuals are not as useful for model and data insights, since insights from counterfactual are specific to one instance and one counterfactual prediction. This is a very limited insight, even when compared to other local methods.

## 15.5 Software and alternatives

The multi-objective counterfactual explanation method by Dandl et al. is implemented in a [GitHub repository](#).

In the Python package [Alibi](#), authors implemented a [simple counterfactual method](#), as well as an [extended method](#) that uses class prototypes to improve the interpretability and convergence of the algorithm outputs (Van Looveren and Klaise 2021).

Karimi et al. (2020) also provided a Python implementation of their algorithm MACE in a [GitHub repository](#). They translated necessary criteria for proper counterfactuals into logical formulae and used satisfiability solvers to find counterfactuals that satisfy them.

Mothilal, Sharma, and Tan (2020) developed [DiCE \(Diverse Counterfactual Explanation\)](#) to generate a diverse set of counterfactual explanations based on determinantal point processes. DiCE implements both a model-agnostic and a gradient-based method.

Another way to search counterfactuals is the Growing Spheres algorithm by Laugel et al. (2017). 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 first drawing a sphere around the point of interest, sampling points within that sphere, and checking whether one of the sampled points yields the desired prediction. Then they contract or expand the sphere accordingly until a (sparse) counterfactual is found and finally returned.

Anchors by Ribeiro, Singh, and Guestrin (2018) are the opposite of counterfactuals; see the chapter about [Scoped Rules \(Anchors\)](#).

# 16 Scoped Rules (Anchors)

*Authors: Tobias Goerke & Magdalena Lang (with later edits from Christoph Molnar)*

The anchors method explains individual predictions of any black box classification model by finding a decision rule that “anchors” the prediction sufficiently. A rule anchors a prediction if changes in other feature values do not affect the prediction. Anchors utilizes reinforcement learning techniques in combination with a graph search algorithm to reduce the number of model calls (and hence the required runtime) to a minimum while still being able to recover from local optima. Ribeiro, Singh, and Guestrin (2018) proposed the anchors algorithm – the same researchers who introduced the [LIME](#) algorithm.

Like its predecessor, the anchors approach deploys a *perturbation-based* strategy to generate *local* explanations for predictions of black box machine learning models. However, instead of surrogate models used by LIME, the resulting explanations are expressed as easy-to-understand *IF-THEN* rules, called *anchors*. These rules are reusable since they are *scoped*: anchors include the notion of coverage, stating precisely to which other, possibly unseen, instances they apply. Finding anchors involves an exploration or multi-armed bandit problem, which originates in the discipline of reinforcement learning. To this end, neighbors, or perturbations, are created and evaluated for every instance that is being explained. Doing so allows the approach to disregard the black box’s structure and its internal parameters so that these can remain both unobserved and unaltered. Thus, the algorithm is *model-agnostic*, meaning it can be applied to **any** class of model.

In their paper, the authors compare both of their algorithms and visualize how differently these consult an instance’s neighborhood to derive results. For this, Figure 16.1 depicts both LIME and anchors locally explaining a complex binary classifier (predicts either – or +) using two exemplary instances. LIME’s results do not indicate how faithful they are, as LIME solely learns a linear decision boundary that best approximates the model given a perturbation space  $\mathcal{D}$ . Given the same perturbation space, the anchors approach constructs explanations whose coverage is adapted to the model’s behavior, and the approach clearly expresses their boundaries. Thus, they are faithful by design and state exactly for which instances they are valid. This property makes anchors particularly intuitive and easy to comprehend.

As mentioned before, the algorithm’s results or explanations come in the form of rules, called anchors. The following simple example illustrates such an anchor. For instance, suppose we are given a bivariate black box model that predicts whether or not a passenger survived the Titanic disaster. Now we would like to know *why* the model predicts for one specific person

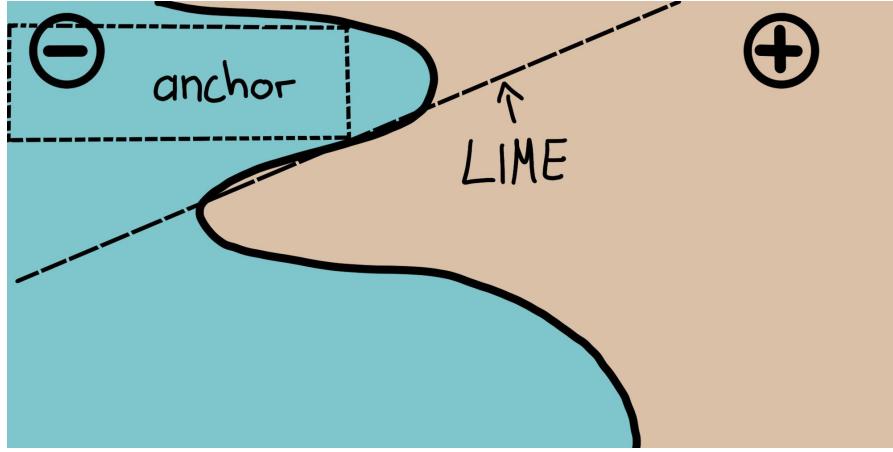


Figure 16.1: LIME vs. Anchors – A Toy Visualization. Inspired by figure from Ribeiro, Singh, and Guestrin (2018).

(see Table 16.1) that they survived. The anchors algorithm provides a result explanation like the one shown below.

Table 16.1: Example to be explained

Feature	Value
Age	20
Sex	female
Class	first
Ticket price	300\$
More attributes	...
Survived	true

And the corresponding anchors explanation is:

IF SEX = female AND Class = first THEN PREDICT Survived = true WITH PRECISION 97% AND COVERAGE 15%

The example shows how anchors can provide essential insights into a model's prediction and its underlying reasoning. The result shows which attributes were taken into account by the model, which in this case, are female and first class. Humans, being paramount for correctness, can use this rule to validate the model's behavior. The anchor additionally tells us that it applies to 15% of the perturbation space's instances. In those cases, the explanation is 97% accurate, meaning the displayed predicates are almost exclusively responsible for the predicted outcome.

An anchor  $A$  is formally defined as follows:

$$\mathbb{E}_{\mathcal{D}_x(\mathbf{z}|A)}[1_{\hat{f}(\mathbf{x})=\hat{f}(\mathbf{z})}] \geq \tau; A(\mathbf{x}) = 1$$

Wherein:

- $\mathbf{x}$  represents the instance being explained (e.g. one row in a tabular dataset).
- $A$  is a set of predicates, i.e., the resulting rule or anchor, such that  $A(\mathbf{x}) = 1$  when all feature predicates defined by  $A$  correspond to  $\mathbf{x}$ 's feature values.
- $\hat{f}$  denotes the classification model to be explained (e.g. an artificial neural network model). It can be queried to predict a label for  $\mathbf{x}$  and its perturbations.
- $\mathcal{D}_x(\cdot|A)$  indicates the distribution of neighbors of  $\mathbf{x}$ , matching  $A$ .
- $0 \leq \tau \leq 1$  specifies a precision threshold. Only rules that achieve a local fidelity of at least  $\tau$  are considered a valid result.

The formal description may be intimidating and can be put in words:

Given an instance  $\mathbf{x}$  to be explained, a rule or an anchor  $A$  is to be found, such that it applies to  $\mathbf{x}$ , while the same class as for  $\mathbf{x}$  gets predicted for a fraction of at least  $\tau$  of  $\mathbf{x}$ 's neighbors where the same  $A$  is applicable. A rule's precision results from evaluating neighbors or perturbations (following  $\mathcal{D}_x(\mathbf{z}|A)$ ) using the provided machine learning model (denoted by the indicator function  $1_{\hat{f}(\mathbf{x})=\hat{f}(\mathbf{z})}$ ).

#### Carefully set precision

Set the precision threshold  $\tau$  wisely: A higher  $\tau$  ensures stronger rules, but may reduce coverage. Experiment with different values to balance precision and coverage.

## 16.1 Finding anchors

Although anchors' mathematical description may seem clear and straightforward, constructing particular rules is infeasible. It would require evaluating  $1_{\hat{f}(\mathbf{x})=\hat{f}(\mathbf{z})}$  for all  $\mathbf{z} \in \mathcal{D}_x(\cdot|A)$ , which is not possible in continuous or large input spaces. Therefore, the authors propose to introduce the parameter  $0 \leq \delta \leq 1$  to create a probabilistic definition. This way, samples are drawn until there is statistical confidence concerning their precision. The probabilistic definition reads as follows:

$$\mathbb{P}(prec(A) \geq \tau) \geq 1 - \delta \quad \text{with} \quad prec(A) = \mathbb{E}_{\mathcal{D}_x(\mathbf{z}|A)}[1_{\hat{f}(\mathbf{x})=\hat{f}(\mathbf{z})}]$$

The previous two definitions are combined and extended by the notion of coverage. Its rationale consists of finding rules that apply to a preferably large part of the model's input space.

Coverage is formally defined as an anchor's probability of applying to its neighbors, i.e., its perturbation space:

$$\text{cov}(A) = \mathbb{E}_{\mathcal{D}_{(\mathbf{x})}}[A(\mathbf{z})]$$

Including this element leads to the anchor's final definition taking into account the maximization of coverage:

$$\max_{A \text{ s.t. } \mathbb{P}(\text{prec}(A) \geq \tau) \geq 1 - \delta} \text{cov}(A)$$

Thus, the proceeding strives for a rule that has the highest coverage among all eligible rules (all those that satisfy the precision threshold given the probabilistic definition). These rules are thought to be more important, as they describe a larger part of the model. Note that rules with more predicates tend to have higher precision than rules with fewer predicates. In particular, a rule that fixes every feature of  $\mathbf{x}$  reduces the evaluated neighborhood to identical instances. Thus, the model classifies all neighbors equally, and the rule's precision is 1. At the same time, a rule that fixes many features is overly specific and only applicable to a few instances. Hence, there is a *trade-off between precision and coverage*.

The anchors approach uses four main components to find explanations.

**Candidate Generation:** Generates new explanation candidates. In the first round, one candidate per feature of  $\mathbf{x}$  gets created and fixes the respective value of possible perturbations. In every other round, the best candidates of the previous round are extended by one feature predicate that is not yet contained therein.

**Best Candidate Identification:** Candidate rules are to be compared in regard to which rule explains  $\mathbf{x}$  the best. To this end, perturbations that match the currently observed rule are created and evaluated by calling the model. However, these calls need to be minimized to limit computational overhead. This is why, at the core of this component, there is a pure-exploration Multi-Armed Bandit (*MAB*). To be more precise, it's *KL-LUCB* by Kaufmann and Kalyanakrishnan (2013). MABs are used to efficiently explore and exploit different strategies (called arms in an analogy to slot machines) using sequential selection. In the given setting, each candidate rule is to be seen as an arm that can be pulled. Each time it is pulled, respective neighbors get evaluated, and we thereby obtain more information about the candidate rule's payoff (precision in the anchor's case). The precision thus states how well the rule describes the instance to be explained.

**Candidate Precision Validation:** Takes more samples in case there is no statistical confidence yet that the candidate exceeds the  $\tau$  threshold.

**Modified Beam Search:** All of the above components are assembled in a beam search, which is a graph search algorithm and a variant of the breadth-first algorithm. It carries the  $B$  best candidates of each round over to the next one (where  $B$  is called the *Beam Width*).

These  $B$  best rules are then used to create new rules. The beam search conducts at most  $\text{featureCount}(\mathbf{x})$  rounds, as each feature can only be included in a rule at most once. Thus, at every round  $i$ , it generates candidates with exactly  $i$  predicates and selects the  $B$  best thereof. Therefore, by setting  $B$  high, the algorithm is more likely to avoid local optima. In turn, this requires a high number of model calls and thereby increases the computational load.

These four components are shown in Figure 16.2.

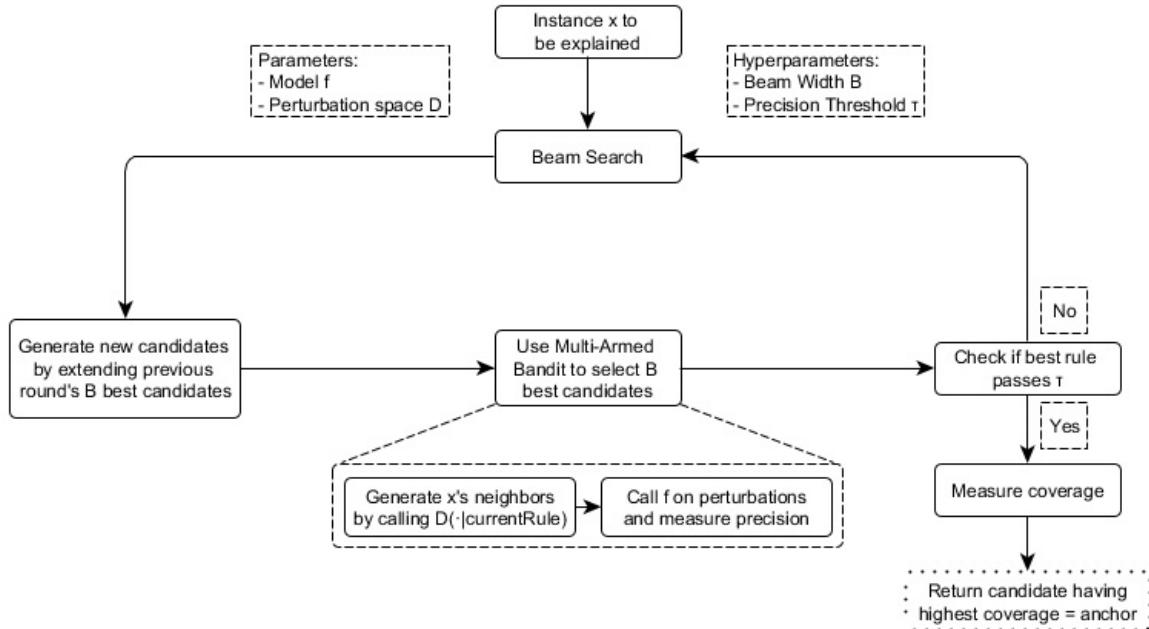


Figure 16.2: The anchors algorithm's components and their interrelations (simplified)

The approach is a seemingly perfect recipe for efficiently deriving statistically sound information about why any system classified an instance the way it did. It systematically experiments with the model's input and concludes by observing respective outputs. It relies on well-established and researched Machine Learning methods (MABs) to reduce the number of calls made to the model. This, in turn, drastically reduces the algorithm's runtime.

## 16.2 Complexity and runtime

Knowing the anchors approach's asymptotic runtime behavior helps to evaluate how well it is expected to perform on specific problems. Let  $B$  denote the beam width and  $p$  the number of all features. Then the anchors algorithm is subject to:

$$\mathcal{O}(B \cdot p^2 + p^2 \cdot \mathcal{O}_{\text{MAB}[B \cdot p, B]})$$

This boundary abstracts from problem-independent hyperparameters, such as the statistical confidence  $\delta$ . Ignoring hyperparameters helps reduce the boundary's complexity (see original paper for more info). Since the MAB extracts the  $B$  best out of  $B \cdot p$  candidates in each round, most MABs and their runtimes multiply the  $p^2$  factor more than any other parameter.

It thus becomes apparent: the algorithm's efficiency decreases when many features are present.

### 16.3 Tabular data example

Tabular data is structured data represented by tables, wherein columns embody features and rows instances. For instance, we use the [bike rental data](#) to demonstrate the anchors approach's potential to explain machine learning predictions for selected instances. For this, we turn the regression into a classification problem and train a random forest as our black box model. It's to classify whether the number of rented bikes lies above or below the trend line. We also use additional features like the number of days since 2011 as a time trend, the month, and the weekday.

Before creating anchor explanations, one needs to define a perturbation function. An easy way to do so is to use an intuitive default perturbation space for tabular explanation cases, which can be built by sampling from, e.g., the training data. When perturbing an instance, this default approach maintains the features' values that are subject to the anchors' predicates, while replacing the non-fixed features with values taken from another randomly sampled instance with a specified probability. This process yields new instances that are similar to the explained one but have adopted some values from other random instances. Thus, they resemble neighbors of the explained instance.

The results (Figure 16.3) are instinctively interpretable and show, for each explained instance, which features are most important for the model's prediction. As the anchors only have a few predicates, they additionally have high coverage and hence apply to other cases. The rules shown above were generated with  $\tau = 0.9$ . Thus, we ask for anchors whose evaluated perturbations faithfully support the label with an accuracy of at least 90%. Also, discretization was used to increase the expressiveness and applicability of numerical features.

All of the previous rules were generated for instances where the model decides confidently based on a few features. However, other instances are not as distinctly classified by the model, as more features are of importance. In such cases, anchors get more specific, comprise more features, and apply to fewer instances, as for example for Figure 16.4.

While choosing the default perturbation space is a comfortable choice to make, it may have a great impact on the algorithm and can thus lead to biased results. For example, if the training

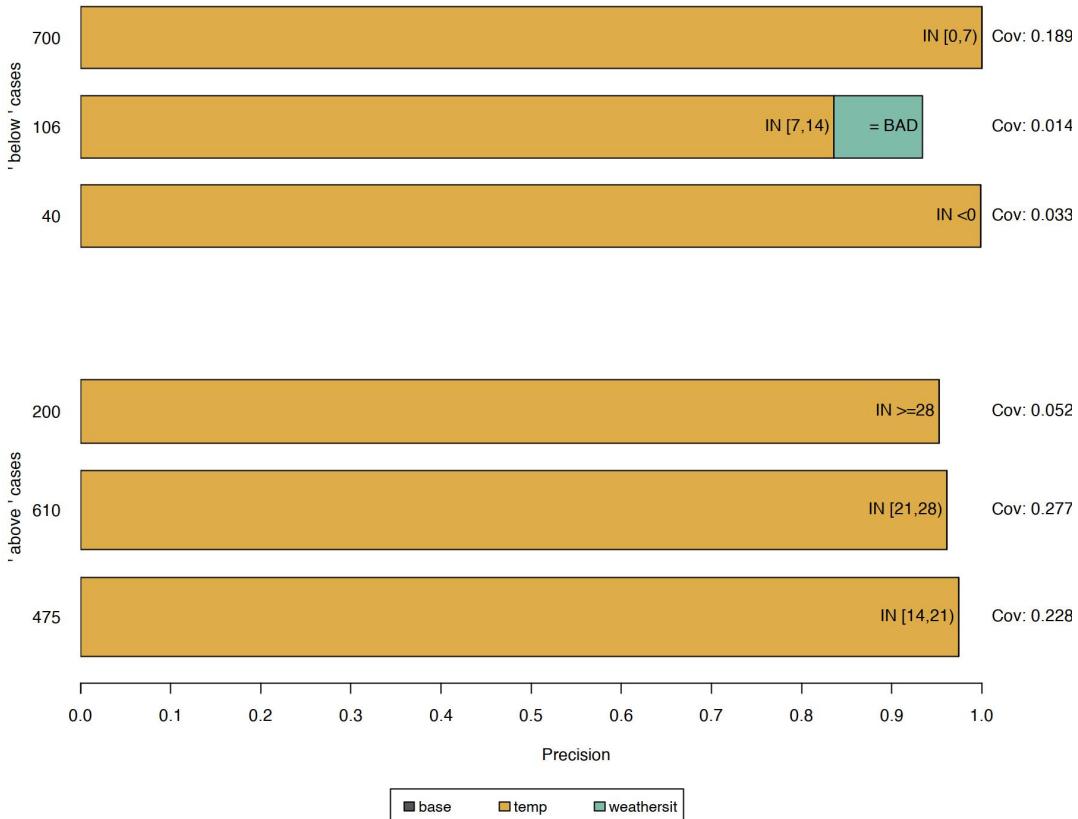


Figure 16.3: Anchors explaining six instances of the bike rental dataset. Each row represents one explanation or anchor, and each bar depicts the feature predicates contained by it. The x-axis displays a rule’s precision, and a bar’s thickness corresponds to its coverage. The “base” rule contains no predicates. These anchors show that the model mainly considers the temperature for predictions.

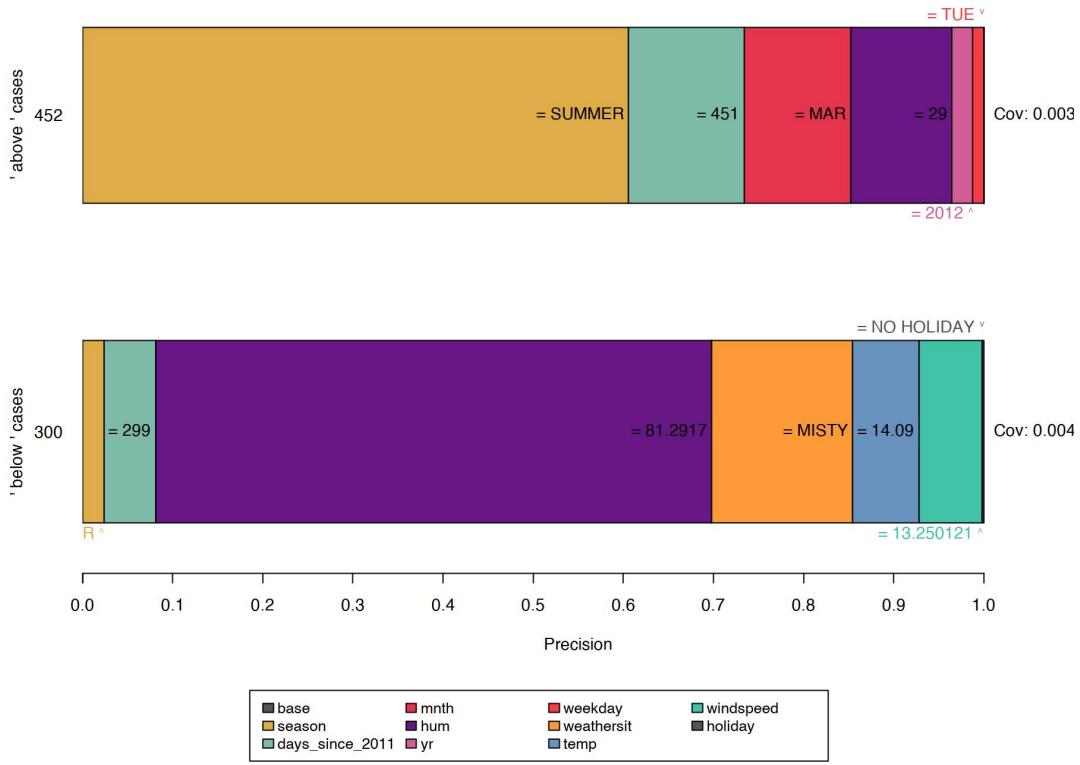


Figure 16.4: Explaining instances near decision boundaries leads to specific rules comprising a higher number of feature predicates and lower coverage. Also, the empty rule, i.e., the base feature, gets less important. This can be interpreted as a signal for a decision boundary, as the instance is located in a volatile neighborhood.

set is unbalanced (there is an unequal number of instances of each class), the perturbation space is as well. This condition further affects the rule-finding and the result's precision. This may be unwanted and can be approached in multiple ways. For example, a custom perturbation space can be defined. This custom perturbation can sample differently, e.g., from an unbalanced data set or a normal distribution. This, however, comes with a side effect: the sampled neighbors are not representative and change the coverage's scope. Alternatively, we could modify the MAB's confidence  $\delta$  and error parameter values  $\epsilon$ . This would cause the MAB to draw more samples, ultimately leading to the minority being sampled more often in absolute terms.

#### 💡 Check for class imbalance

When defining a custom perturbation space, consider sampling strategies that respect the class imbalance. For example, use stratified sampling to ensure that minority classes are appropriately represented in the perturbation space.

## 16.4 Strengths

The anchors approach offers multiple advantages over LIME. First, the algorithm's output is easier to understand, as rules are **easy to interpret** (even for laypersons).

Furthermore, **anchors are subsettable** and even state a measure of importance by including the notion of coverage. Second, the anchors approach **works when model predictions are non-linear or complex** in an instance's neighborhood. As the approach deploys reinforcement learning techniques instead of fitting surrogate models, it is less likely to underfit the model.

Apart from that, the algorithm is **model-agnostic** and thus applicable to any model.

Anchors can be useful for justifying predictions as they **show the robustness of predictions** (or lack thereof) to changes in certain features.

Furthermore, it is **highly efficient as it can be parallelized** by making use of MABs that support batch sampling (e.g. BatchSAR).

## 16.5 Limitations

The algorithm suffers from a **highly configurable** and impactful setup, just like most perturbation-based explainers. Not only do hyperparameters such as the beam width or precision threshold need to be tuned to yield meaningful results, but also the perturbation function needs to be explicitly designed for one domain/use case. Think of how tabular data get perturbed, and think of how to apply the same concepts to image data (hint: these

cannot be applied). Luckily, default approaches may be used in some domains (e.g., tabular), facilitating an initial explanation setup.

Also, **many scenarios require discretization** as otherwise results are too specific, have low coverage, and do not contribute to understanding the model. While discretization can help, it may also blur decision boundaries if used carelessly and thus have the exact opposite effect. Since there is no best discretization technique, users need to be aware of the data before deciding on how to discretize data, not to obtain poor results.

Constructing anchors requires **many calls to the machine learning model**, just like all perturbation-based explainers. While the algorithm deploys MABs to minimize the number of calls, its runtime still very much depends on the model's performance and is therefore highly variable.

Lastly, the notion of **coverage is undefined in some domains**. For example, there is no obvious or universal definition of how superpixels in one image compare to those in other images.

## 16.6 Software and alternatives

Currently, there are two implementations available: [anchor, a Python package](#) (also integrated by [Alibi](#)), and a [Java implementation](#). The former is the anchors algorithm's authors' reference, and the latter a high-performance implementation which comes with an R interface, called [anchors](#), which was used for the examples in this chapter. By now, the anchors implementation supports tabular data only. However, anchors may theoretically be constructed for any domain or type of data.

# 17 Shapley Values

A prediction can be explained by assuming that each feature value of the instance is a “player” in a game where the prediction is the payout. Shapley values – a method from coalitional game theory – tell us how to fairly distribute the “payout” among the features.

## 17.1 General idea

Assume the following scenario: You’ve 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 an area of  $50 \text{ m}^2$ , is located on the 2nd floor, has a park nearby, and cats are banned, as illustrated in Figure 17.1. The average prediction for all apartments is €310,000. Our goal is to explain how each of these feature values contributed to the prediction. How much has each feature value contributed to the prediction compared to the average prediction?

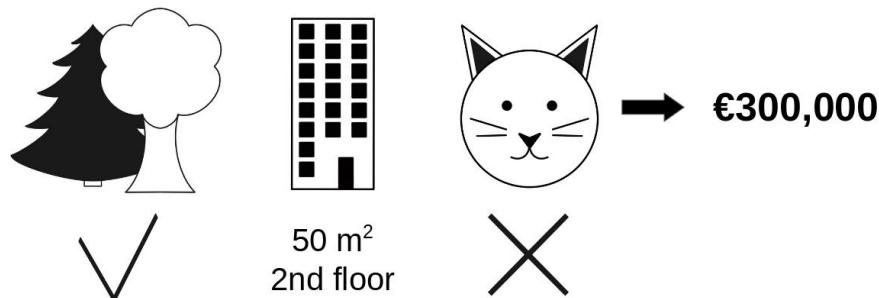


Figure 17.1: The predicted price for a  $50 \text{ m}^2$  2nd floor apartment with a nearby park and cat ban is €300,000.

The answer is simple for linear regression models. The effect of each feature is the weight of the feature times the feature value. This only works because of the linearity of the model. For more complex models, we need a different solution. For example, [LIME](#) suggests local models to estimate effects. Another solution comes from cooperative game theory: The Shapley value, coined by Shapley (1953), is a method for assigning payouts to players depending on their contribution to the total payout. Players cooperate in a coalition and receive a certain profit from this cooperation.

Players? Game? Payout? What's the connection to machine learning predictions and interpretability? The “game” is the prediction task for a single instance of the dataset. The “gain” is the actual prediction for this instance minus the average prediction for all instances. The “players” are the feature values of the instance that collaborate to receive the gain (= predict a certain value). In our apartment example, the feature values `park-nearby`, `cat-banned`, `area-50`, and `floor-2nd` worked together to achieve the prediction of €300,000. Our goal is to explain the difference between the actual prediction (€300,000) and the average prediction (€310,000): a difference of -€10,000.

The answer could be: The `park-nearby` contributed €30,000; `area-50` contributed €10,000; `floor-2nd` contributed €0; `cat-banned` contributed -€50,000. The contributions add up to -€10,000, the final prediction minus the average predicted apartment price.

### How do we calculate the Shapley value for one feature?

The Shapley value is the average marginal contribution of a feature value across all possible coalitions. All clear now?

Figure 17.2 shows how to calculate the marginal contribution of the `cat-banned` feature value when it is added to a coalition of `park-nearby` and `area-50`. We simulate that only `park-nearby`, `cat-banned`, and `area-50` are in a coalition by randomly drawing another apartment from the data and using its value for the floor feature. The value `floor-2nd` was replaced by the randomly drawn `floor-1st`. Then we predict the price of the apartment with this combination (€310,000). In a second step, we remove `cat-banned` from the coalition by replacing it with a random value of the cat allowed/banned feature from the randomly drawn apartment. In the example, it was `cat-allowed`, but it could have been `cat-banned` again. We predict the apartment price for the coalition of `park-nearby` and `area-50` (€320,000). The contribution of `cat-banned` was €310,000 - €320,000 = -€10,000. This estimate depends on the values of the randomly drawn apartment that served as a “donor” for the cat and floor feature values. We'll get better estimates if we repeat this sampling step and average the contributions.

We repeat this computation for all possible coalitions. The Shapley value is the average of all the marginal contributions to all possible coalitions. The computation time increases exponentially with the number of features. One solution to keep the computation time manageable is to compute contributions for only a few samples of the possible coalitions.

Figure 17.3 shows all coalitions of feature values that are needed to determine the exact Shapley value for `cat-banned`. The first row shows the coalition without any feature values. The second, third, and fourth rows show different coalitions with increasing coalition size, separated by “|”. All in all, the following coalitions are possible:

- {} (empty coalition)
- {`park-nearby`}
- {`area-50`}
- {`floor-2nd`}
- {`park-nearby`, `area-50`}

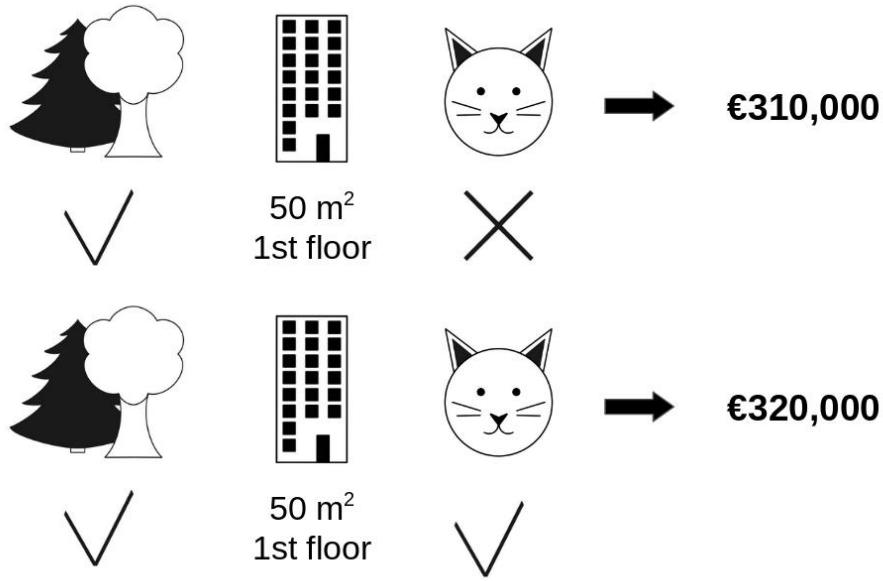


Figure 17.2: One sample repetition to estimate the contribution of `cat-banned` to the prediction when added to the coalition of `park-nearby` and `area-50`.

- {`park-nearby`, `floor-2nd`}
- {`area-50`, `floor-2nd`}
- {`park-nearby`, `area-50`, `floor-2nd`}

For each of these coalitions, we compute the predicted apartment price with and without the feature value `cat-banned` and take the difference to get the marginal contribution. The Shapley value is the (weighted) average of all 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. If we estimate the Shapley values for all feature values, we get the complete distribution of the prediction (minus the average) among the feature values.

## 17.2 Examples and interpretation

The interpretation of the Shapley value for feature  $j$  is: The value of the  $j$ -th feature contributed  $\phi_j$  to the prediction of this particular instance compared to the average prediction for the dataset. The Shapley value works for both classification (if we are dealing with probabilities) and regression.

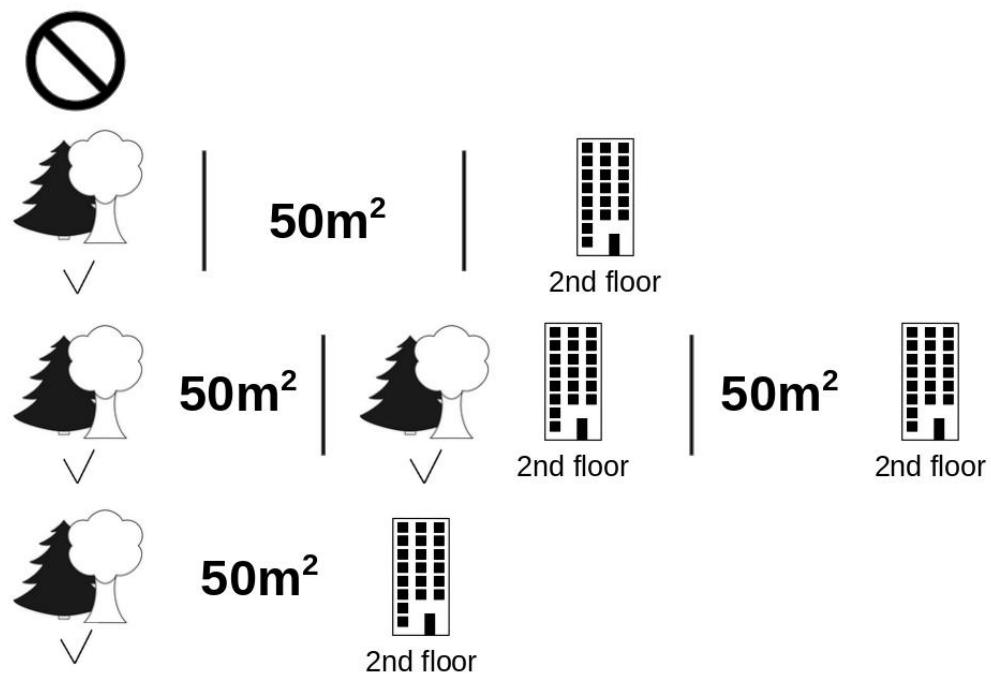


Figure 17.3: All 8 coalitions needed for computing the exact Shapley value of the `cat-banned` feature value

We use the Shapley value to analyze the predictions of a random forest model predicting [penguin sex](#). Figure 17.4 shows the Shapley values for a male penguin. This penguin's predicted  $P(\text{female})=0.21$  is -0.31 below the average probability for  $P(\text{female})$  of 0.51. The bill length contributed most to the probability of being female, but most factors were (correctly) pointing to male. The sum of the contributions yields the difference between actual and average prediction (0.51).

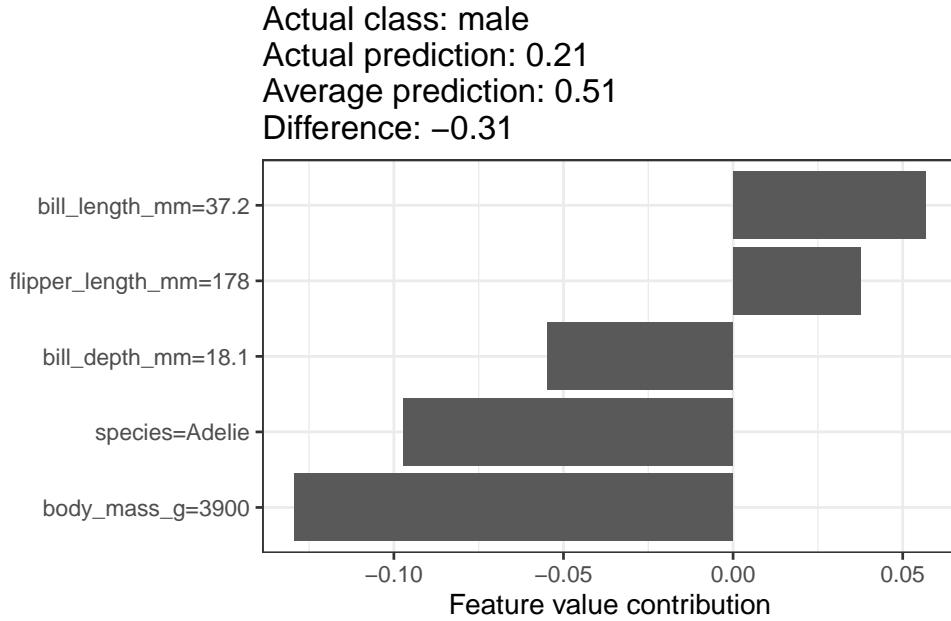


Figure 17.4: Shapley values for a penguin being classified as female.

Shapley values always need a reference dataset from which to sample the missing team members. In this case, I used all of the data points, which means all of the penguins, no matter the species. The current penguin is an Adelie penguin, and we can also compute Shapley values that only compare against other penguins of the same species. This reduces the problem of sampling and combining unrealistic values. Figure 17.5 shows also a different interpretation. When comparing our penguin with other Adelie penguins, then the reason why it has a low probability of being female is its body weight.

 Pick the reference data

Shapley value interpretation is always in reference to the dataset that was used for replacing values for absent players. Make sure to pick a meaningful reference dataset.

For the [bike rental dataset](#), we also train a random forest to predict the number of rented bikes for a day, given weather and calendar information. The explanations created for the random forest prediction of a particular day are shown in Figure 17.6.

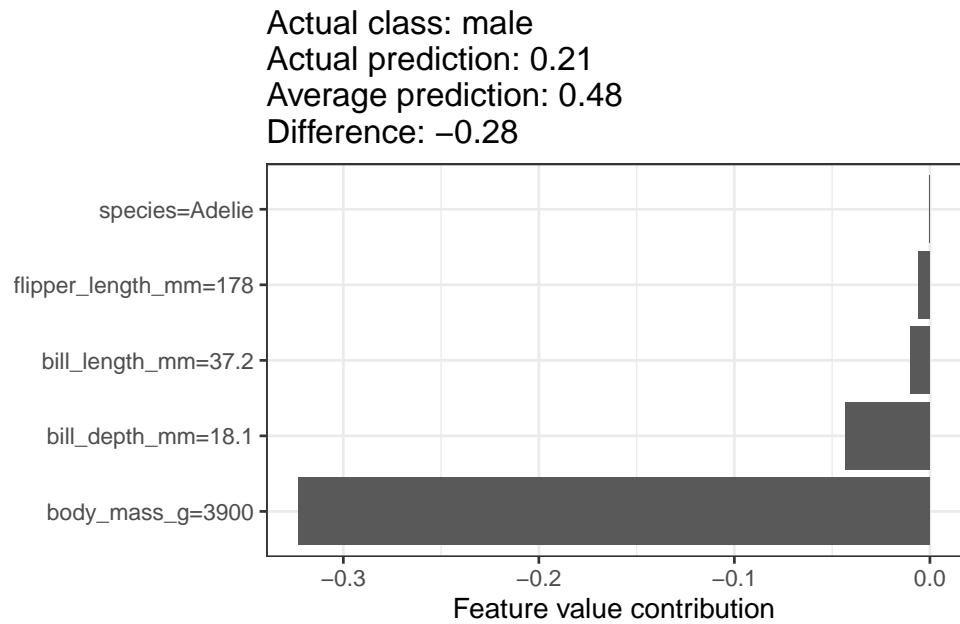


Figure 17.5: Shapley values computed only based on data from the same penguin species.

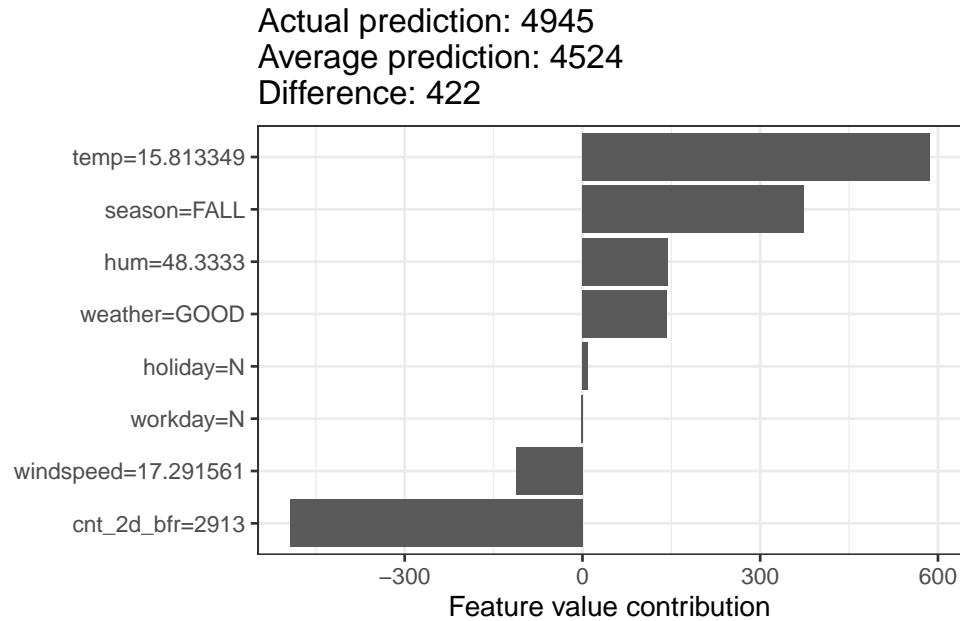


Figure 17.6: Shapley values for day 285 of the bike data.

With a predicted 4945 rental bikes, this day is 422 below the average prediction of 4524. The temperature and humidity had the largest positive contributions. The low count two days before had the largest negative contribution. The sum of Shapley values yields the difference of actual and average prediction (422).

**⚠️ Shapley values are not counterfactuals**

Be careful to interpret the Shapley value correctly: The Shapley value is the average contribution of a feature value to the prediction in different coalitions. The Shapley value is NOT the difference in prediction when we remove the feature from the model.

### 17.3 Shapley value theory

This section goes deeper into the definition and computation of the Shapley value for the curious reader. Skip this section and go directly to “Strengths and Limitations” if you are not interested in the technical details.

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

$$\hat{f}(\mathbf{x}) = \beta_0 + \beta_1 x_1 + \dots + \beta_p x_p$$

where  $\mathbf{x}$  is the instance for which we want to compute the contributions. Each  $x_j$  is a feature value, with  $j = 1, \dots, p$ . The  $\beta_j$  is the weight corresponding to feature  $j$ .

The contribution  $\phi_j$  of the  $j$ -th feature to the prediction  $\hat{f}(\mathbf{x})$  is:

$$\phi_j(\hat{f}) = \beta_j x_j - \beta_j \mathbb{E}[X_j]$$

where  $\beta_j \mathbb{E}[X_j]$  is the mean effect estimate for feature  $j$ . The contribution is the difference between the feature effect and the average effect. Nice! Now we know how much each feature contributed to the prediction. If we sum all the feature contributions for one instance, the result is the following:

$$\begin{aligned}
\sum_{j=1}^p \phi_j(\hat{f}) &= \sum_{j=1}^p (\beta_j x_j - \beta_j \mathbb{E}[X_j]) \\
&= (\beta_0 + \sum_{j=1}^p \beta_j x_j) - (\beta_0 + \sum_{j=1}^p \beta_j \mathbb{E}[X_j]) \\
&= \hat{f}(\mathbf{x}) - \mathbb{E}[\hat{f}(\mathbf{X})]
\end{aligned}$$

This is the predicted value for the data point  $\mathbf{x}$  minus the average predicted value. Feature contributions can be negative.

Can we do the same for any type of model? It would be great to have this as a model-agnostic tool. Since we usually do not have similar weights in other model types, we need a different solution.

Help comes from unexpected places: cooperative game theory. The Shapley value is a solution for computing feature contributions for single predictions for any machine learning model.

### 17.3.1 Definition

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

The Shapley value of a feature value is its contribution to the payout, weighted and summed over all possible feature value combinations:

$$\phi_j(val) = \sum_{S \subseteq \{1, \dots, p\} \setminus \{j\}} \frac{|S|! (p - |S| - 1)!}{p!} (val(S \cup \{j\}) - val(S))$$

where  $S$  is a subset of the features used in the model,  $\mathbf{x}$  is the vector of feature values of the instance to be explained, and  $p$  is the number of features.  $val_{\mathbf{x}}(S)$  is the prediction for feature values in set  $S$  that are marginalized over features  $X_C$ , which are all the features not included in  $S$  ( $S \cap C = \emptyset$  and  $S \cup C = \{1, \dots, p\}$ ):

$$val_{\mathbf{x}}(S) = \int \hat{f}(x_1, \dots, x_p) d\mathbb{P}_{X_C} - \mathbb{E}[\hat{f}(\mathbf{X})]$$

You actually perform multiple integrations for each feature that is not contained in  $S$ . A concrete example: The machine learning model works with 4 features  $X_1$ ,  $X_2$ ,  $X_3$ , and  $X_4$ , and we evaluate the prediction for the coalition  $S$  consisting of feature values  $x_1$  and  $x_3$ :

$$val_{\mathbf{x}}(S) = val_{\mathbf{x}}(\{1, 3\}) = \int_{\mathbb{R}} \int_{\mathbb{R}} \hat{f}(x_1, X_2, x_3, X_4) d\mathbb{P}_{X_2 X_4} - \mathbb{E}[\hat{f}(\mathbf{X})]$$

This looks similar to the feature contributions in the linear model!

**i** Don't get confused by the many uses of the word "value"

The feature value is the numerical or categorical value of a feature and instance; the Shapley value is the feature contribution to the prediction; the value function is the payout function for coalitions of players (feature values).

The Shapley value is the only attribution method that satisfies the properties **Efficiency**, **Symmetry**, **Dummy**, and **Additivity**, which together can be considered a definition of a fair payout.

**Efficiency** The feature contributions must add up to the difference of prediction for  $\mathbf{x}$  and the average.

$$\sum_{j=1}^p \phi_j = \hat{f}(\mathbf{x}) - \mathbb{E}[\hat{f}(\mathbf{X})]$$

**Symmetry** The contributions of two feature values  $j$  and  $k$  should be the same if they contribute equally to all possible coalitions. If

$$val(S \cup \{j\}) = val(S \cup \{k\})$$

for all

$$S \subseteq \{1, \dots, p\} \setminus \{j, k\}$$

then

$$\phi_j = \phi_k$$

**Dummy** A feature  $j$  that does not change the predicted value – regardless of which coalition of feature values it is added to – should have a Shapley value of 0. If

$$val(S \cup \{j\}) = val(S)$$

for all

$$S \subseteq \{1, \dots, p\}$$

then

$$\phi_j = 0$$

**Additivity** For a game with combined payouts  $val + val^+$  the respective Shapley values are as follows:

$$\phi_j + \phi_j^+$$

Suppose you trained a random forest, which means that the prediction is an average of many decision trees. The Additivity property guarantees that for a feature value, you can calculate the Shapley value for each tree individually, average them, and get the Shapley value for the feature value for the random forest.

### i Intuitive understanding of Shapley values

The feature values enter a room in random order. All feature values in the room participate in the game (= contribute to the prediction). The Shapley value of a feature value is the average change in the prediction that the coalition already in the room receives when the feature value joins them.

## 17.4 Estimating Shapley values

All possible coalitions (sets) of feature values have to be evaluated with and without the  $j$ -th feature to calculate the exact Shapley value. For more than a few features, the exact solution to this problem becomes problematic as the number of possible coalitions exponentially increases as more features are added. Štrumbelj and Kononenko (2014) proposed an approximation with Monte-Carlo sampling:

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

where  $\hat{f}(\mathbf{x}_{+j}^{(m)})$  is the prediction for  $\mathbf{x}$ , but with a random number of feature values replaced by feature values from a random data point  $\mathbf{z}$ , except for the respective value of feature  $j$ . The feature vector  $\mathbf{x}_{-j}^{(m)}$  is almost identical to  $\mathbf{x}_{+j}^{(m)}$ , but the value  $x_j^{(m)}$  is also taken from the sampled  $\mathbf{z}$ . Each of these  $M$  new instances is a kind of “Frankenstein’s Monster” assembled from two instances. Note that in the following algorithm, the order of features is not actually changed – each feature remains at the same vector position when passed to the predict function. The order is only used as a “trick” here: By giving the features a new order, we get a random mechanism that helps us put together the “Frankenstein’s Monster.” For features that appear

left of the feature  $X_j$ , we take the values from the original observations, and for the features on the right, we take the values from a random instance.

### Approximate Shapley estimation for a single feature value:

- Output: Shapley value for the value of the  $j$ -th feature
- Required: Number of iterations  $M$ , instance of interest  $\mathbf{x}$ , feature index  $j$ , data matrix  $\mathbf{X}$ , and machine learning model  $f$
- For all  $m = 1, \dots, M$ :
  - Draw random instance  $\mathbf{z}$  from the data matrix  $\mathbf{X}$
  - Choose a random permutation  $\mathbf{o}$  of the feature values
  - Order instance  $\mathbf{x}$ :  $\mathbf{x}_{\mathbf{o}} = (x_{(1)}, \dots, x_{(j)}, \dots, x_{(p)})$
  - Order instance  $\mathbf{z}$ :  $\mathbf{z}_{\mathbf{o}} = (z_{(1)}, \dots, z_{(j)}, \dots, z_{(p)})$
  - Construct two new instances
    - \* With  $j$ :  $\mathbf{x}_{+j} = (x_{(1)}, \dots, x_{(j-1)}, x_{(j)}, z_{(j+1)}, \dots, z_{(p)})$
    - \* Without  $j$ :  $\mathbf{x}_{-j} = (x_{(1)}, \dots, x_{(j-1)}, z_{(j)}, z_{(j+1)}, \dots, z_{(p)})$
  - Compute marginal contribution:  $\phi_j^{(m)} = \hat{f}(\mathbf{x}_{+j}) - \hat{f}(\mathbf{x}_{-j})$
- Compute Shapley value as the average:  $\phi_j(\mathbf{x}) = \frac{1}{M} \sum_{m=1}^M \phi_j^{(m)}$

 Reduce sample size for speed

Consider using a smaller sample size  $M$  when estimating Shapley values to reduce computation time, but beware of increased variance in the estimate.

The procedure has to be repeated for each of the features to get all Shapley values. In the [SHAP chapter](#), we will see other efficient ways to estimating Shapley values.

## 17.5 Strengths

The difference between the prediction and the average prediction is **fairly distributed** among the feature values of the instance – the Efficiency property of Shapley values. This property distinguishes the Shapley value from other methods such as [LIME](#). LIME does not guarantee that the prediction is fairly distributed among the features. The Shapley values deliver a **full explanation**.

The Shapley value allows **contrastive explanations**. Instead of comparing a prediction to the average prediction of the entire dataset, you could compare it to a subset or even to a single data point. This contrastiveness is also something that local models like LIME do not have.

The Shapley value is the only explanation method with a **solid theory**. The axioms – efficiency, symmetry, dummy, additivity – give the explanation a reasonable foundation. Methods like LIME assume linear behavior of the machine learning model locally, but there is no theory as to why this should work.

## 17.6 Limitations

The Shapley value requires **a lot of computing time**. In 99.9% of real-world problems, only the approximate solution is feasible. An exact computation of the Shapley value is computationally expensive because there are  $2^k$  possible coalitions of the feature values and the “absence” of a feature has to be simulated by drawing random instances, which increases the variance for the estimate of the Shapley values estimation. The exponential number of the coalitions is dealt with by sampling coalitions and limiting the number of iterations M. Decreasing M reduces computation time, but increases the variance of the Shapley value. There’s no good rule of thumb for the number of iterations M. M should be large enough to accurately estimate the Shapley values, but small enough to complete the computation in a reasonable time. It should be possible to choose M based on Chernoff bounds, but I have not seen any paper on doing this for Shapley values for machine learning predictions.

The Shapley value **can be misinterpreted**. The Shapley value of a feature value is not the difference of the predicted value after removing the feature from the model training. The interpretation of the Shapley value is: Given the current set of feature values, the contribution of a feature value to the difference between the actual prediction and the mean prediction is the estimated Shapley value.

Shapley value explanations **are not to be interpreted as local in the sense of gradients or neighborhood**. (Bilodeau et al. 2024) For example, a positive Shapley value doesn’t mean that increasing the feature would increase the prediction. Instead, the Shapley value has to be interpreted with respect to the reference dataset that was used for the estimation. That’s why I would also recommend pairing Shapley values with, for example, [ceteris paribus plots](#) or [ICE plots](#), so you get the full picture.

💡 Combine with ceteris paribus and ICE

Pair Shapley values with ceteris paribus or ICE plots to get a sense of local sensitivity to feature changes.

The Shapley value is the wrong explanation method if you seek sparse explanations (explanations that contain few features). Explanations created with the Shapley value method **use all the features**. Humans prefer selective explanations, such as those produced by LIME. LIME might be the better choice for explanations laypersons have to deal with. Another solution is

[SHAP](#) introduced by Lundberg and Lee (2017), which is based on the Shapley value but can also provide explanations with few features.

The Shapley value returns a simple value per feature, but **no prediction model** like LIME. This means it cannot be used to make statements about changes in prediction for changes in the input, such as: “If I were to earn €300 more a year, my credit score would increase by 5 points.”

Like many other permutation-based interpretation methods, the Shapley value method suffers from **inclusion of unrealistic data instances** when features are correlated. To simulate that a feature value is missing from a coalition, we marginalize the feature. This is achieved by sampling values from the feature’s marginal distribution. This is fine as long as the features are independent. When features are dependent, then we might sample feature values that do not make sense for this instance. But we would use those to compute the feature’s Shapley value. One solution might be to permute correlated features together and get one mutual Shapley value for them. Another adaptation is conditional sampling: Features are sampled conditional on the features that are already in the team. While conditional sampling fixes the issue of unrealistic data points, a new issue is introduced: The resulting values are no longer the Shapley values to our game, since they violate the symmetry axiom, as found out by Sundararajan and Najmi (2020) and further discussed by Janzing, Minorics, and Blöbaum (2020).

## 17.7 Software and alternatives

Shapley values are implemented in both the `iml` and `fastshap` packages for R. In Julia, you can use [Shapley.jl](#).

SHAP, an alternative estimation method and ecosystem for Shapley values, is presented in the [next chapter](#).

Another approach is called `breakDown`, which is implemented in the `breakDown` R package (Staniak and Biecek 2018). `BreakDown` also shows the contributions of each feature to the prediction but computes them step by step. Let’s reuse the game analogy: We start with an empty team, add the feature value that would contribute the most to the prediction and iterate until all feature values are added. How much each feature value contributes depends on the respective feature values that are already in the “team”, which is the big drawback of the `breakDown` method. It’s faster than the Shapley value method, and for models without interactions, the results are the same.

# 18 SHAP

SHAP (SHapley Additive exPlanations) by Lundberg and Lee (2017) is a method to explain individual predictions. SHAP is based on the game-theoretically optimal Shapley values. I recommend reading the chapter on [Shapley values](#) first.

To understand why SHAP is a thing and not just an extension of the [Shapley values chapter](#), a bit of history: In 1953, Lloyd Shapley introduced the concept of Shapley values for game theory. Shapley values for explaining machine learning predictions were suggested for the first time by Štrumbelj and Kononenko (2011) and Štrumbelj and Kononenko (2014). However, they didn't become so popular. A few years later, Lundberg and Lee (2017) proposed SHAP, which was basically a new way to estimate Shapley values for interpreting machine learning predictions, along with a theory connecting Shapley values with [LIME](#) and other post-hoc attribution methods, and a bit of additional theory on Shapley values.

You might say that SHAP is just a rebranding of Shapley values (which is true), but that would miss the fact that SHAP also marks a change in popularity and usage of Shapley values, and introduced new ways of estimating Shapley values and aggregating them in new ways. Also, SHAP brought Shapley values to text and image models.

While this chapter could totally be a subchapter of the [Shapley value chapter](#), you can also see it as Shapley values 2.0 for interpreting machine learning models. This chapter emphasizes the new ways to estimate Shapley values and the new types of plots. But first, a bit of theory.

## 18.1 SHAP theory

The goal of SHAP is to explain the prediction of an instance  $\mathbf{x}$  by computing the contribution of each feature to the prediction. SHAP computes Shapley values from coalitional game theory, the same as we discussed in [the Shapley value chapter](#). The feature values of a data instance act as players in a coalition. Shapley values tell us how to fairly distribute the “payout” (= the prediction) among the features. A player can be an individual feature value, e.g., for tabular data. A player can also be a group of feature values. For example, to explain an image, pixels can be grouped into superpixels, and the prediction distributed among them. One innovation that SHAP brings to the table is that the Shapley value explanation is represented as an additive feature attribution method, a linear model. That view connects LIME and Shapley values. SHAP specifies the explanation as:

$$g(\mathbf{z}') = \phi_0 + \sum_{j=1}^M \phi_j z'_j$$

where  $g$  is the explanation model,  $\mathbf{z}' = (z'_1, \dots, z'_M)^T \in \{0, 1\}^M$  is the coalition vector,  $M$  is the maximum coalition size, and  $\phi_j \in \mathbb{R}$  is the feature attribution for feature  $j$ , the Shapley values. What I call “coalition vector” is called “simplified features” in the SHAP paper. I think this name was chosen because, for e.g., image data, the images are not represented on the pixel level, but aggregated into superpixels. It’s helpful to think about the  $\mathbf{z}'$  as describing coalitions: In the coalition vector, an entry of 1 means that the corresponding feature value is “present” and 0 that it is “absent”. But again, a feature here can be different from features the model used, like superpixels instead of pixels. The important part is that we can map between the two representations. To compute Shapley values, we simulate that only some feature values are playing (“present”) and some are not (“absent”). The representation as a linear model of coalitions is a trick for the computation of the  $\phi_j$ ’s. For  $\mathbf{x}$ , the instance of interest, the coalition vector  $\mathbf{x}'$  is a vector of all 1’s, i.e. all feature values are “present”. The formula simplifies to:

$$g(\mathbf{x}') = \phi_0 + \sum_{j=1}^M \phi_j$$

You can find this formula in similar notation in the [Shapley value](#) chapter. More about the actual estimation comes later in this chapter. Let’s first talk about the properties of the  $\phi$ ’s before we go into the details of their estimation.

Shapley values are the only solution that satisfies properties of Efficiency, Symmetry, Dummy, and Additivity. SHAP also satisfies these since it computes Shapley values. In the SHAP paper by Lundberg and Lee (2017), you will find discrepancies between SHAP properties and Shapley properties. SHAP describes the following three desirable properties:

### 1) Local accuracy

$$\hat{f}(\mathbf{x}) = g(\mathbf{x}') = \phi_0 + \sum_{j=1}^M \phi_j x'_j$$

If you define  $\phi_0 = \mathbb{E}[\hat{f}(X)]$  and set all  $x'_j$  to 1, this is the Shapley efficiency property. Only with a different name and using the coalition vector.

$$\hat{f}(\mathbf{x}) = \phi_0 + \sum_{j=1}^M \phi_j x'_j = \mathbb{E}[\hat{f}(X)] + \sum_{j=1}^M \phi_j$$

## 2) Missingness

$$x'_j = 0 \Rightarrow \phi_j = 0$$

Missingness says that a missing feature gets an attribution of zero. Note that  $x'_j$  refers to the coalitions where a value of 0 represents the absence of a feature value. In coalition notation, all feature values  $x'_j$  of the instance to be explained should be ‘1’. This property is not among the properties of the “normal” Shapley values. So why do we need it for SHAP? Lundberg calls it a “[minor book-keeping property](#)”. A missing feature could – in theory – have an arbitrary Shapley value without hurting the local accuracy property, since it is multiplied with  $x'_j = 0$ . The Missingness property enforces that missing features get a Shapley value of 0. In practice, this is only relevant for features that are constant.

## 3) Consistency

Let  $\hat{f}_{\mathbf{x}}(\mathbf{z}') = \hat{f}(h_{\mathbf{x}}(\mathbf{z}'))$  and  $\mathbf{z}'_{-j}$  indicate that  $z'_j = 0$ . For any two models  $\hat{f}$  and  $\hat{f}'$  that satisfy:

$$\hat{f}'_{\mathbf{x}}(\mathbf{z}') - \hat{f}'_{\mathbf{x}}(\mathbf{z}'_{-j}) \geq \hat{f}_{\mathbf{x}}(\mathbf{z}') - \hat{f}_{\mathbf{x}}(\mathbf{z}'_{-j})$$

for all inputs  $\mathbf{z}' \in \{0, 1\}^M$ , then:

$$\phi_j(\hat{f}', \mathbf{x}) \geq \phi_j(\hat{f}, \mathbf{x})$$

The notation  $\phi_j(\hat{f}, \mathbf{x})$  emphasizes which model and data point the Shapley values depend on. The consistency property says that if a model changes so that the marginal contribution of a feature value increases or stays the same (regardless of other features), the Shapley value also increases or stays the same. From Consistency, the Shapley properties Linearity, Dummy, and Symmetry follow, as described in the [Supplemental](#) of Lundberg and Lee (2017).

## 18.2 SHAP estimation

This section is about three ways to estimate Shapley values for explaining predictions: KernelSHAP, Permutation Method, and TreeSHAP.

### 18.2.1 KernelSHAP

The situation with KernelSHAP is a bit confusing: It has been the entire motivation for SHAP, linked it with LIME and other attribution methods, and was introduced in the original paper Lundberg and Lee (2017). Also, many blog posts and so on about SHAP talk about KernelSHAP. But KernelSHAP is slow compared to TreeSHAP and the Permutation Method, and that's why the `shap` Python package no longer uses it. Even if no longer the default, KernelSHAP helps to understand Shapley values and shows how Shapley values and LIME are connected, and some implementations still use it.

The KernelSHAP estimation has five steps:

- Sample coalition vectors  $\mathbf{z}'_k \in \{0, 1\}^M$ ,  $k \in \{1, \dots, K\}$  ( $1 =$  feature present in coalition,  $0 =$  feature absent).
- Get prediction for each  $\mathbf{z}'_k$  by first converting  $\mathbf{z}'_k$  to the original feature space and then applying model  $\hat{f} : \hat{f}(h_{\mathbf{x}}(\mathbf{z}'_k))$ .
- Compute the weight for each coalition  $\mathbf{z}'_k$  with the SHAP kernel.
- Fit weighted linear model.
- Return Shapley values  $\phi_k$ , the coefficients from the linear model.

We can create a random coalition by repeated coin flips until we have a chain of 0's and 1's. For example, the vector of  $(1, 0, 1, 0)^T$  means that we have a coalition of the first and third features. The  $K$  sampled coalitions become the dataset for the regression model. The target for the regression model is the prediction for a coalition. (“Hold on!,” you say. “The model has not been trained on these binary coalition data and cannot make predictions for them.”) To get from coalitions of feature values to valid data instances, we need a function  $h_{\mathbf{x}}(\mathbf{z}') = \mathbf{z}$  where  $h_{\mathbf{x}} : \{0, 1\}^M \rightarrow \mathbb{R}^p$ . The function  $h_{\mathbf{x}}$  maps 1's to the corresponding value from the instance  $\mathbf{x}$  that we want to explain. For tabular data, it maps 0's to the values of another instance that we sample from the data. This means that we equate “feature value is absent” with “feature value is replaced by random feature value from data”. For tabular data, Figure 18.1 visualizes the mapping from coalitions to feature values.

$h_{\mathbf{x}}$  for tabular data treats feature  $X_j$  and  $X_{-j}$  (the other features) as independent and integrates over the marginal distribution:

$$\hat{f}(h_{\mathbf{x}}(\mathbf{z}')) = \mathbb{E}_{X_{-j}}[\hat{f}(x_j, X_{-j})]$$

Sampling from the marginal distribution means ignoring the dependence structure between present and absent features. KernelSHAP therefore suffers from the same problem as all permutation-based interpretation methods. The estimation potentially puts weight on unlikely instances. Results can become unreliable. Would we sample from the conditional distribution, the value function would change, and therefore the game to which Shapley values are the solution. As a result, the Shapley values have a different interpretation: For example, a feature that might not have been used by the model at all can have a non-zero Shapley value when

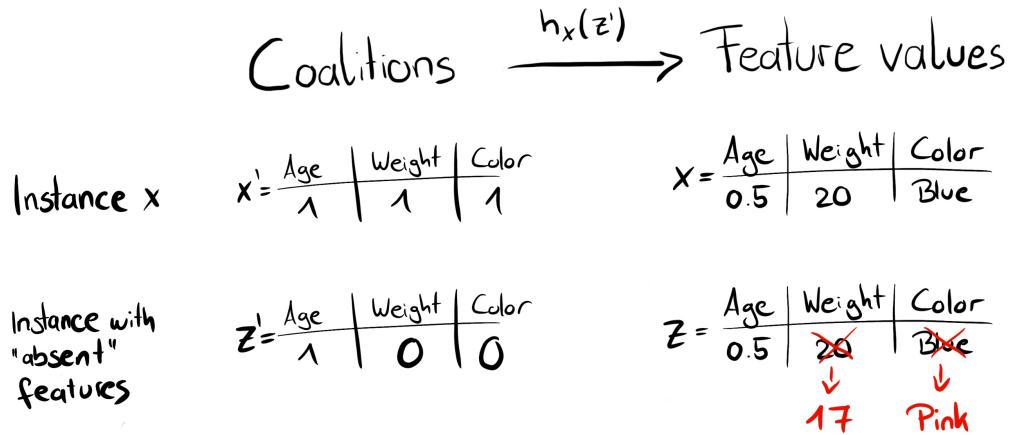


Figure 18.1: Function  $h_x$  maps a coalition to a valid instance. For present features (1),  $h_x$  maps to the feature values of  $x$ . For absent features (0),  $h_x$  maps to the values of a randomly sampled data instance.

the conditional sampling is used. For the marginal game, this feature value would always get a Shapley value of 0, because otherwise it would violate the Dummy axiom. This conditional sampling version of SHAP exists and was suggested by Aas, Jullum, and Løland (2021).

⚠️ Conditional Shapley values may give non-zero Shapley values to unused features

The problem with the conditional expectation is that features that have no influence on the prediction function  $\hat{f}$  can get a TreeSHAP estimate different from zero, as shown by Sundararajan and Najmi (2020) and Janzing, Minorics, and Blöbaum (2020). The non-zero estimate can happen when the feature is correlated with another feature that actually has an influence on the prediction.

For images, Figure 18.2 describes a possible mapping function. Assigning the average color of surrounding pixels or similar would also be an option.

The big difference between SHAP and LIME is the weighting of the instances in the regression model. LIME weights the instances according to how close they are to the original instance. The more 0's in the coalition vector, the smaller the weight in LIME. SHAP weights the sampled instances according to the weight the coalition would get in the Shapley value estimation. Small coalitions (few 1's) and large coalitions (i.e. many 1's) get the largest weights. The intuition behind it is: We learn most about individual features if we can study their effects in isolation. If a coalition consists of a single feature, we can learn about this feature's isolated main effect on the prediction. If a coalition consists of all but one feature, we can learn about this feature's total effect (main effect plus feature interactions). If a coalition consists of half the features, we learn little about an individual feature's contribution, as there are many pos-

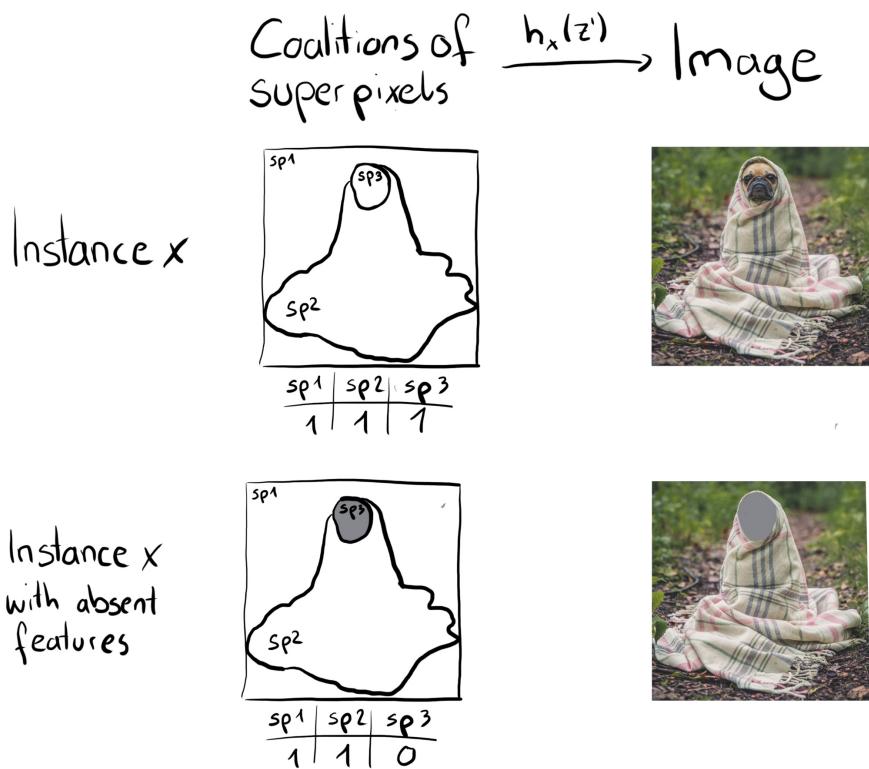


Figure 18.2: Function  $h_x$  maps coalitions of superpixels (sp) to images. Superpixels are groups of pixels. For present features (1),  $h_x$  returns the corresponding part of the original image. For absent features (0),  $h_x$  greys out the corresponding area.

sible coalitions with half of the features. To achieve Shapley compliant weighting, Lundberg and Lee (2017) proposed the SHAP kernel:

$$\pi_{\mathbf{x}}(\mathbf{z}') = \frac{(M-1)}{\binom{M}{|\mathbf{z}'|} |\mathbf{z}'| (M - |\mathbf{z}'|)}$$

Here,  $M$  is the maximum coalition size and  $|\mathbf{z}'|$  the number of present features in instance  $\mathbf{z}'$ . Lundberg and Lee (2017) show that linear regression with this kernel weight yields Shapley values. If you were to use the SHAP kernel with LIME on the coalition data, LIME would also estimate Shapley values!

We can be a bit smarter about the sampling of coalitions: The smallest and largest coalitions take up most of the weight. We get better Shapley value estimates by using some of the sampling budget  $K$  to include these high-weight coalitions instead of sampling blindly. We start with all possible coalitions with 1 and  $M-1$  features, which makes  $2M$  coalitions in total. When we have enough budget left (current budget is  $K-2M$ ), we can include coalitions with 2 features and with  $M-2$  features and so on. From the remaining coalition sizes, we sample with readjusted weights.

We have the data, the target and the weights; Everything we need to build our weighted linear regression model:

$$g(\mathbf{z}') = \phi_0 + \sum_{j=1}^M \phi_j z'_j$$

We train the linear model  $g$  by optimizing the following loss function  $L$ :

$$L(\hat{f}, g, \pi_{\mathbf{x}}) = \sum_{\mathbf{z}' \in \mathbf{Z}} [\hat{f}(h_{\mathbf{x}}(\mathbf{z}')) - g(\mathbf{z}')]^2 \pi_{\mathbf{x}}(\mathbf{z}')$$

where  $\mathbf{Z}$  is the training data. This is the good old boring sum of squared errors that we usually optimize for linear models. The estimated coefficients of the model, the  $\phi_j$ 's, are the Shapley values.

Since we are in a linear regression setting, we can also make use of the standard tools for regression. For example, we can add regularization terms to make the model sparse. If we add an L1 penalty to the loss  $L$ , we can create sparse explanations. (I'm not so sure whether the resulting coefficients would still be valid Shapley values though.)

## 18.2.2 TreeSHAP

Lundberg, Erion, and Lee (2019) proposed TreeSHAP, a variant of SHAP for tree-based machine learning models such as decision trees, random forests, and gradient-boosted trees. TreeSHAP was introduced as a fast, model-specific alternative to KernelSHAP. How much faster is TreeSHAP? Compared to exact KernelSHAP, it reduces the computational complexity from  $O(TL2^M)$  to  $O(TLD^2)$ , where T is the number of trees, L is the maximum number of leaves in any tree, and D is the maximal depth of any tree.

TreeSHAP comes in two versions:

- “Interventional” computes the classic Shapley values.
- “Tree-path dependent” computes something akin to conditional SHAP values.

The original implementation in the `shap` Python package used to be the tree-path dependent version, but is now the interventional one.

The estimation for both interventional and tree-path dependent estimation is rather complex, so I’ll just give you the rough idea (read: I haven’t fully understood the full version myself). Essentially, TreeSHAP leverages the tree structure to compute the Shapley values more efficiently.

Interventional TreeSHAP calculates the usual SHAP values. The following description is for estimating Shapley values for a single tree, for a data point  $\mathbf{x}$  to explain, and, for simplicity, the background dataset contains just one data point  $\mathbf{z}$ :

The Shapley value, as we know, is computed by repeatedly forming coalitions of feature values, which includes present players (feature values from  $\mathbf{x}$ ) and absent players (feature values from  $\mathbf{z}$ ). However, for many of these coalitions, adding a feature from  $\mathbf{x}$  wouldn’t change the prediction, since a decision tree only has a limited amount of distinct predictions due to the decision structure. For example, a binary tree of depth 5 has a maximum of 32 possible predictions. So instead of iterating through all coalitions, the Interventional TreeSHAP estimator explores the tree paths to only work with the coalitions that would actually change the predictions. The tricky part is to correctly weight and combine these marginal contributions, which makes the interventional TreeSHAP algorithm more difficult to understand. And of course, damn recursion. To get the Shapley values for an ensemble of trees, for example for a random forest, simply combine the Shapley values the same way the ensemble predictions are combined. In the case of a random forest, it would be averaging the values.

Path-dependent TreeSHAP works in a similar fashion, making use of the tree structure. The basic idea is to push all subsets of coalition S down the tree simultaneously. And in each split, we keep track of the number of instances for the subsets.

### 18.2.3 Permutation Method

The most efficient model-agnostic estimator is the Permutation Method. The idea is to sample cleverly from the coalitions by creating permutations of the features.

Let's look at an example (taken from the book [Interpreting Machine Learning Models with SHAP](#)) with four feature values:  $x_{\text{park}}$ ,  $x_{\text{cat}}$ ,  $x_{\text{area}}$ , and  $x_{\text{floor}}$ . For simplicity, I'll shorten  $x_{\text{park}}^{(i)}$  as  $x_{\text{park}}$ .

A random permutation of these features would be:

$$(x_{\text{cat}}, x_{\text{area}}, x_{\text{park}}, x_{\text{floor}})$$

Based on this permutation, we can compute marginal contributions by starting to build coalitions from left to right:

- Add  $x_{\text{cat}}$  to  $\emptyset$
- Add  $x_{\text{area}}$  to  $\{x_{\text{cat}}\}$
- Add  $x_{\text{park}}$  to  $\{x_{\text{cat}}, x_{\text{area}}\}$
- Add  $x_{\text{floor}}$  to  $\{x_{\text{cat}}, x_{\text{area}}, x_{\text{park}}\}$

And the same we can do backwards:

- Add  $x_{\text{floor}}$  to  $\emptyset$
- Add  $x_{\text{park}}$  to  $\{x_{\text{floor}}\}$
- Add  $x_{\text{area}}$  to  $\{x_{\text{park}}, x_{\text{floor}}\}$
- Add  $x_{\text{cat}}$  to  $\{x_{\text{area}}, x_{\text{park}}, x_{\text{floor}}\}$

The permutation changes one feature at a time. This reduces the number of model calls since the second term of a marginal contribution (a model prediction) is also needed to compute the next marginal contribution. For example, the coalition  $\{x_{\text{cat}}, x_{\text{area}}\}$  is used to calculate both the marginal contribution of  $x_{\text{park}}$  to  $\{x_{\text{cat}}, x_{\text{area}}\}$  and of  $x_{\text{area}}$  to  $\{x_{\text{cat}}\}$ . Each forward and backward permutation gives us two marginal contributions per feature. By doing a couple of permutations like this, we get a pretty good estimate of the Shapley values. To actually compute the Shapley values, we have to define the Shapley values in terms of permutations instead of coalitions: There are  $p!$  possible permutations of the features and  $o(k)$  the k-th permutation, then the Shapley value for feature  $j$  can be computed as:

$$\hat{\phi}_j^{(i)} = \frac{1}{m} \sum_{k=1}^m \hat{\Delta}_{o(k),j}$$

The term  $\hat{\Delta}_{o(k),j}$  is the  $k$ -th marginal contribution. This means we can compute the Shapley value as a simple average over all contributions. However, the motivation for permutation

estimation was to avoid computing all possible coalitions or permutations. Good thing is we can sample permutations and still take the average. Since we perform forward and backward iterations, we compute the Shapley value as:

$$\hat{\phi}_j^{(i)} = \frac{1}{2m} \sum_{k=1}^m (\hat{\Delta}_{o(k),j} + \hat{\Delta}_{-o(k),j})$$

Permutation  $-o(k)$  is the reverse version of the permutation. The permutation procedure with forward and backward iterations, also known as antithetic sampling, performs quite well compared to other SHAP value sampling estimators (Mitchell et al. 2022). The permutation method additionally ensures that the efficiency axiom is always satisfied, meaning when you add up the SHAP values, they equal prediction minus average prediction. For a rough idea of how many permutations you might need: the `shap` package defaults to 10.

### 18.3 Example

I trained a random forest classifier with 100 trees to predict the [penguin sex](#). We'll use SHAP to explain individual predictions. We can use the fast Interventional TreeSHAP estimation method instead of the slower KernelSHAP method, since a random forest is an ensemble of trees. But instead of relying on the conditional distribution, this example uses the marginal distribution. This is described in the package, but not in the original paper. The Python TreeSHAP function is slower with the marginal distribution, but still faster than KernelSHAP, since it scales linearly with the rows in the data.

Because we use the marginal distribution here, the interpretation is the same as in the [Shapley value chapter](#). But with the Python `shap` package comes a different visualization: You can visualize feature attributions such as Shapley values as “forces.” Each feature value is a force that either increases or decreases the prediction. The prediction starts from the baseline. The baseline for Shapley values is the average of all predictions. In the plot, each Shapley value is an arrow that pushes to increase (positive value) or decrease (negative value) the prediction. These forces balance each other out at the actual prediction of the data instance.

Figure 18.3 shows SHAP explanation force plots for two penguins from the Palmer penguins dataset: The first penguin has a high probability of being an Adelie penguin due to its small bill length. The second penguin is unlikely to be Adelie due to its large bill length and flipper length.

### 18.4 SHAP aggregation plots

The section before showed explanations for individual predictions.

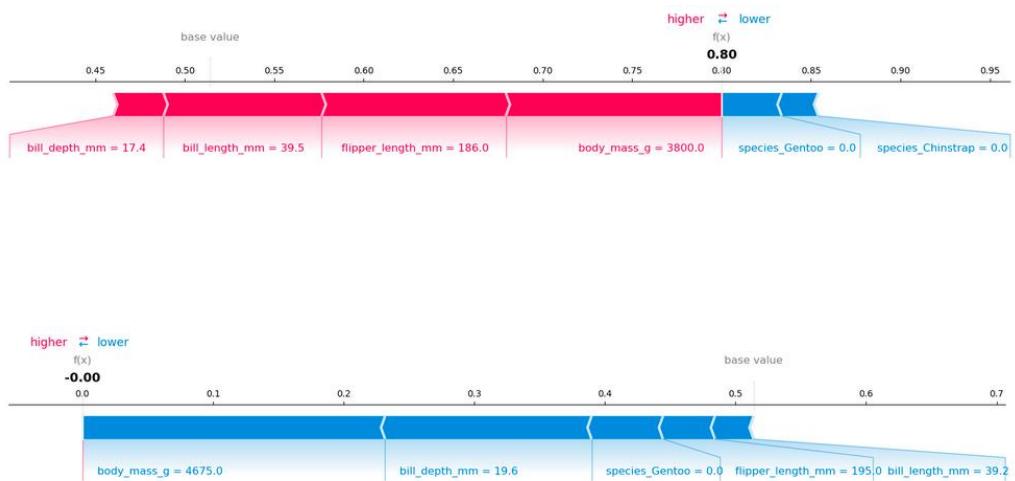


Figure 18.3: SHAP values for two penguins. The baseline – the average predicted probability – is 0.47. Each feature can be seen as a force that pushes the prediction either up or down from the baseline.

Shapley values can be combined into global explanations. If we run SHAP for every instance, we get a matrix of Shapley values. This matrix has one row per data instance and one column per feature. We can interpret the entire model by analyzing the Shapley values in this matrix.

We start with SHAP feature importance.

### 18.4.1 SHAP Feature Importance

The idea behind SHAP feature importance is simple: Features with large absolute Shapley values are important. Since we want the global importance, we average the **absolute** Shapley values per feature across the data:

$$I_j = \frac{1}{n} \sum_{i=1}^n |\phi_j^{(i)}|$$

Next, we sort the features by decreasing importance and plot them. Figure 18.4 shows the SHAP feature importance for the random forest trained before for classifying penguins. The body mass was the most important feature, changing the predicted probability for Adelie 25 percentage points (0.25 on the x-axis).

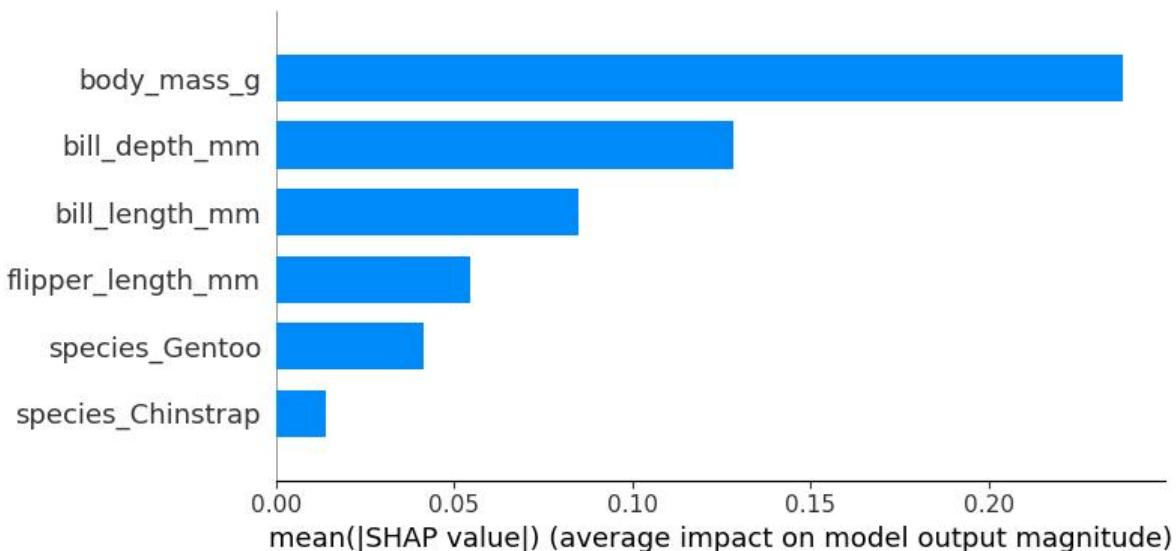


Figure 18.4: SHAP feature importance measured as the mean absolute Shapley values.

SHAP feature importance is an alternative to [permutation feature importance](#). There's a big difference between both importance measures: Permutation feature importance is based on the decrease in model performance. SHAP is based on the magnitude of feature attributions.

The feature importance plot is useful, but contains no information beyond the importances. For a more informative plot, we will next look at the summary plot.

### 18.4.2 SHAP Summary Plot

The summary plot combines feature importance with feature effects. Each point on the summary plot is a Shapley value for a feature and an instance. The position on the y-axis is determined by the feature and on the x-axis by the Shapley value. The color represents the value of the feature from low to high. Overlapping points are jittered in the y-axis direction, so we get a sense of the distribution of the Shapley values per feature. The features are ordered according to their importance. In the summary plot, Figure 18.5, we see first indications of the relationship between the value of a feature and the impact on the prediction. A higher body mass contributes negatively to  $P(\text{female})$ . The summary plot also shows that body mass has the widest range of effects for the different penguins.

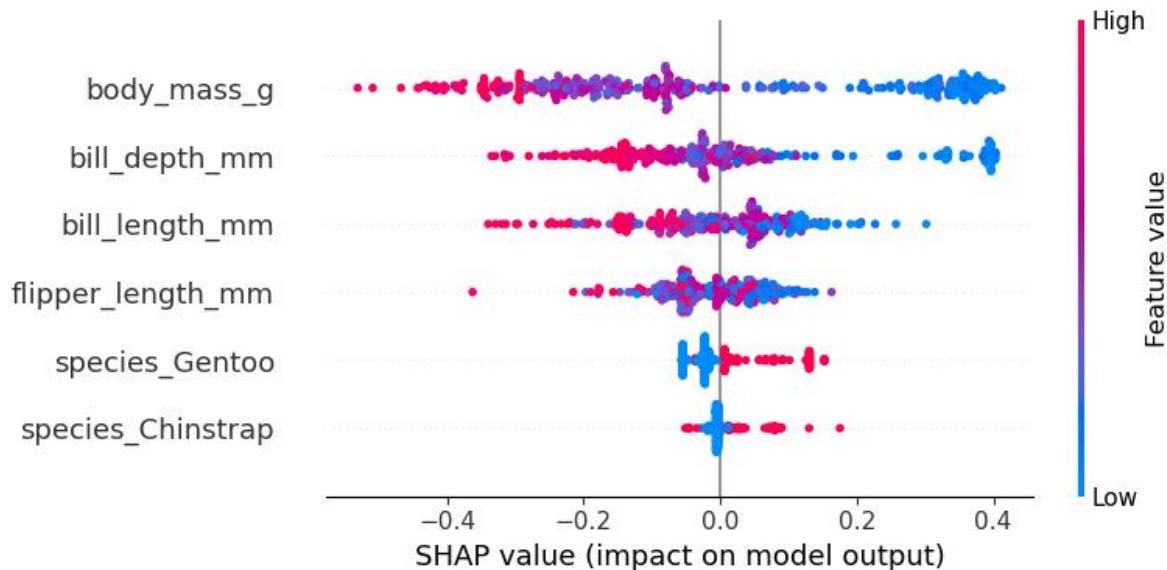


Figure 18.5: SHAP summary plot.

But to see the exact form of the relationship, we have to look at SHAP dependence plots.

### 18.4.3 SHAP Dependence Plot

SHAP feature dependence might be the simplest global interpretation plot: 1) Pick a feature. 2) For each data instance, plot a point with the feature value on the x-axis and the corresponding Shapley value on the y-axis. 3) Done.

Mathematically, the plot contains the following points:  $\{(x_j^{(i)}, \phi_j^{(i)})\}_{i=1}^n$

Figure 18.6 shows the SHAP feature dependence for the body mass. The heavier the penguin, the less likely it's female.

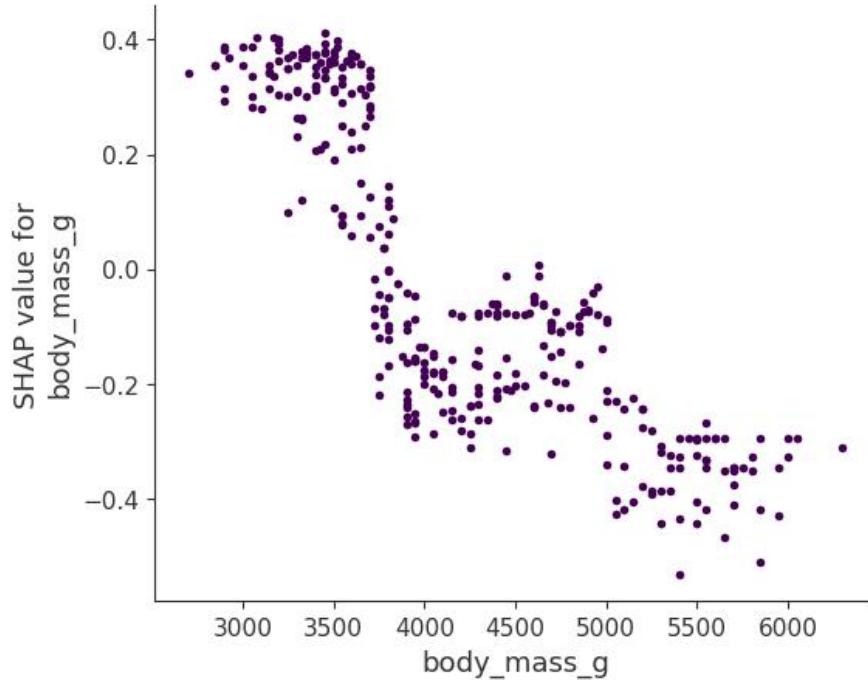


Figure 18.6: SHAP dependence plot for body mass. The x-axis shows the feature, the y-axis the SHAP values. Each dot represents a data point.

SHAP dependence plots are an alternative to global feature effect methods like the [partial dependence plots](#) and [accumulated local effects](#). While PDP and ALE plots show average effects, SHAP dependence also shows the variance on the y-axis. Especially in the case of interactions, the SHAP dependence plot will be much more dispersed on the y-axis. The dependence plot can be improved by highlighting these feature interactions.

#### 18.4.4 SHAP Interaction Values

The interaction effect is the additional combined feature effect after accounting for the individual feature effects. The Shapley interaction index from game theory is defined as:

$$\phi_{i,j} = \sum_{S \subseteq \{i,j\}} \frac{|S|!(M-|S|-2)!}{2(M-1)!} \delta_{ij}(S)$$

when  $i \neq j$  and  $\delta_{ij}(S) = \hat{f}_{\mathbf{x}}(S \cup \{i, j\}) - \hat{f}_{\mathbf{x}}(S \cup \{i\}) - \hat{f}_{\mathbf{x}}(S \cup \{j\}) + \hat{f}_{\mathbf{x}}(S)$ .

This formula subtracts the main effect of the features so that we get the pure interaction effect after accounting for the individual effects. We average the values over all possible feature coalitions  $S$ , as in the Shapley value computation. When we compute SHAP interaction values for all features, we get one matrix per instance with dimensions  $M \times M$ , where  $M$  is the number of features. How can we use the interaction index? For example, to automatically color the SHAP feature dependence plot with the strongest interaction, as in Figure 18.7. Here, body mass interacts with bill depth.

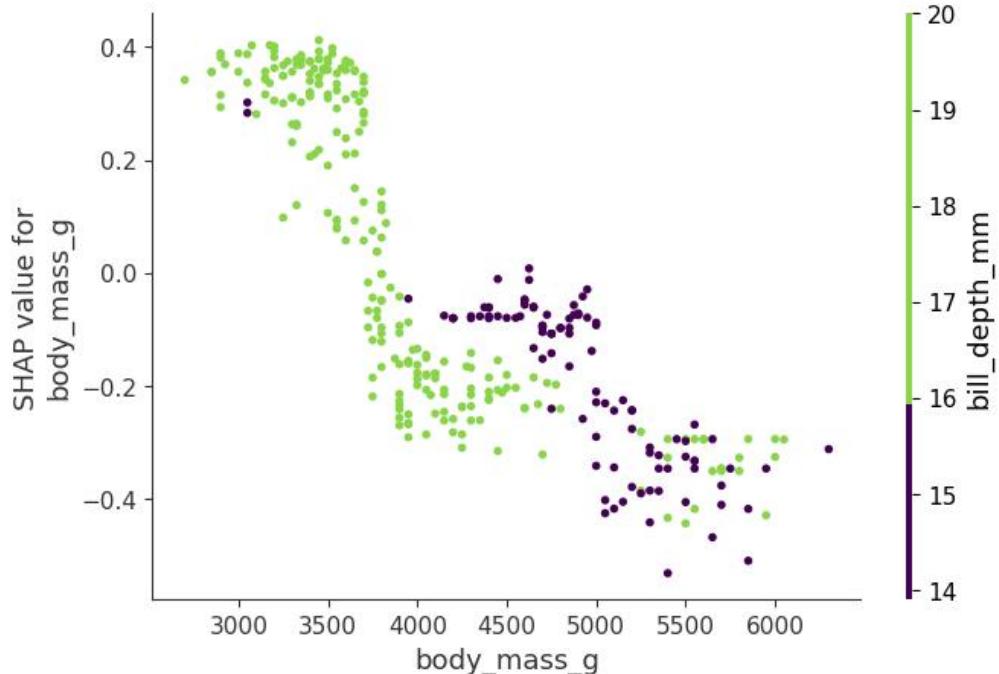


Figure 18.7: SHAP feature dependence plot with interaction visualization. Here we see the dependence plot of bill length in interaction with the body mass. Especially for longer bills, the contribution of the bill length to  $P(\text{Adelie})$  differs based on body mass.

#### Analyze interactions in depth

The topic of SHAP and interactions goes a lot deeper. If you require more sophisticated analysis of SHAP interactions, I recommend looking into the [shapiq package](#) (Muschalik et al. 2024).

#### 18.4.5 Clustering Shapley Values

You can cluster your data with the help of Shapley values. The goal of clustering is to find groups of similar instances. Normally, clustering is based on features. Features are often on different scales. For example, height might be measured in meters, color intensity from 0 to 100, and some sensor output between -1 and 1. The difficulty is to compute distances between instances with such different, non-comparable features.

SHAP clustering works by clustering the Shapley values of each instance. This means that you cluster instances by explanation similarity. All SHAP values have the same unit – the unit of the prediction space. You can use any clustering method.

Figure 18.8 uses hierarchical agglomerative clustering to order the instances. The plot consists of many force plots, each of which explains the prediction of an instance. We rotate the force plots vertically and place them side by side according to their clustering similarity.

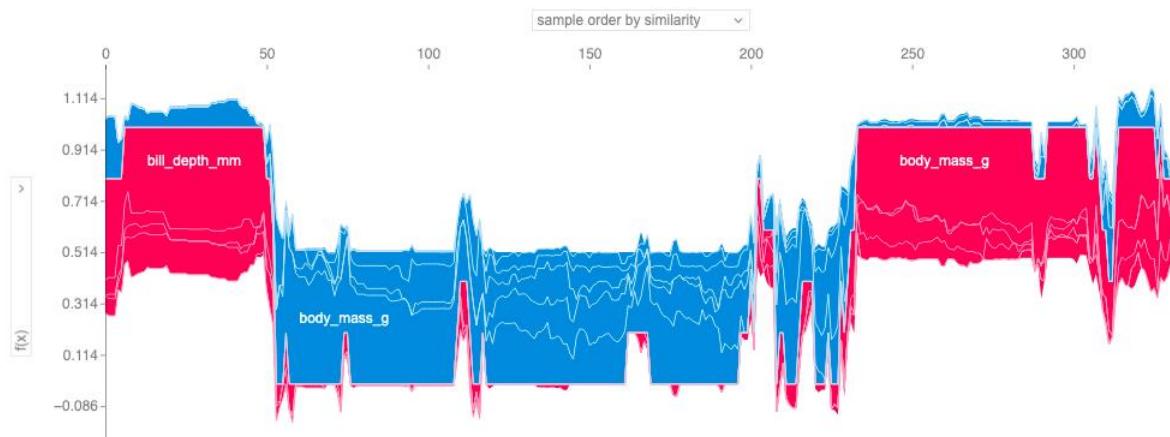


Figure 18.8: Stacked SHAP explanations clustered by explanation similarity. Each position on the x-axis is an instance of the data. Red SHAP values increase the prediction, blue values decrease it.

### 18.5 Strengths

Since SHAP computes Shapley values, all the advantages of Shapley values apply: SHAP has a **solid theoretical foundation** in game theory. The prediction is **fairly distributed** among the feature values. We get **contrastive explanations** that compare the prediction with the average prediction.

**SHAP connects LIME and Shapley values.** This is very useful to better understand both methods. It also helps to unify the field of interpretable machine learning.

SHAP has a **fast implementation for tree-based models**. I believe this was key to the popularity of SHAP because the biggest barrier for adoption of Shapley values is the slow computation.

The fast computation makes it possible to compute the many Shapley values needed for the **global model interpretations**. The global interpretation methods include feature importance, feature dependence, interactions, clustering, and summary plots. With SHAP, global interpretations are consistent with the local explanations, since the Shapley values are the “atomic unit” of the global interpretations. If you use LIME for local explanations and partial dependence plots plus permutation feature importance for global explanations, you lack a common foundation.

## 18.6 Limitations

**KernelSHAP is slow.** This makes KernelSHAP impractical to use when you want to compute Shapley values for many instances. Also, all global SHAP methods, such as SHAP feature importance, require computing Shapley values for a lot of instances.

**KernelSHAP ignores feature dependence.** Most other permutation-based interpretation methods have this problem. By replacing feature values with values from random instances, it is usually easier to randomly sample from the marginal distribution. However, if features are dependent, e.g., correlated, this leads to putting too much weight on unlikely data points.

**Path-dependent TreeSHAP can produce unintuitive feature attributions.** While TreeSHAP solves the problem of extrapolating to unlikely data points, it does so by changing the value function and therefore slightly changes the game. TreeSHAP changes the value function by relying on the conditional expected prediction. With the change in the value function, features that have no influence on the prediction can get a TreeSHAP value different from zero.

The disadvantages of Shapley values also apply to SHAP: Shapley values **can be misinterpreted**.

It’s **possible to create intentionally misleading interpretations** with SHAP, which can hide biases (Slack et al. 2020). If you are the data scientist creating the explanations, this is not an actual problem (it would even be an advantage if you are the evil data scientist who wants to create misleading explanations). For the receivers of a SHAP explanation, it’s a disadvantage: they cannot be sure about the truthfulness of the explanation.

## 18.7 Software

Lundberg implemented SHAP in the [shap](#) Python package, which is now maintained by a much bigger team.

This implementation works for models trained with the [scikit-learn](#) machine learning library for Python. The shap package was also used for the examples in this chapter. SHAP is integrated into the tree boosting frameworks [xgboost](#) and [LightGBM](#), and you can find it in [PiML](#), a more general interpretability library. In R, there are the [shapper](#) and [fastshap](#) packages. SHAP is also included in the R [xgboost](#) package. Specifically for SHAP interactions, there is the Python [shapiq](#) package.

## **Part III**

# **Global Model-Agnostic Methods**

# 19 Partial Dependence Plot (PDP)

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

## 19.1 Definition and estimation

The partial dependence function for regression is defined as:

$$\hat{f}_S(\mathbf{x}_S) = \mathbb{E}_{\mathbf{X}_C} [\hat{f}(\mathbf{x}_S, X_C)] = \int \hat{f}(\mathbf{x}_S, X_C) d\mathbb{P}(\mathbf{X}_C)$$

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

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

$$\hat{f}_S(\mathbf{x}_S) = \frac{1}{n} \sum_{i=1}^n \hat{f}(\mathbf{x}_S, \mathbf{x}_C^{(i)})$$

This is equivalent to averaging all the [ICE curves](#) of a dataset. The partial function tells us for given value(s) of features in set S what the average marginal effect on the prediction is. In this formula,  $\mathbf{x}_C^{(i)}$  are actual feature values from the dataset for the features in which we are not interested, and  $n$  is the number of instances in the dataset. The PDP treats the features

in  $C$  regardless of their correlation with features in  $S$ . In the case of correlation, the averages calculated for the partial dependence plot will include data points that are very unlikely or even impossible (see disadvantages).

**⚠ Correlated features are a problem**

Be cautious when interpreting PDPs for (strongly) correlated features: In this case, the partial dependence plot includes unrealistic data instances.

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

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

### Categorical features

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

## 19.2 Examples

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

Let’s return to the regression example, in which we predict the number of [bikes that will be rented on a given day](#). First, we fit a machine learning model, then we analyze the partial dependencies. In this case, we have fitted a random forest to predict the number of bikes and used the partial dependence plot to visualize the relationships the model has learned, see Figure 19.1. The influence of the weather features on the predicted bike counts is visualized in the following figure. The largest differences can be seen in the temperature. The hotter, the more bikes are rented. This trend goes up to 20 degrees Celsius, then flattens, and drops slightly around 30.

Potential bikers are increasingly inhibited in renting a bike when humidity exceeds 60%. In addition, the more wind, the fewer people like to cycle, which makes sense. Interestingly, the predicted number of bike rentals does not fall when wind speed increases from 25 to 35 km/h, but there is not much training data, so the machine learning model could probably not learn a

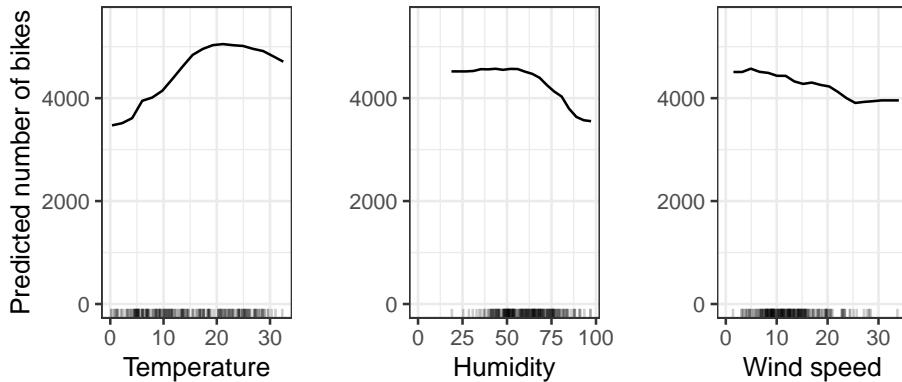


Figure 19.1: PDPs for the bike count prediction model and temperature, humidity and wind speed. Marks on the x-axis indicate the data distribution.

meaningful prediction for this range. At least intuitively, I would expect the number of bikes to decrease with increasing wind speed, especially when the wind speed is very high.

To illustrate a partial dependence plot with a categorical feature, we examine the effect of the season feature on the predicted bike rentals, see Figure 19.2. All seasons show a similar effect on the model predictions; only for winter, the model predicts fewer bike rentals, and slightly fewer for spring.

We also compute the partial dependence for [penguin classification](#) (male/female). To mix things up a little bit, we analyze both the random forest and the logistic regression approach. Both predict  $\mathbb{P}(Y^{(i)} = \text{female})$  based on body measurements. We compute and visualize the partial dependence of the probability for being female based on body mass and bill depth for the random forest (Figure 19.3). The heavier the penguin, the less likely it is female. We see a similar pattern for bill depth. The random forest PDP is way more rugged due to the decision-tree basis.

Figure 19.3 has a big problem: It throws all penguin species together. But we can easily compute the PDP by species. For this, we only have to compute the individual PDPs by subsetting the data and plotting the curves in the same plot, as I did in Figure 19.4. A more nuanced interpretation emerges. For Adelie, more weight means less likely to be female. For Chinstrap, the weight doesn't affect the probability very much. Gentoo penguins are, in general, heavier but also show the “males are more heavy” pattern. Also, logistic regression is, surprise, surprise, smoother, and the random forest probabilities are drawn to the mean.

We can also visualize the partial dependence of two features at once (Figure 19.5): here, bill depth and bill length. The analyzed model is the random forest. There's an interaction between the two, because at small bill depths, bill length doesn't make a difference, but for longer bills,  $P(\text{female})$  becomes smaller.

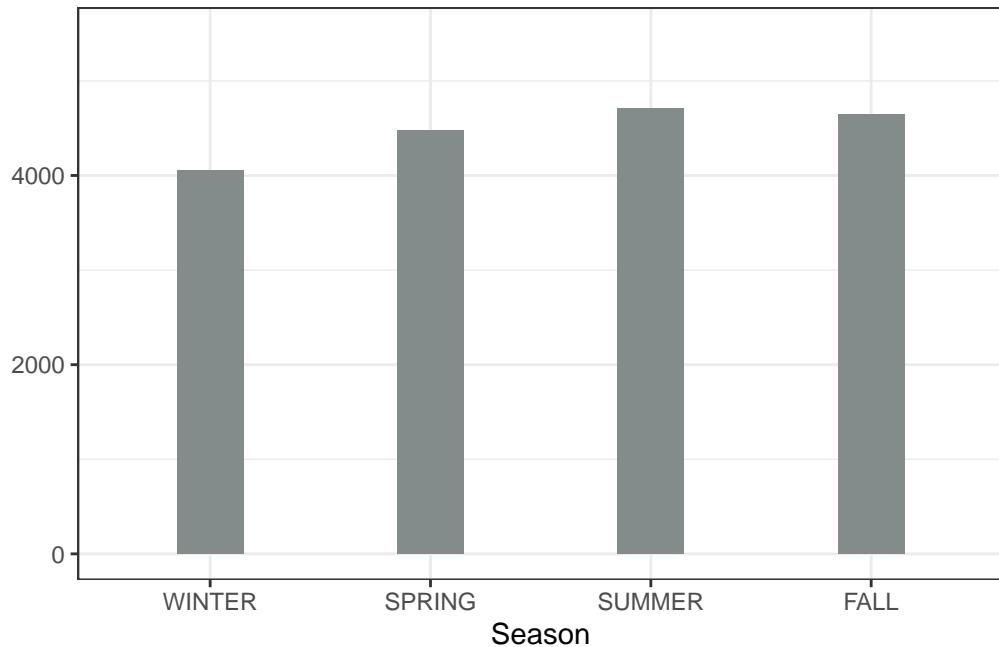


Figure 19.2: PDPs for the bike count prediction model and the season.

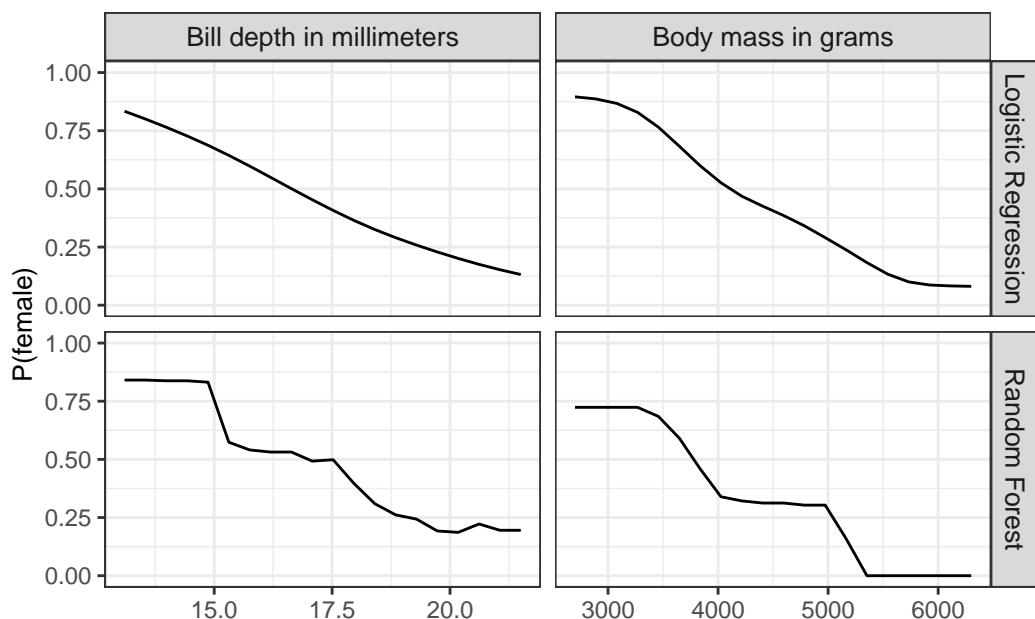


Figure 19.3: PDPs for bill depth and body mass for both the logistic regression model and the random forest.

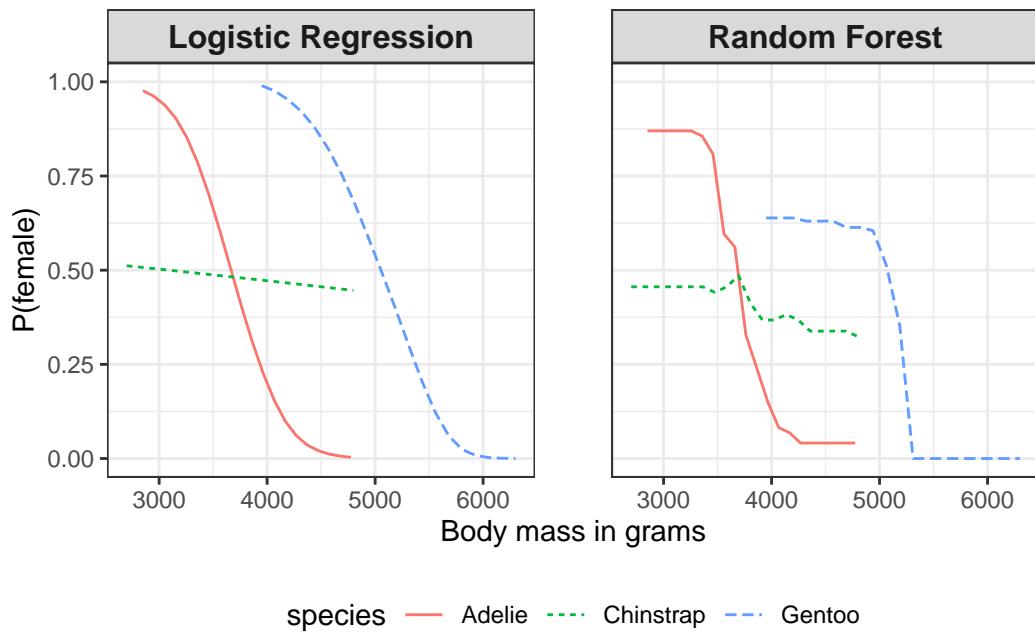


Figure 19.4: PDP split by penguin species, comparing logistic regression and random forest.

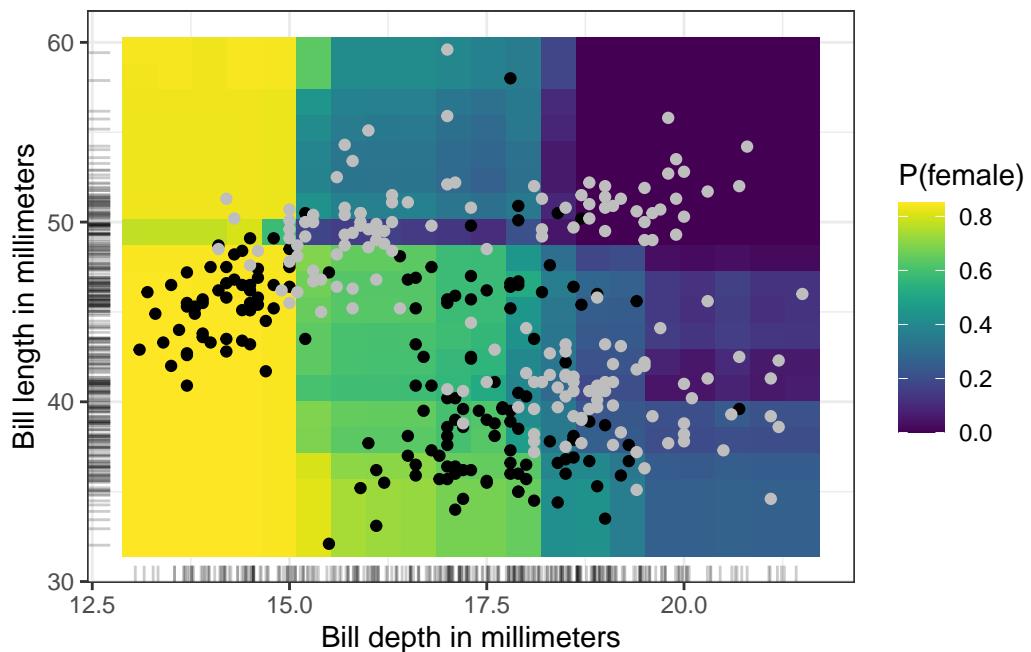


Figure 19.5: PDP for  $P(\text{female})$  and the interaction between bill depth and bill length. The dots are black for female penguins, gray for male.

### Reduce interactions, improve PDP interpretability

Even though PDP is a model-agnostic method, having a model with fewer interactions and more homogeneous effects makes the PDP interpretation simpler and reduces (some) risks of misinterpretation. Means to do this are reducing tree depth when using tree-based methods or adding monotonicity constraints. Alternatively, you can optimize for lower interactions, sparsity, and less complex effects in a model-agnostic way, see Molnar, Casalicchio, and Bischl (2020b).

## 19.3 PDP-based feature importance

Greenwell, Boehmke, and McCarthy (2018) proposed a simple partial dependence-based feature importance measure. The basic motivation is that a flat PDP indicates that the feature is not important, and the more the PDP varies, the more important the feature is. For numerical features, importance is defined as the deviation of each unique feature value from the average curve:

$$I(\mathbf{x}_S) = \sqrt{\frac{1}{K-1} \sum_{k=1}^K (\hat{f}_S(\mathbf{x}_S^{(k)}) - \frac{1}{K} \sum_{k=1}^K \hat{f}_S(\mathbf{x}_S^{(k)}))^2}$$

Note that here the  $\mathbf{x}_S^{(k)}$  are the  $K$  unique values of feature  $X_S$ . For categorical features, we have:

$$I(X_S) = \frac{\max_k(\hat{f}_S(\mathbf{x}_S^{(k)})) - \min_k(\hat{f}_S(\mathbf{x}_S^{(k)}))}{4}$$

This is the range of the PDP values for the unique categories divided by four. This strange way of calculating the deviation is called the range rule. It helps to get a rough estimate for the deviation when you only know the range. And the denominator four comes from the standard normal distribution: In the normal distribution, 95% of the data are minus two and plus two standard deviations around the mean. So the range divided by four gives a rough estimate that probably underestimates the actual variance.

This PDP-based feature importance should be interpreted with care. It captures only the main effect of the feature and ignores possible feature interactions. A feature could be very important based on other methods such as [permutation feature importance](#), but the PDP could be flat as the feature affects the prediction mainly through interactions with other features. Another drawback of this measure is that it is defined over the unique values. A unique feature value with just one instance is given the same weight in the importance computation as a value with many instances.

## 19.4 Strengths

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

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

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

The calculation for the partial dependence plots has a **causal interpretation**. We intervene on a feature and measure the changes in the predictions. In doing so, we analyze the causal relationship between the feature and the prediction (Zhao and Hastie 2019). The relationship is causal for the model – because we explicitly model the outcome as a function of the features – but not necessarily for the real world!

## 19.5 Limitations

The realistic **maximum number of features** in a partial dependence function that can be meaningfully visualized is two. This is not the fault of PDPs, but of the 2-dimensional representation (paper or screen) and also of our inability to imagine more than 3 dimensions. However, it's still possible to compute higher-order PDPs.

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

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

problem is [Accumulated Local Effect plots](#) or short ALE plots that work with the conditional instead of the marginal distribution.

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

## 19.6 Software and alternatives

There are a number of R packages that implement PDPs. I used the `iml` package for the examples, but there is also `pdp` and `DALEX`. In Python, partial dependence plots are built into `scikit-learn`, and you can use `PDPBox`. Or you can use the [Python Interpretable Machine Learning \(PiML\) library](#).

Alternatives to PDPs presented in this book are [ALE plots](#) and [ICE curves](#).

# 20 Accumulated Local Effects (ALE)

Accumulated local effects (Apley and Zhu 2020) describe how features influence the prediction of a machine learning model on average. ALE plots are a faster and unbiased alternative to partial dependence plots (PDPs).

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

## 20.1 Motivation and intuition

If features of a machine learning model are correlated, the partial dependence plot cannot be trusted. The computation of a partial dependence plot for a feature that is strongly correlated with other features involves averaging predictions of artificial data instances that are unlikely in reality. This can greatly bias the estimated feature effect. Imagine calculating partial dependence plots for a machine learning model that predicts the value of a house depending on the number of rooms and the size of the living area. We're interested in the effect of the living area on the predicted value. As a reminder, the recipe for partial dependence plots is: 1) Select feature. 2) Define grid. 3) Per grid value: a) Replace feature with grid value and b) average predictions. 4) Draw curve. For the calculation of the first grid value of the PDP – say  $30 \text{ m}^2$  – we replace the living area for **all** instances by  $30 \text{ m}^2$ , even for houses with 10 rooms. Sounds to me like a very unusual house. The partial dependence plot includes these unrealistic houses in the feature effect estimation and pretends that everything is fine. Figure 20.1 illustrates two correlated features and how it comes that the partial dependence plot method averages predictions of unlikely instances.

What can we do to get a feature effect estimate that respects the correlation of the features? We could average over the conditional distribution of the feature, meaning at a grid value of  $X_1$ , we average the predictions of instances with a similar value for  $X_1$ . The solution for calculating feature effects using the conditional distribution is called Marginal Plots, or M-Plots (confusing name, since they are based on the conditional, not the marginal distribution). Wait, did I not promise you to talk about ALE plots? M-Plots are not the solution we are looking for. Why do M-Plots not solve our problem? If we average the predictions of all houses of about  $30 \text{ m}^2$ , we estimate the **combined** effect of living area and of number of rooms, because

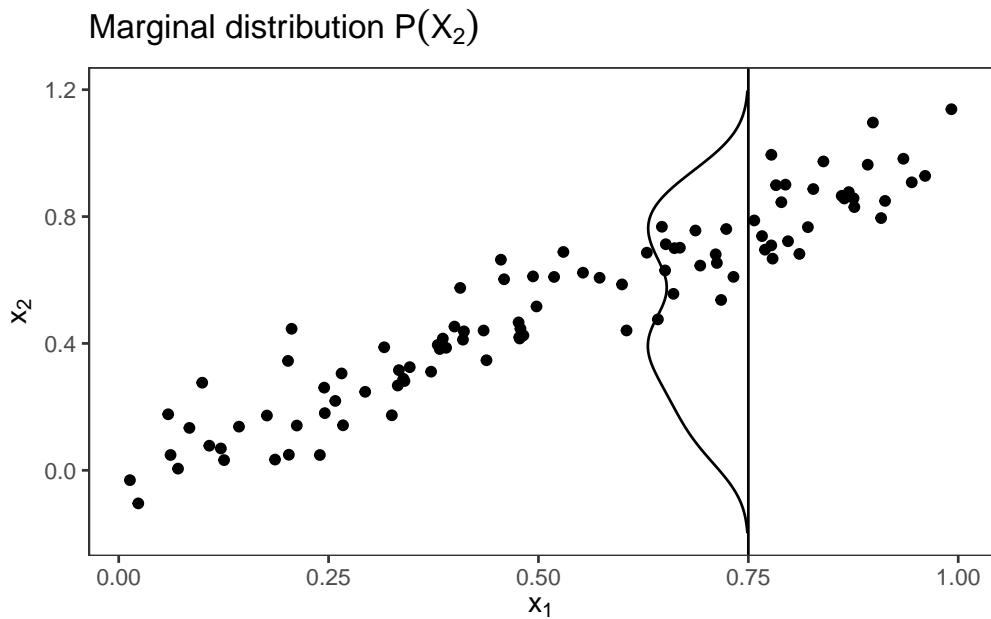


Figure 20.1: Correlated features  $X_1$  and  $X_2$ . When calculating the PDP at  $x_1 = 0.75$  values for  $x_2$  from the entire range of  $X_2$  are sampled (vertical line). This results in the PDP using unlikely combinations of  $X_1$  and  $X_2$  (e.g.,  $x_2 = 0.2$  at  $x_1 = 0.75$ ).

of their correlation. Suppose that the living area has no effect on the predicted value of a house, only the number of rooms has. The M-Plot would still show that the size of the living area increases the predicted value, since the number of rooms increases with the living area. Figure 20.2 shows for two correlated features how M-Plots work.

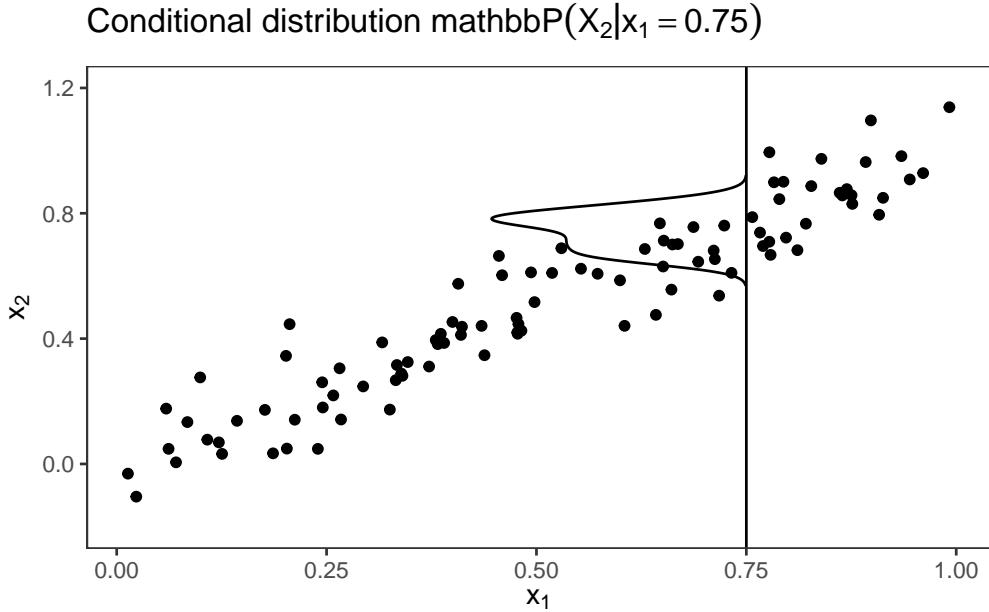


Figure 20.2: Correlated features  $X_1$  and  $X_2$ . M-Plots average over the conditional distribution. Here the conditional distribution of  $X_2$  at  $x_1 = 0.75$ . Averaging the local predictions leads to mixing the effects of both features.

M-Plots avoid averaging predictions of unlikely data instances, but they mix the effect of a feature with the effects of all correlated features. ALE plots solve this problem by calculating – also based on the conditional distribution of the features – **differences in predictions instead of averages**. For the effect of living area at  $30 \text{ m}^2$ , the ALE method uses all houses with about  $30 \text{ m}^2$ , gets the model predictions pretending these houses were  $31 \text{ m}^2$ , minus the prediction pretending they were  $29 \text{ m}^2$ . This gives us the pure effect of the living area and does not mix the effect with the effects of correlated features. The use of differences blocks the effect of other features. Figure 20.3 provides intuition on how ALE plots are calculated.

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

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

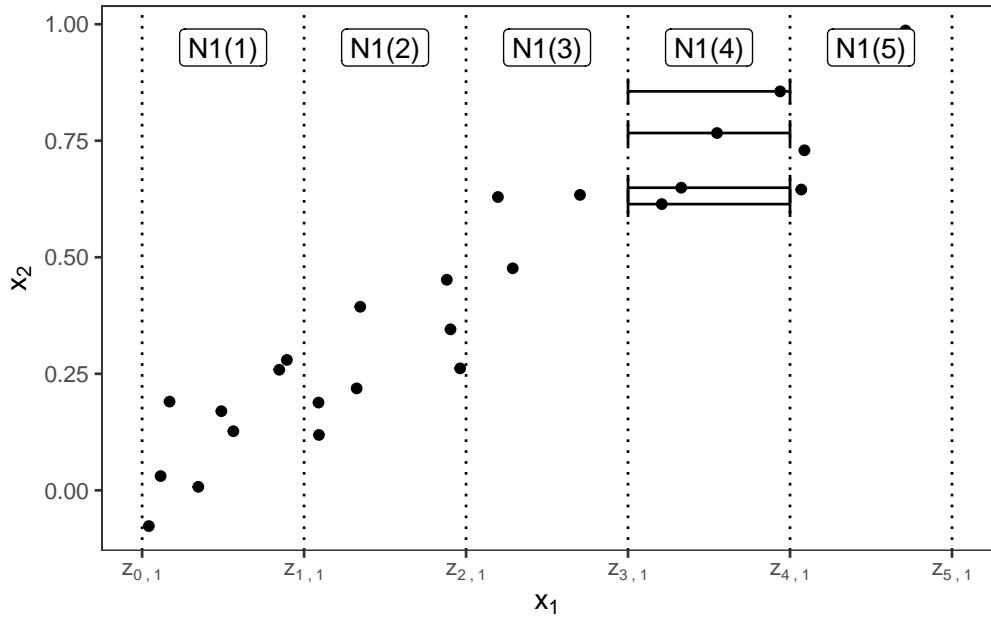


Figure 20.3: Calculation of ALE for feature  $X_1$ , which is correlated with  $X_2$ . First, we divide the feature into intervals (vertical lines). For the data instances (points) in an interval, we calculate the difference in the prediction when we replace the feature with the upper and lower limit of the interval (horizontal lines). These differences are later accumulated and centered, resulting in the ALE curve.

- **M-Plots:** “Let me show you what the model predicts on average for data instances that have values close to  $v$  for that feature. The effect could be due to that feature, but also due to correlated features.”
- **ALE plots:** “Let me show you how the model predictions change in a small ‘window’ of the feature around  $v$  for data instances in that window.”

## 20.2 Theory

How do PDP, M-plot, and ALE plot differ mathematically? Common to all three methods is that they reduce the complex prediction function  $\hat{f}$  to a function that depends on only one (or two) features. All three methods reduce the function by averaging the effects of the other features, but they differ in whether averages of predictions or of **differences in predictions** are calculated and whether averaging is done over the marginal or conditional distribution.

Partial dependence plots average predictions over the marginal distribution.

$$\begin{aligned}\hat{f}_{S,PDP}(\mathbf{x}_S) &= \mathbb{E}_{X_C}[\hat{f}(\mathbf{x}_S, X_C)] \\ &= \int_{X_C} \hat{f}(\mathbf{x}_S, X_C) d\mathbb{P}(X_C)\end{aligned}$$

This is the value of the prediction function  $f$ , at feature value(s)  $\mathbf{x}_S$ , averaged over all features in  $X_C$  (here treated as random variables). Averaging means calculating the marginal expectation  $\mathbb{E}$  over the features in set  $C$ , which is the integral over the predictions weighted by the probability distribution. Sounds fancy, but to calculate the expected value over the marginal distribution, we simply take all our data instances, force them to have a certain grid value for the features in set  $S$ , and average the predictions for this manipulated dataset. This procedure ensures that we average over the marginal distribution of the features.

M-plots average predictions over the conditional distribution.

$$\begin{aligned}\hat{f}_{S,M}(\mathbf{x}_S) &= \mathbb{E}_{X_C|X_S} [\hat{f}(X_S, X_C)|X_S = \mathbf{x}_S] \\ &= \int_{X_C} \hat{f}(x_S, X_C) d\mathbb{P}(X_C|X_S = \mathbf{x}_S)\end{aligned}$$

The only thing that changes compared to PDPs is that we average the predictions conditional on each grid value of the feature of interest, instead of assuming the marginal distribution at each grid value. In practice, this means that we have to define a neighborhood; for example, for the calculation of the effect of  $30 \text{ m}^2$  on the predicted house value, we could average the predictions of all houses between  $28$  and  $32 \text{ m}^2$ .

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

$$\begin{aligned}\hat{f}_{S,ALE}(\mathbf{x}_S) &= \int_{\mathbf{z}_{0,S}}^{\mathbf{x}_S} \mathbb{E}_{X_C|X_S=\mathbf{x}_S} [\hat{f}^S(X_S, X_C)|X_S = \mathbf{z}_S] d\mathbf{z}_S - \text{constant} \\ &= \int_{\mathbf{z}_{0,S}}^{\mathbf{x}_S} \left( \int_{\mathbf{x}_C} \hat{f}^S(\mathbf{z}_S, X_C) d\mathbb{P}(X_C|X_S = \mathbf{z}_S) \right) d\mathbf{z}_S - \text{constant}\end{aligned}$$

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

$$\hat{f}^S(\mathbf{x}_S, \mathbf{x}_C) = \frac{\partial \hat{f}(\mathbf{x}_S, \mathbf{x}_C)}{\partial \mathbf{x}_S}$$

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

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

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

## 20.3 Estimation

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

First we estimate the uncentered effect:

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

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

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

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

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

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

### ALE plots for the interaction of two features

ALE plots can also show the interaction effect of two features. The calculation principles are the same as for a single feature, but we work with rectangular cells instead of intervals because we have to accumulate the effects in two dimensions. In addition to adjusting for the overall mean effect, we also adjust for the main effects of both features. This means that ALE for two features estimates the second-order effect, which does not include the main effects of the features. In other words, ALE for two features only shows the additional interaction effect of the two features. I spare you the formulas for 2D ALE plots because they are long and

unpleasant to read. If you are interested in the calculation, I refer you to the paper by Apley and Zhu (2020), formulas (13) - (16). I'll rely on visualizations to develop intuition about the second-order ALE calculation.

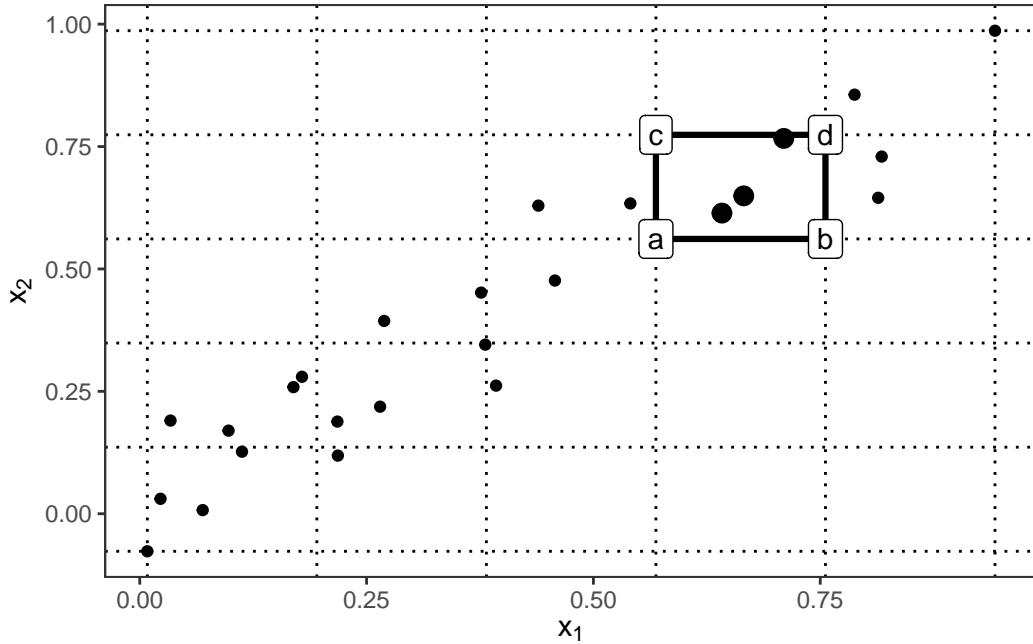


Figure 20.4: Calculation of 2D-ALE. We place a grid over the two features. In each grid cell we calculate the 2nd-order differences for all instance within. We first replace values of  $x_1$  and  $x_2$  with the values from the cell corners. If a, b, c and d represent the “corner”-predictions of a manipulated instance (as labeled in the graphic), then the 2nd-order difference is  $(d - c) - (b - a)$ . The mean 2nd-order difference in each cell is accumulated over the grid and centered.

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

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

effects in a partial dependence plot to get the pure main or second-order effects, or you can get an estimate of the total ALE plots by refraining from subtracting the lower-order effects.

The accumulated local effects could also be calculated for arbitrarily higher orders (interactions of three or more features), but as argued in the [PDP chapter](#), only up to two features makes sense because higher interactions cannot be visualized or even interpreted meaningfully.

### ALE for categorical features

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

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

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

#### 💡 Utilize natural order of categories

If there is a meaningful order to the categories of a categorical feature, use this order for ALE.

## 20.4 ALE versus PDP

Let's see ALE plots in action. I've constructed a scenario in which partial dependence plots fail. The scenario depicted in Figure 20.5 consists of a prediction model and two strongly correlated features. The prediction model is mostly a linear regression model but does something weird at a combination of the two features for which we have never observed instances. This "weird" area is far from the distribution of data (point cloud) and does not affect the performance of the model and, arguably, should not affect its interpretation.

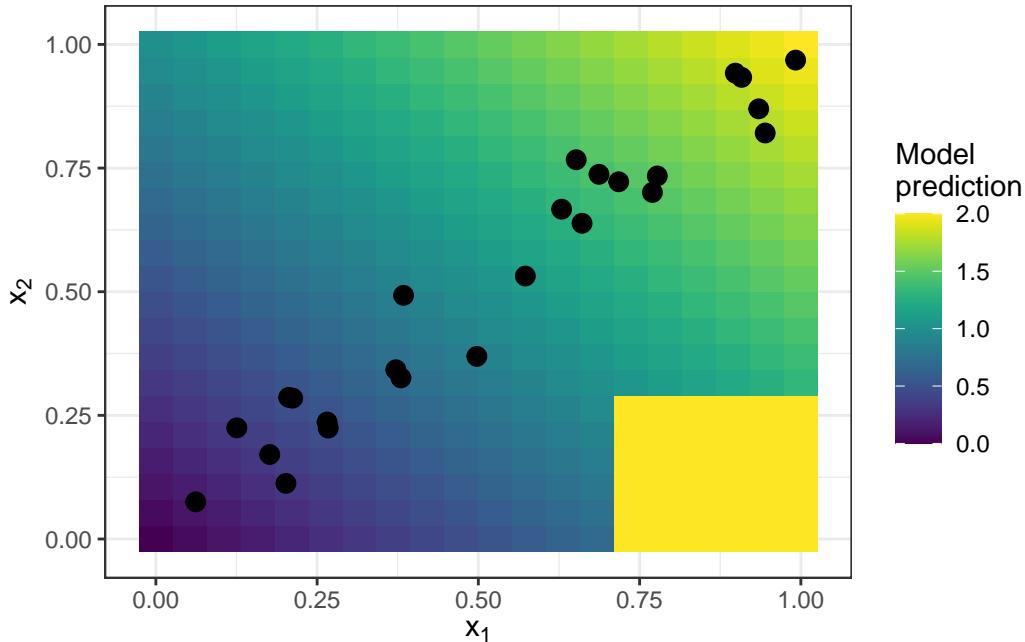


Figure 20.5: Two features and the predicted outcome. The model predicts the sum of the two features (shaded background), with the exception that if  $x_1 > 0.7$  and  $x_2 < 0.3$ , the model always predicts 2.

Is this a realistic, relevant scenario at all? When you train a model, the learning algorithm minimizes the loss for the existing training data instances. Weird stuff can happen outside the distribution of training data because the model is not penalized for doing weird stuff in these areas. Leaving the data distribution is called extrapolation, which can also be used to fool machine learning models, as described in the [chapter on adversarial examples](#). See Figure 20.6 how the partial dependence plots behave compared to ALE plots. The PDP estimates are influenced by the odd behavior of the model outside the data distribution (steep jumps in the plots). The ALE plots correctly identify that the machine learning model has a linear relationship between features and predictions, ignoring areas without data.

But is it not interesting to see that our model behaves oddly at  $x_1 > 0.7$  and  $x_2 < 0.3$ ? Well,

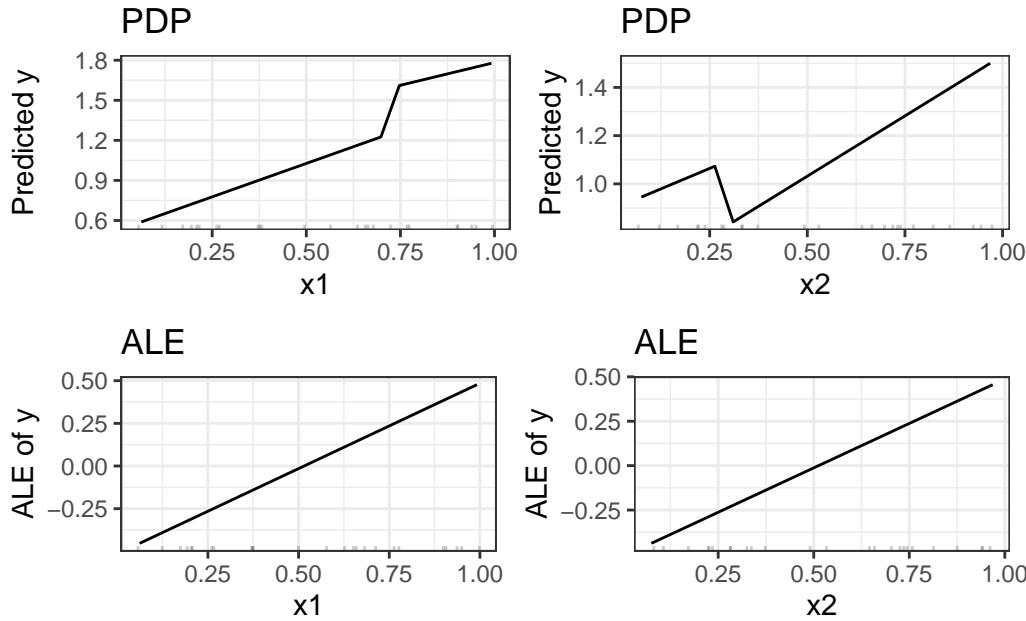


Figure 20.6: Feature effects computed with PDP (upper row) and ALE (lower row) for the simulated example.

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

Produce both PDP and ALE

Anecdotal observation: For my applications, the ALE and PDP plots looked quite similar, despite correlation. Correlation can ruin the interpretability, but it doesn't have to. If you ALE and PDP show the same curve for a correlated feature, just interpret the PDP since it has a simpler interpretation.

## 20.5 Examples

Turning to a real dataset, let's predict the [number of rented bikes](#) based on weather and day, and check if the ALE plots really work as well as promised. We train a random forest to predict the number of rented bikes and use ALE plots to analyze how wind speed and weather influence the predictions.

Figure 20.7 (left) shows that an increasing wind speed has a negative effect on bike rentals. For the weather situation (right), we see that especially bad weather has a strong negative effect on the number of rented bikes. Both effects align with domain knowledge, which is a good sign.

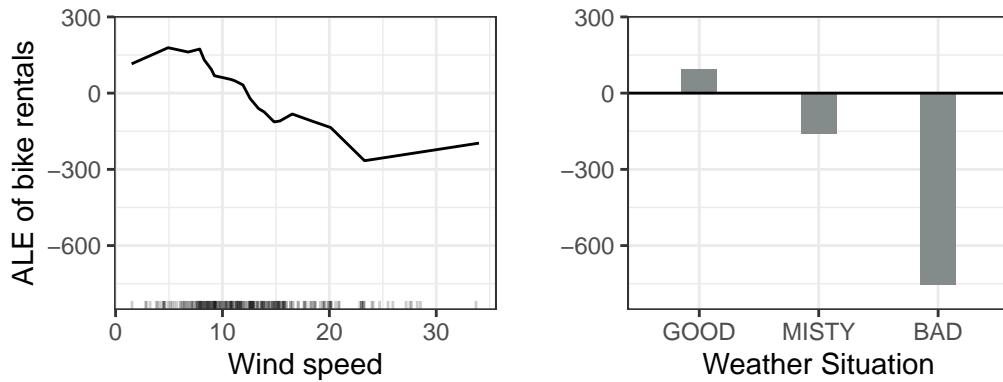


Figure 20.7: ALE plots for the bike prediction model for wind speed and weather situation.

Next, we consider the effects of humidity and temperature, and their interaction on the predicted number of bikes. Remember that the second-order effect is the additional interaction effect of the two features and does not include the main effects. This means that, for example, you will not see the main effect that high humidity leads to a lower number of predicted bikes on average in the second-order ALE plot. Figure 20.8 shows both the main effects of temperature and humidity, and their interaction. The plot reveals an interaction: cold and humid weather increases the prediction. Keep in mind that both main effects of humidity and temperature say that the predicted number of bikes decreases in very cold and humid weather. In cold and humid weather, the combined effect of temperature and humidity is therefore not the sum of the main effects, but larger than the sum.

You can also get the total interaction effect by adding the two main effects and the mean prediction on top. If you are only interested in the interaction, you should look at the second-order effects because the total effect mixes the main effects into the plot. But if you want to know the combined effect of the features, you should look at the total effect. However, in a scenario where two features have no interaction, the total effect of the two features could be misleading because it probably shows a complex landscape, suggesting some interaction, but it

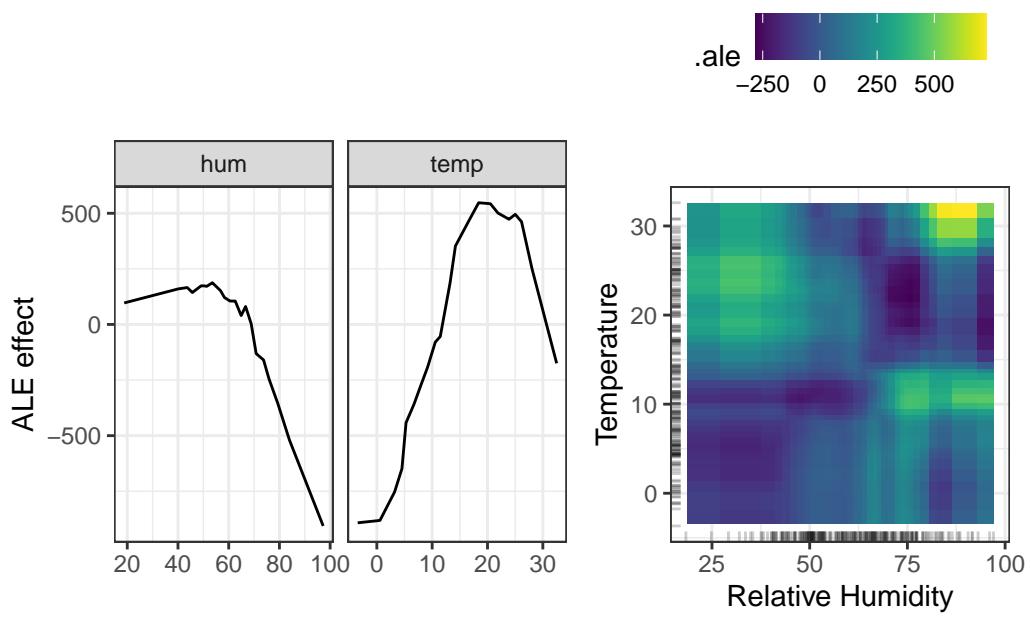


Figure 20.8: ALE plot for the 2nd-order effect of humidity and temperature on the predicted number of rented bikes. Lighter shade indicates an above average and darker shade a below average prediction when the main effects are already taken into account.

is simply the product of the two main effects. The pure second-order effect would immediately show that there is no interaction.

Enough bikes for now, let's turn to a classification task. We train a random forest to predict the  $\mathbb{P}(Y = \text{female})$  based on body measurements. How does the body weight affect the probability of a penguin being female? Figure 20.9 (left) visualizes the ALE plot for body mass. The heavier the penguin, the less likely it is to be female. But we get a better picture when we visualize the ALE plots by species (Figure 20.9 on the right). Each of the species has a clear cut-off point, above which the body mass is more typical for male penguins. In general, the effects are quite similar to the PDPs in [the PDP chapter](#).

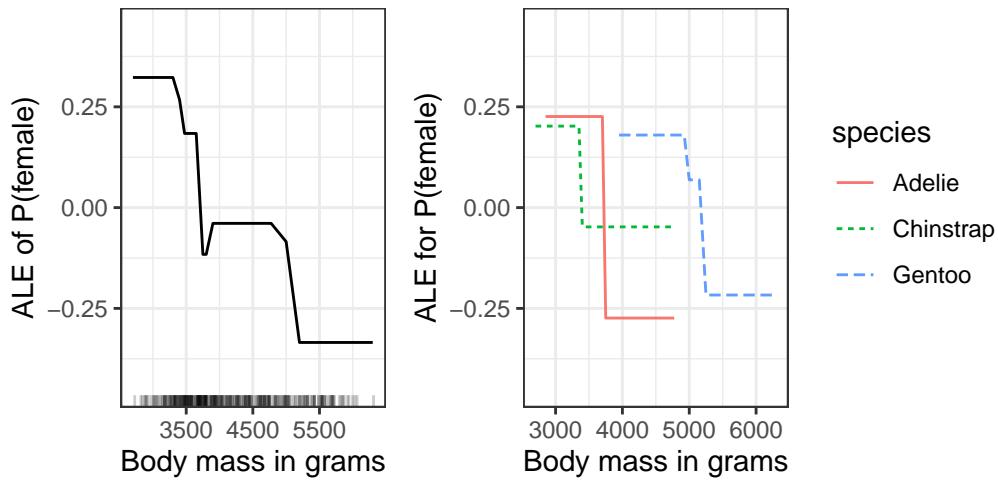


Figure 20.9: Left: ALE plot for body mass across all penguin species. Right: Separate body mass ALE plots for each species.

## 20.6 Strengths

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

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

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

The entire **prediction function can be decomposed** into a sum of lower-dimensional ALE functions, as explained in the chapter on [functional decomposition](#).

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

## 20.7 Limitations

An **interpretation of the effect across intervals is not permissible** if the features are strongly correlated. Consider the case where your features are highly correlated, and you are looking at the left end of a 1D-ALE plot. The ALE curve might invite the following misinterpretation: “The ALE curve shows how the prediction changes, on average, when we gradually change the value of the respective feature for a data instance, and keeping the instances’ other feature values fixed.” The effects are computed per interval (locally) and therefore the interpretation of the effect can only be local. For convenience, the interval-wise effects are accumulated to show a smooth curve, but keep in mind that each interval is created with different data instances.

ALE effects **may differ from the coefficients specified in a linear regression model** when features interact and are correlated. Grömping (2020) showed that in a linear model with two correlated features and an additional interaction term ( $\hat{f}(x) = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_1 x_2$ ), the first-order ALE plots do not show a straight line. Instead, they are slightly curved because they incorporate parts of the multiplicative interaction of the features. To understand what is happening here, I recommend reading the chapter on [functional decomposition](#). In short, ALE defines first-order (or 1D) effects differently than the linear formula describes them. This is not necessarily wrong, because when features are correlated, the attribution of interactions is not as clear. But it is certainly unintuitive that ALE and linear coefficients do not match.

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

Unlike PDPs, **ALE plots are not accompanied by ICE curves**. For PDPs, ICE curves are great because they can reveal heterogeneity in the feature effect, which means that the effect of a feature looks different for subsets of the data. For ALE plots, you can only check per

interval whether the effect is different between the instances, but each interval has different instances, so it is not the same as ICE curves.

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

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

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

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

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

The list of disadvantages has become quite long, but do not be fooled by the number of words I use. As a rule of thumb: Use PDP if features are not correlated. If features are correlated, but PDP and ALE show more or less the same curves, go for the simpler PDP interpretation. If features are correlated and PDP and ALE differ, go with the ALE plot.

## 20.8 Software and alternatives

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

ALE plots are implemented in R in the [ALEPlot R package](#) by the inventor himself, and once in the [iml package](#). ALE also has a couple of Python implementations: [ALEPython](#), [Alibi](#), and [PiML](#).

# 21 Feature Interaction

When features interact with each other in a prediction model, the prediction cannot be expressed as the sum of the feature effects because the effect of one feature depends on the value of the other feature. Aristotle's predicate "The whole is greater than the sum of its parts" applies in the presence of interactions.

## 21.1 What are feature interactions?

If a machine learning model makes a prediction based on two features, we can decompose the prediction into four terms: a constant term, a term for the first feature, a term for the second feature, and a term for the interaction between the two features. The interaction between two features is the change in the prediction that occurs by varying the features after considering the individual feature effects.

For example, a model predicts the value of a house, using house size (big or small) and location (good or bad) as features, which yields four possible predictions, see Table 21.1.

Table 21.1: Example predictions for house prices without interactions

Location	Size	Prediction
good	big	300,000
good	small	200,000
bad	big	250,000
bad	small	150,000

We decompose the model prediction into the following parts: A constant term (150,000), an effect for the size feature (+100,000 if big; +0 if small), and an effect for the location (+50,000 if good; +0 if bad). This decomposition fully explains the model predictions. There's no interaction effect because the model prediction is a sum of the single feature effects for size and location. When you make a small house big, the prediction always increases by 100,000, regardless of location. Also, the difference in prediction between a good and a bad location is 50,000, regardless of size.

Let's now look at an example with interaction in Table 21.2.

Table 21.2: Example predictions for house prices with interactions

Location	Size	Prediction
good	big	400,000
good	small	200,000
bad	big	250,000
bad	small	150,000

We decompose the prediction table into the following parts: A constant term (150,000), an effect for the size feature (+100,000 if big; +0 if small), and an effect for the location (+50,000 if good; +0 if bad). For this table, we need an additional term for the interaction: +100,000 if the house is big and in a good location. So for a big house in a good location, we have: 150,000 (base) + 50,000 (good location) + 100,000 (big) + 100,000 (interaction) = 400,000. This is an interaction between size and location because, in this case, the difference in prediction between a big and a small house depends on the location.

One way to estimate the interaction strength is to measure how much of the variation of the prediction depends on the interaction of the features. This measurement is called H-statistic, introduced by J. H. Friedman and Popescu (2008).

## 21.2 Friedman's H-statistic

We are going to deal with two cases: First, a two-way interaction measure that tells us whether and to what extent two features in the model interact with each other; second, a total interaction measure that tells us whether and to what extent a feature interacts in the model with all the other features. In theory, arbitrary interactions between any number of features can be measured, but these two are the most interesting cases.

If two features do not interact, we can decompose the [partial dependence function](#) as follows (assuming the partial dependence functions are centered at zero):

$$PD_{jk}(\mathbf{x}_j, \mathbf{x}_k) = PD_j(\mathbf{x}_j) + PD_k(\mathbf{x}_k)$$

where  $PD_{jk}(\mathbf{x}_j, \mathbf{x}_k)$  is the 2-way partial dependence function of both features, and  $PD_j(\mathbf{x}_j)$  and  $PD_k(\mathbf{x}_k)$  are the partial dependence functions of the single features.

Likewise, if a feature has no interaction with any of the other features, we can express the prediction function  $\hat{f}(\mathbf{x})$  as a sum of partial dependence functions, where the first summand depends only on  $j$  and the second on all other features except  $j$ :

$$\hat{f}(\mathbf{x}) = PD_j(x_j) + PD_{-j}(\mathbf{x}_{-j})$$

where  $PD_{-j}(\mathbf{x}_{-j})$  is the partial dependence function that depends on all features except the  $j$ -th feature.

This decomposition expresses the partial dependence (or full prediction) function without interactions (between features  $j$  and  $k$ , or respectively  $j$  and all other features). In a next step, we measure the difference between the observed partial dependence function and the decomposed one without interactions. We calculate the variance of the output of the partial dependence (to measure the interaction between two features) or of the entire function (to measure the interaction between a feature and all other features). The amount of the variance explained by the interaction (difference between observed and no-interaction PD) is used as an interaction strength statistic. The statistic is 0 if there is no interaction at all, and 1 if all of the variance of the  $PD_{jk}$  or  $\hat{f}$  is explained by the sum of the partial dependence functions. An interaction statistic of 1 between two features means that each single PD function is constant, and the effect on the prediction only comes through the interaction. The H-statistic can also be larger than 1, which is more difficult to interpret. This can happen when the variance of the 2-way interaction is larger than the variance of the 2-dimensional partial dependence plot.

Mathematically, the H-statistic proposed by Friedman and Popescu for the interaction between feature  $j$  and  $k$  is:

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

The same applies to measuring whether a feature  $j$  interacts with any other feature:

$$H_j^2 = \frac{\sum_{i=1}^n [\hat{f}(\mathbf{x}^{(i)}) - PD_j(x_j^{(i)}) - PD_{-j}(\mathbf{x}_{-j}^{(i)})]^2}{\sum_{i=1}^n (\hat{f}(\mathbf{x}^{(i)}))^2}$$

The H-statistic is expensive to evaluate because it iterates over all data points, and at each point the partial dependence has to be evaluated, which in turn is done with all  $n$  data points. In the worst case, we need  $2n^2$  calls to the machine learning model's predict function to compute the two-way H-statistic ( $j$  vs.  $k$ ) and  $3n^2$  for the total H-statistic ( $j$  vs. all). To speed up the computation, we can sample from the  $n$  data points. This has the disadvantage of increasing the variance of the partial dependence estimates, which makes the H-statistic unstable. So if you are using sampling to reduce the computational burden, make sure to sample enough data points.

Friedman and Popescu also propose a test statistic to evaluate whether the H-statistic differs significantly from zero. The null hypothesis is the absence of interaction. To generate the interaction statistic under the null hypothesis, you must be able to adjust the model so that it has no interaction between feature  $j$  and  $k$  or all others. This is not possible for all types

of models. Therefore, this test is model-specific, not model-agnostic, and as such not covered here.

The interaction strength statistic can also be applied in a classification setting if the prediction is a probability.

## 21.3 Examples

Let's see what feature interactions look like in practice! We analyze the interactions between features in a random forest trained to predict [penguin sex](#), given body measurements; see Figure 21.1 on the top. The body mass has the highest interaction strength. After looking at the feature interactions of each feature with all other features, we can select one of the features and dive deeper into all the 2-way interactions between the selected feature and the other features. Body mass has the strongest interaction, so let's take a deeper look in Figure 21.1 on the bottom. The plot shows that body mass interacts mostly with bill depth and species.

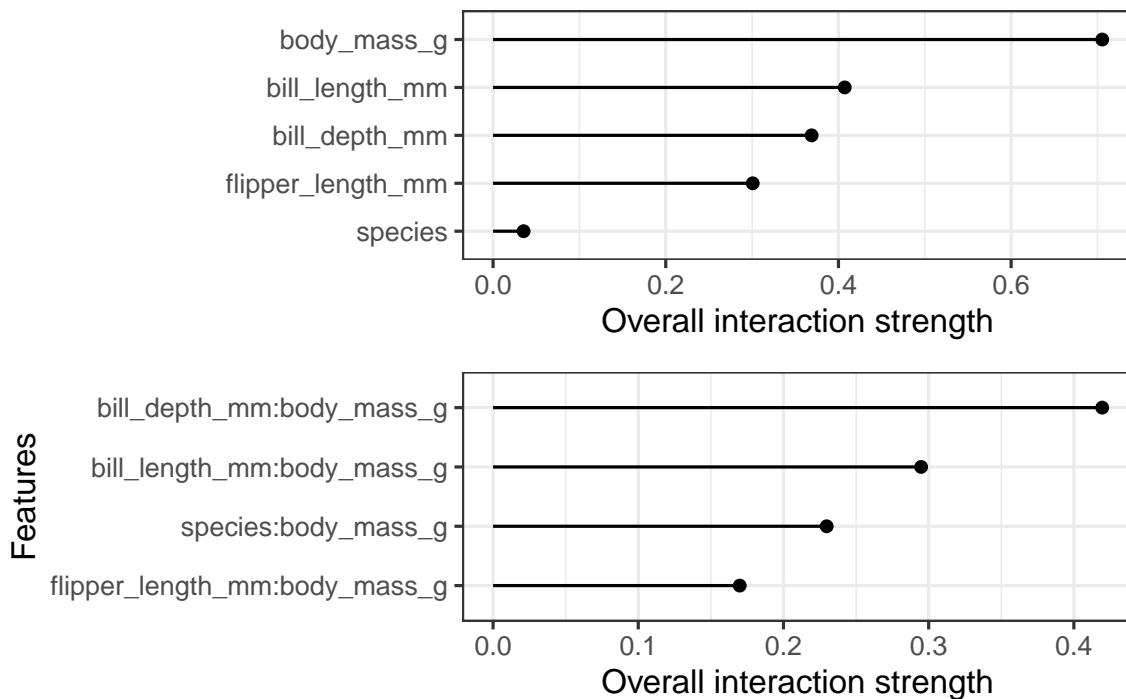


Figure 21.1: Top: The interaction strength (H-statistic) for each feature with all other features for a random forest predicting P(female). Bottom: The 2-way interaction strengths (H-statistic) between body mass and all other features.

Bonus: We are interested in all interaction by species, which I visualize in Figure 21.2. Especially for body mass, the interaction strengths differ between species.

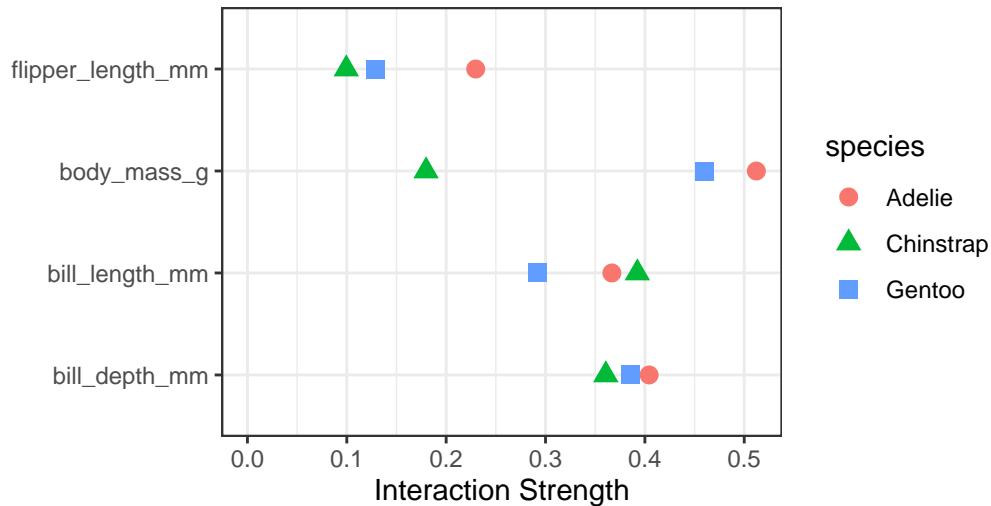


Figure 21.2: The interaction strength (H-statistic) for each feature with all other features for a random forest by species predicting  $P(\text{female})$ .

## 21.4 Strengths

The interaction H-statistic has an **underlying theory** through the partial dependence decomposition.

The H-statistic has a **meaningful interpretation**: The interaction is defined as the share of variance that is explained by the interaction.

Since the statistic is **dimensionless**, it is comparable across features and even across models.

The statistic **detects all kinds of interactions**, regardless of their particular form.

With the H-statistic, it's also possible to analyze arbitrary **higher-order interactions** such as the interaction strength between 3 or more features.

## 21.5 Limitations

The first thing you will notice: The interaction H-statistic takes a long time to compute because it's **computationally expensive**.

The computation involves estimating marginal distributions. These **estimates also have a certain variance** if we do not use all data points. This means that as we sample points, the estimates also vary from run to run, and the **results can be unstable**. I recommend repeating the H-statistic computation a few times to see if you have enough data to get a stable result.

It's unclear whether an interaction is significantly greater than 0. We would need to conduct a statistical test, but this **test is not (yet) available in a model-agnostic version**.

Concerning the test problem, it is difficult to say when the H-statistic is large enough for us to consider an interaction "strong."

Also, the **H-statistic can be larger than 1**, which makes the interpretation difficult.

When the total effect of two features is weak but mostly consists of interactions, then the H-statistic will be very large. These spurious interactions require a small denominator of the H-statistic and are made worse when features are correlated. A **spurious interaction can be over-interpreted** as a strong interaction effect when, in reality, both features play a minor role in the model. A possible remedy is to visualize the unnormalized version of the H-statistic, which is the square root of the numerator of the H-statistic (Inglis, Parnell, and Hurley 2022). This scales the H-statistic to the same level as the response, at least for regression, and puts less emphasis on spurious interactions.

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

The H-statistic tells us the strength of interactions, **but it does not tell us how the interactions look like**. 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 you are interested in.

The interaction statistic works under the assumption that we can shuffle features independently. If the features correlate strongly, the assumption is violated and **we integrate over feature combinations that are very unlikely in reality**. That's the same problem that partial dependence plots have. Correlated features can lead to large values of the H-statistic.

Sometimes the results are strange, and for small simulations **do not yield the expected results**. But this is more of an anecdotal observation.

## 21.6 Software and alternatives

For the examples in this book, I used the R package `iml`, which is available on [CRAN](#) and the development version on [GitHub](#). There are other implementations, which focus on specific models: The R package `pre` implements [RuleFit](#) and the H-statistic. The R package

[gbm](#) implements gradient boosted models and the H-statistic. In Python, you can find an implementation in the [PiML package](#).

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

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

Partial dependence-based feature interaction by Greenwell, Boehmke, and McCarthy (2018) measures the interaction between two features. This approach measures the feature importance (defined as the variance of the partial dependence function) of one feature conditional on different, fixed points of the other feature. If the variance is high, then the features interact with each other; if it's zero, they do not interact. The corresponding R package [vip](#) is available on [GitHub](#). The package also covers partial dependence plots and feature importance.

## 22 Functional Decomposition

A supervised machine learning model can be viewed as a function that takes a high-dimensional feature vector as input and produces a prediction or classification score as output. Functional decomposition is an interpretation technique that deconstructs the high-dimensional function and expresses it as a sum of individual feature effects and interaction effects that can be visualized. In addition, functional decomposition is a fundamental principle underlying many interpretation techniques – it helps you better understand other interpretation methods.

Let's jump right in and look at a particular function. This function takes two features as input and produces a one-dimensional output:

$$\hat{f}(x_1, x_2) = 2 + e^{x_1} - x_2 + x_1 \cdot x_2$$

Think of the function as a machine learning model. We can visualize the function with a 3D plot or a heatmap with contour lines as in Figure 22.1.

The function takes large values when  $x_1$  is large and  $x_2$  is small, and it takes small values for large  $x_2$  and small  $x_1$ . The prediction function is not simply an additive effect between the two features, but an interaction between the two. The presence of an interaction can be seen in the figure – the effect of changing values for feature  $x_1$  depends on the value that feature  $x_2$  has.

Our job now is to decompose this function into main effects of features  $X_1$  and  $X_2$  and an interaction term. For a two-dimensional function  $\hat{f}$  that depends on only two input features:  $\hat{f}(x_1, x_2)$ , we want each component to represent a main effect ( $\hat{f}_1$  and  $\hat{f}_2$ ), interaction ( $\hat{f}_{1,2}$ ), or intercept ( $\hat{f}_0$ ):

$$\hat{f}(x_1, x_2) = \hat{f}_0 + \hat{f}_1(x_1) + \hat{f}_2(x_2) + \hat{f}_{1,2}(x_1, x_2)$$

The main effects indicate how each feature affects the prediction, independent of the values the other feature. The interaction effect indicates the joint effect of the features. The intercept is a fixed value that is part of all predictions. If all feature values were set to zero, the prediction would only consist of the intercept. Note that the components themselves are functions (except for the intercept) with different input dimensionality.

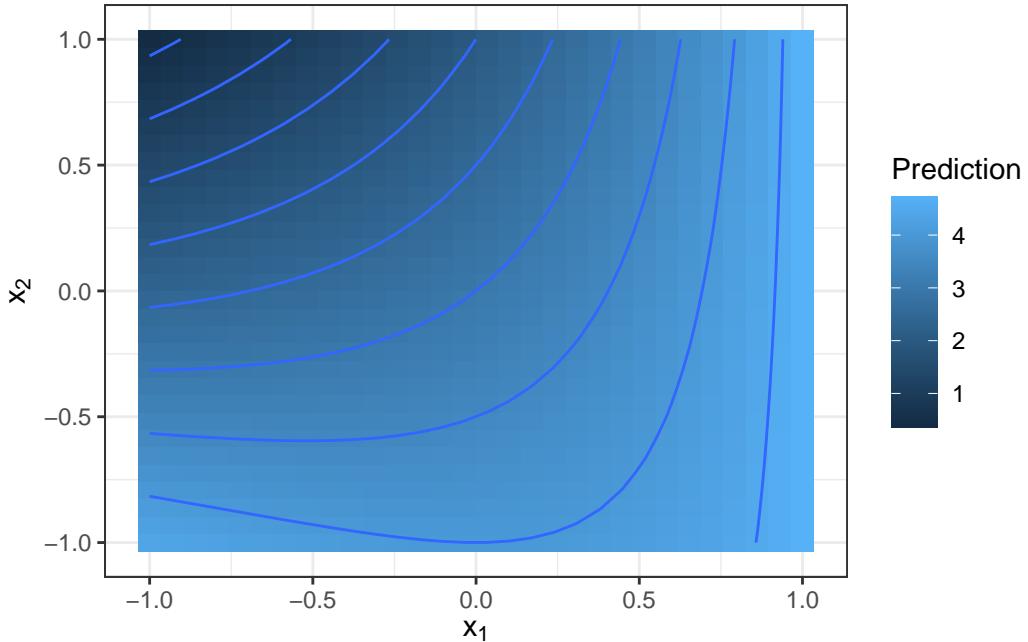


Figure 22.1: Prediction surface of a function with two features  $X_1$  and  $X_2$ . The lighter the color, the larger the prediction.

I'll just give you the components now and explain where they come from later. The intercept is given as  $\hat{f}_0 \approx 3.18$ . Since the other components are functions, we can visualize them in Figure 22.2.

Do you think the components make sense given the above true formula, ignoring that the intercept value seems a bit random? The  $X_1$  feature shows an exponential main effect, and  $X_2$  shows a negative linear effect. The interaction term looks a bit like a Pringles chip. In less crunchy and more mathematical terms, it's a hyperbolic paraboloid, as we would expect for  $x_1 \cdot x_2$ . Spoiler alert: the decomposition is based on accumulated local effect plots, which we will discuss later in the chapter.

But why all the excitement? A glance at the formula already gives us the answer to the decomposition, so no need for fancy methods, right? For feature  $X_1$ , we can take all the summands that contain only  $x_1$  as the component for that feature. That would be  $\hat{f}_1(x_1) = e^{x_1}$  and  $\hat{f}_2(x_2) = -x_2$  for feature  $X_2$ . The interaction is then  $\hat{f}_{12}(x_1, x_2) = x_1 \cdot x_2$ . While this is the correct answer for this example (up to constants), there are two problems with this approach: Problem 1): While the example started with the formula, the reality is that only structurally simple machine learning models can be described with such a neat formula. Problem 2) is much more intricate and concerns what an interaction is. Imagine a simple function  $\hat{f}(x_1, x_2) = x_1 \cdot x_2$ , where both features take values larger than zero and are independent of each other. Using our look-at-the-formula tactic, we would conclude that there is an interaction between

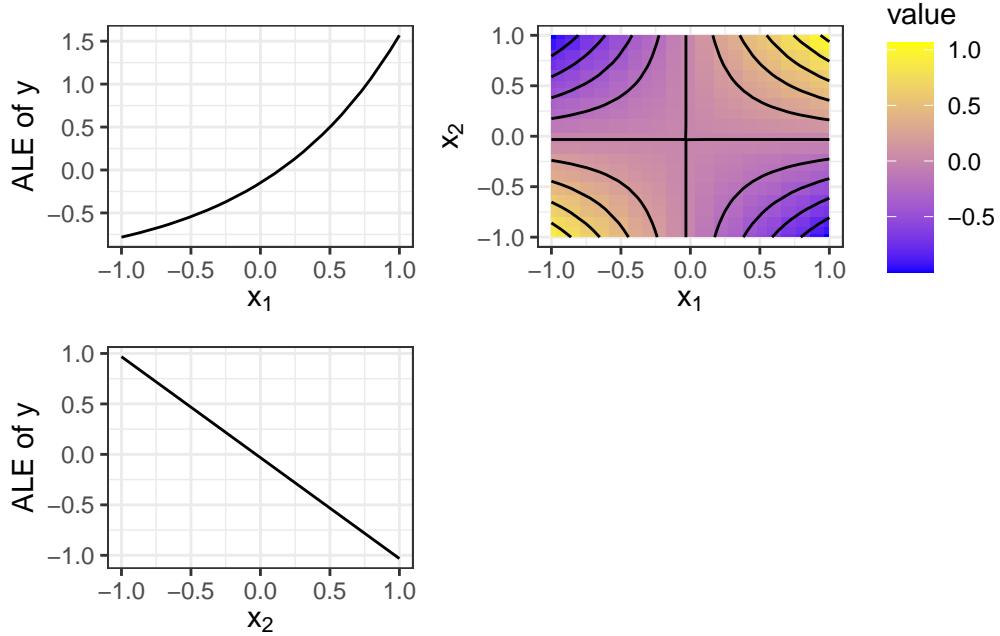


Figure 22.2: Decomposition of a 2-dimensional function into main effects and interaction. Top row: Main effects  $f_1$  (left) and  $f_2$ . Bottom: Interaction  $f_{12}$  between  $X_1$  and  $X_2$

features  $X_1$  and  $X_2$ , but not individual feature effects. But can we really say that feature  $x_1$  has no individual effect on the prediction function? Regardless of what value  $x_2$  takes on, the prediction increases as we increase  $x_1$ . For example, for  $x_2 = 1$ , the effect of  $x_1$  is  $\hat{f}(x_1, 1) = x_1$ , and when  $x_2 = 10$  the effect is  $\hat{f}(x_1, 10) = 10 \cdot x_1$ . Thus, it's clear that feature  $X_1$  has a positive effect on the prediction, independent of  $X_2$ , and is not zero.

To solve problem 1) of lack of access to a neat formula, we need a method that uses only the prediction function or classification score. To solve problem 2) of lack of definition, we need some axioms that tell us what the components should look like and how they relate to each other. But first, we should define more precisely what functional decomposition is.

## 22.1 Decomposing a function

A prediction function takes  $p$  features as input,  $\hat{f} : \mathbb{R}^p \mapsto \mathbb{R}$ , and produces an output. This can be a regression function, but it can also be the classification probability for a given class or the score for a given cluster (unsupervised machine learning). Fully decomposed, we can represent the prediction function as the sum of functional components:

$$\begin{aligned}
\hat{f}(\mathbf{x}) = & \hat{f}_0 + \hat{f}_1(x_1) + \dots + \hat{f}_p(x_p) \\
& + \hat{f}_{1,2}(x_1, x_2) + \dots + \hat{f}_{1,p}(x_1, x_p) + \dots + \hat{f}_{p-1,p}(x_{p-1}, x_p) \\
& + \dots \\
& + \hat{f}_{1,\dots,p}(x_1, \dots, x_p)
\end{aligned}$$

We can make the decomposition formula a bit nicer by indexing all possible subsets of feature combinations:  $S \subseteq \{1, \dots, p\}$ . This set contains the intercept ( $S = \emptyset$ ), main effects ( $|S| = 1$ ), and all interactions ( $|S| \geq 1$ ). With this subset defined, we can write the decomposition as follows:

$$\hat{f}(\mathbf{x}) = \sum_{S \subseteq \{1, \dots, p\}} \hat{f}_S(\mathbf{x}_S)$$

In the formula,  $\mathbf{x}_S$  is the vector of features in the index set  $S$ . And each subset  $S$  represents a functional component, for example, a main effect if  $S$  contains only one feature, or an interaction if  $|S| > 1$ .

How many components are in the above formula? The answer boils down to how many possible subsets  $S$  of the features  $1, \dots, p$  we can form. And these are  $\sum_{i=0}^p \binom{p}{i} = 2^p$  possible subsets! For example, if a function uses 10 features, we can decompose the function into 1,024 components: 1 intercept, 10 main effects, 90 2-way interaction terms, 720 3-way interaction terms, ... And with each additional feature, the number of components doubles. Clearly, for most functions, it is not feasible to compute all components. Another reason NOT to compute all components is that components with  $|S| > 2$  are difficult to visualize and interpret.

So far I've avoided talking about how the components are defined and computed. The only constraints we have implicitly talked about were the number and dimensionality of the components, and that the sum of components should yield the original function. But without further constraints on what the components should be, they are not unique. This means we could shift effects between main effects and interactions, or lower-order interactions (few features) and higher-order interactions (more features). In the example at the beginning of the chapter, we could set both main effects to zero and add their effects to the interaction effect.

Here's an even more extreme example that illustrates the need for constraints on components. Suppose you have a 3-dimensional function. It does not really matter what this function looks like, but the following decomposition would **always** work:  $\hat{f}_0$  is 0.12.  $\hat{f}_1(\mathbf{x}) = 2 \cdot x_1 + \text{number of shoes you own}$ .  $\hat{f}_2, \hat{f}_3, \hat{f}_{1,2}, \hat{f}_{2,3}, \hat{f}_{1,3}$  are all zero. And to make this trick work, I define  $\hat{f}_{1,2,3}(x_1, x_2, x_3) = \hat{f}(\mathbf{x}) - \sum_{S \subseteq \{1, \dots, p\}} \hat{f}_S(\mathbf{x}_S)$ . So the interaction term containing all features simply sucks up all the remaining effects, which by definition always works, in the sense that the sum of all components gives us the original prediction function. This

decomposition would not be very meaningful and quite misleading if you were to present this as the interpretation of your model.

The ambiguity can be avoided by specifying further constraints or specific methods for computing the components. In this chapter, we will discuss different approaches to functional decomposition:

- (Generalized) functional ANOVA
- Accumulated Local Effects
- Statistical regression models
- Decomposing tree ensembles

## 22.2 Functional ANOVA

Functional ANOVA was proposed by Hooker (2004). A requirement for this approach is that the model prediction function  $\hat{f}$  is square integrable. As with any functional decomposition, the functional ANOVA decomposes the function into components:

$$\hat{f}(\mathbf{x}) = \sum_{S \subseteq \{1, \dots, p\}} \hat{f}_S(\mathbf{x}_S)$$

Hooker (2004) defines each component with the following formula:

$$\hat{f}_S(\mathbf{x}) = \int_{X_{-S}} \left( \hat{f}(\mathbf{x}) - \sum_{V \subset S} \hat{f}_V(\mathbf{x}) \right) d\mathbb{P}(X_{-S})$$

Okay, let's take this thing apart. We can rewrite the component as:

$$\hat{f}_S(\mathbf{x}) = \int_{X_{-S}} (\hat{f}(\mathbf{x})) d\mathbb{P}(X_{-S}) - \int_{X_{-S}} \left( \sum_{V \subset S} \hat{f}_V(\mathbf{x}) \right) d\mathbb{P}(X_{-S})$$

On the left side is the integral over the prediction function with respect to the features excluded from the set  $S$ , denoted with  $-S$ . For example, if we compute the 2-way interaction component for features 2 and 3, we would integrate over features 1, 4, 5, ... The integral can also be viewed as the expected value of the prediction function with respect to  $X_{-S}$ , assuming that all features follow a uniform distribution from their minimum to their maximum. From this interval, we subtract all components with subsets of  $S$ . This subtraction removes the effect of all lower-order effects and centers the effect. For  $S = \{1, 2\}$ , we subtract the main effects of both features  $\hat{f}_1$  and  $\hat{f}_2$ , as well as the intercept  $\hat{f}_0$ . The occurrence of these lower-order effects makes the formula recursive: We have to go through the hierarchy of subsets to the intercept

and compute all these components. For the intercept component  $\hat{f}_0$ , the subset is the empty set  $S = \{\emptyset\}$  and therefore  $-S$  contains all features:

$$\hat{f}_0(\mathbf{x}) = \int_X \hat{f}(\mathbf{x}) d\mathbb{P}(X)$$

This is simply the prediction function integrated over all features. The intercept can also be interpreted as the expectation of the prediction function when we assume that all features are uniformly distributed. Now that we know  $\hat{f}_0$ , we can compute  $\hat{f}_1$  (and equivalently  $\hat{f}_2$ ):

$$\hat{f}_1(\mathbf{x}) = \int_{X_{-1}} (\hat{f}(\mathbf{x}) - \hat{f}_0) d\mathbb{P}(X_{-1})$$

To finish the calculation for the component  $\hat{f}_{1,2}$ , we can put everything together:

$$\begin{aligned}\hat{f}_{1,2}(\mathbf{x}) &= \int_{X_{3,4}} (\hat{f}(\mathbf{x}) - (\hat{f}_0 + \hat{f}_1(\mathbf{x}) - \hat{f}_0 + \hat{f}_2(\mathbf{x}) - \hat{f}_0)) d\mathbb{P}(X_3, X_4) \\ &= \int_{X_{3,4}} (\hat{f}(\mathbf{x}) - \hat{f}_1(\mathbf{x}) - \hat{f}_2(\mathbf{x}) + \hat{f}_0) d\mathbb{P}(X_3, X_4)\end{aligned}$$

This example shows how each higher-order effect is defined by integrating over all other features, but also by removing all the lower-order effects that are subsets of the feature set we are interested in.

Hooker (2004) has shown that this definition of functional components satisfies these desirable axioms:

- Zero Means:  $\int \hat{f}_S(\mathbf{x}_S) d\mathbb{P}(X_S) = 0$  for each  $S \neq \emptyset$ .
- Orthogonality:  $\int \hat{f}_S(\mathbf{x}_S) \hat{f}_V(\mathbf{x}_V) d\mathbb{P}(X) = 0$  for  $S \neq V$ .
- Variance Decomposition: Let  $\sigma_{\hat{f}}^2 = \int \hat{f}(\mathbf{x})^2 d\mathbb{P}(X)$ , then  $\sigma^2(\hat{f}) = \sum_{S \subseteq P} \sigma_S^2(\hat{f}_S)$ , where  $P = \{1, \dots, p\}$ .

The zero means axiom implies that all effects or interactions are centered around zero. As a consequence, the interpretation at a position  $\mathbf{x}$  is relative to the centered prediction and not the absolute prediction.

The orthogonality axiom implies that components do not share information. For example, the first-order effect of feature  $X_1$  and the interaction term of  $X_1$  and  $X_2$  are not correlated. Because of orthogonality, all components are “pure” in the sense that they do not mix effects. It makes a lot of sense that the component for, say, feature  $X_4$  should be independent of the interaction term between features  $X_1$  and  $X_2$ . The more interesting consequence arises for

orthogonality of hierarchical components, where one component contains features of another, for example, the interaction between  $X_1$  and  $X_2$ , and the main effect of feature  $X_1$ . In contrast, a two-dimensional partial dependence plot for  $X_1$  and  $X_2$  would contain four effects: the intercept, the two main effects of  $X_1$  and  $X_2$ , and the interaction between them. The functional ANOVA component for  $\hat{f}_{1,2}(x_1, x_2)$  contains only the pure interaction.

Variance decomposition allows us to divide the variance of the function  $\hat{f}$  among the components and guarantees that it adds up to the total variance of the function in the end. The variance decomposition property can also explain to us why the method is called functional ANOVA. In statistics, ANOVA stands for ANalysis Of VAriance. ANOVA refers to a collection of methods that analyze differences in the mean of a target variable. ANOVA works by dividing the variance and attributing it to the variables. Functional ANOVA can therefore be seen as an extension of this concept to any function.

Problems arise with the functional ANOVA when features are correlated. As a solution, Hooker (2007) proposed the generalized functional ANOVA.

## 22.3 Generalized Functional ANOVA for dependent features

Similar to most interpretation techniques based on sampling data (such as the PDP), the functional ANOVA can produce misleading results when features are correlated. If we integrate over the uniform distribution, when in reality features are dependent, we create a new dataset that deviates from the joint distribution and extrapolates to unlikely combinations of feature values.

Hooker (2007) proposed the generalized functional ANOVA, a decomposition that works for dependent features. It's a generalization of the functional ANOVA we encountered earlier, which means that the functional ANOVA is a special case of the generalized functional ANOVA. The components are defined as projections of  $\hat{f}$  onto the space of additive functions:

$$\hat{f}_S(\mathbf{x}_S) = \arg \min_{g_S \in L^2(\mathbb{R}^S)} \int \left( \hat{f}(\mathbf{x}) - \sum_{S \subset P} g_S(\mathbf{x}_S) \right)^2 w(\mathbf{x}) d\mathbf{x}.$$

Instead of orthogonality, the components satisfy a hierarchical orthogonality condition:

$$\forall \hat{f}_S(\mathbf{x}_S) | S \subset U : \int \hat{f}_S(\mathbf{x}_S) \hat{f}_U(\mathbf{x}_U) w(\mathbf{x}) d\mathbf{x} = 0$$

Hierarchical orthogonality is different from orthogonality. For two feature sets  $S$  and  $U$ , neither of which is a subset of the other (e.g.,  $S = \{1, 2\}$  and  $U = \{2, 3\}$ ), the components  $\hat{f}_S$  and  $\hat{f}_U$  need not be orthogonal for the decomposition to be hierarchically orthogonal. But all components for all subsets of  $S$  must be orthogonal to  $\hat{f}_S$ . As a result, the interpretation

differs in relevant ways: Similar to the M-Plot in the [ALE chapter](#), generalized functional ANOVA components can entangle the (marginal) effects of correlated features. Whether the components entangle the marginal effects also depends on the choice of weight function  $w(\mathbf{x})$ . If we choose  $w$  to be the uniform measure on the unit cube, we obtain the functional ANOVA from the section above. A natural choice for  $w$  is the joint probability distribution function. However, the joint distribution is usually unknown and difficult to estimate. A trick can be to start with the uniform measure on the unit cube and cut out areas without data.

The estimation is done on a grid of points in the feature space and is stated as a minimization problem that can be solved using regression techniques. However, the components cannot be computed independently of each other, nor hierarchically, but a complex system of equations involving other components has to be solved. The computation is therefore quite complex and computationally intensive.

## 22.4 Accumulated Local Effects

ALE plots (Apley and Zhu 2020) also provide a functional decomposition, meaning that adding all ALE plots from intercept, 1D ALE plots, 2D ALE plots, and so on yields the prediction function. ALE differs from the (generalized) functional ANOVA, as the components are not orthogonal but, as the authors call it, pseudo-orthogonal. To understand pseudo-orthogonality, we have to define the operator  $H_S$ , which takes a function  $\hat{f}$  and maps it to its ALE plot for feature subset  $S$ . For example, the operator  $H_{1,2}$  takes as input a machine learning model and produces the 2D ALE plot for features 1 and 2:  $H_{1,2}(\hat{f}) = \hat{f}_{ALE,12}$ . If we apply the same operator twice, we get the same ALE plot. After applying the operator  $H_{1,2}$  to  $\hat{f}$  once, we get the 2D ALE plot  $\hat{f}_{ALE,12}$ . Then we apply the operator again, not to  $\hat{f}$  but to  $\hat{f}_{ALE,12}$ . This is possible because the 2D ALE component is itself a function. The result is again  $\hat{f}_{ALE,12}$ , meaning we can apply the same operator several times and always get the same ALE plot. This is the first part of pseudo-orthogonality. But what is the result if we apply two different operators for different feature sets? For example,  $H_{1,2}$  and  $H_1$ , or  $H_{1,2}$  and  $H_{3,4,5}$ ? The answer is zero. If we first apply the ALE operator  $H_S$  to a function and then apply  $H_U$  to the result (with  $S \neq U$ ), then the result is zero. In other words, the ALE plot of an ALE plot is zero, unless you apply the same ALE plot twice. Or in other words, the ALE plot for the feature set  $S$  does not contain any other ALE plots in it. Or in mathematical terms, the ALE operator maps functions to orthogonal subspaces of an inner product space.

As Apley and Zhu (2020) note, pseudo-orthogonality may be more desirable than hierarchical orthogonality because it does not entangle marginal effects of the features. Furthermore, ALE does not require estimation of the joint distribution; the components can be estimated in a hierarchical manner, which means that calculating the 2D ALE for features 1 and 2 requires only the calculations of individual ALE components of 1 and 2, and the intercept term in addition.

Does the [Partial Dependence Plot](#) also provide a functional decomposition? Short answer: No. Longer answer: The partial dependence plot for a feature set  $S$  always contains all effects of the hierarchy – the PDP for  $\{1, 2\}$  contains not only the interaction, but also the individual feature effects. As a consequence, adding all PDPs for all subsets does not yield the original function, and thus is not a valid decomposition. But could we adjust the PDP, perhaps by removing all lower effects? Yes, we could, but we would get something similar to the functional ANOVA. However, instead of integrating over a uniform distribution, the PDP integrates over the marginal distribution of  $X_{-S}$ , which is estimated using Monte Carlo sampling.

## 22.5 Decomposing tree ensembles

Z. Yang et al. (2024) proposed a functional decomposition of tree ensembles, for example, trained with XGBoost. Their proposal consists of two parts: a decomposition procedure and a set of training constraints to make the decomposition more interpretable.

The strategy to get from an ensemble of trees to the functional composition involves aggregation, purification, and attribution. First, each tree is decomposed into decision rules, with each leaf node becoming a decision rule. These rules are then sorted by the features they use. For example, all rules using only feature  $X_1$  are collected and used to estimate the main effect  $\hat{f}_1$ . The same is done for all other main effects and interactions. However, this decomposition wouldn't be unique since, for example, you could absorb the main effect of  $X_1$  into the interaction term of  $X_1$  and  $X_2$ . They “purify” the components to enforce zero means and orthogonality constraints (should sound familiar by now). In the attribution step, feature contributions for each data point are computed and aggregated across the data to get global importance values for each feature. The individual contributions are the same as the [Shapley values](#).

The other suggestion in the paper is to add constraints to the training so that the functional decomposition can be better interpreted. This includes simple things like setting the maximum tree depth low so that you control the maximum number of features that can interact. For example, setting the maximum depth to two makes the model have only main effects and two-way interactions, but no higher-order interactions. Other suggestions include monotonicity constraints, reduced number of bins, and interaction constraints. They also introduce a post-processing step for pruning effects from the functional decomposition.

## 22.6 Statistical regression models

This approach ties in with interpretable models, in particular [generalized additive models](#). Instead of decomposing a complex function, we can build constraints into the modeling process so that we can easily read out the individual components. While decomposition can be handled in a top-down manner, where we start with a high-dimensional function and decompose it,

generalized additive models provide a bottom-up approach, where we build the model from simple components. Both approaches have in common that their goal is to provide individual and interpretable components. In statistical models, we restrict the number of components so that not all  $2^p$  components have to be fitted. The simplest version is linear regression:

$$\hat{f}(\mathbf{x}) = \beta_0 + \beta_1 x_1 + \dots + \beta_p x_p$$

The formula looks very similar to the functional decomposition, but with two major modifications. Modification 1: All interaction effects are excluded, and we keep only the intercept and main effects. Modification 2: The main effects may only be linear in the features:  $\hat{f}_j(x_j) = \beta_j x_j$ . Viewing the linear regression model through the lens of functional decomposition, we see that the model itself represents a functional decomposition of the true function that maps from features to target, but under strong assumptions that the effects are linear effects and there are no interactions.

The generalized additive model relaxes the second assumption by allowing more flexible functions  $\hat{f}_j$  through the use of splines. Interactions can also be added, but this process is rather manual. Approaches such as GA2M attempt to add 2-way interactions automatically to a GAM (Caruana et al. 2015).

Thinking of a linear regression model or a GAM as functional decomposition can also lead to confusion. If you apply the decomposition approaches from earlier in the chapter (generalized functional ANOVA and accumulated local effects), you may get components that are different from the components read directly from the GAM. This can happen when interaction effects of correlated features are modeled in the GAM. The discrepancy occurs because other functional decomposition approaches split effects differently between interactions and main effects.

So when should you use GAMs instead of a complex model + decomposition? You should stick to GAMs when most interactions are zero, especially when there are no interactions with three or more features. If we know that the maximum number of features involved in interactions is two ( $|S| \leq 2$ ), then we can use approaches like MARS or GA2M. Ultimately, model performance on test data may indicate whether a GAM is sufficient or a more complex model performs much better.

## 22.7 Strengths

I consider functional decomposition to be a **key concept of machine learning interpretability** that helps to better understand many other methods.

Functional decomposition gives us a **theoretical justification** for decomposing high-dimensional and complex machine learning models into individual effects and interactions – a necessary step that allows us to interpret individual effects. Functional decomposition is the

core idea for techniques such as statistical regression models, ALE, (generalized) functional ANOVA, PDP, the H-statistic, and ICE curves.

Functional decomposition also provides a **better understanding of other methods**. For example, [permutation feature importance](#) breaks the association between a feature and the target. Viewed through the functional decomposition lens, we can see that the permutation “destroys” the effect of all components in which the feature was involved. This affects the main effect of the feature, but also all interactions with other features. As another example, Shapley values decompose a prediction into additive effects of the individual features. But the functional decomposition tells us that there should also be interaction effects in the decomposition, so where are they? Shapley values provide a fair attribution of effects to the individual features, meaning that all interactions are also fairly attributed to the features and therefore divided up among the Shapley values.

When considering functional decomposition as a tool, the use of **ALE plots offers many advantages**. ALE plots provide a functional decomposition that is fast to compute, has software implementations (see the ALE chapter), and desirable pseudo-orthogonality properties.

## 22.8 Limitations

The concept of functional decomposition quickly reaches its **limits for high-dimensional components** beyond interactions between two features. Not only does this exponential explosion in the number of features limit practicability, since we cannot easily visualize higher-order interactions, but computational time is insane if we were to compute all interactions.

Each method of functional decomposition has its **individual disadvantages**. The bottom-up approach – constructing regression models – is a quite manual process and imposes many constraints on the model that can affect predictive performance. Functional ANOVA requires independent features. Generalized functional ANOVA is very difficult to estimate. Accumulated local effect plots do not provide a variance decomposition.

The functional decomposition approach is **more appropriate for analyzing tabular data than text or images**.

# 23 Permutation Feature Importance

Permutation feature importance (PFI) measures the increase in the prediction error of the model after we permute the values of the feature, which breaks the relationship between the feature and the true outcome.

The concept is really straightforward: A feature is “important” if shuffling its values increases the model error, because in this case, the model relied on the feature for the prediction. A feature is “unimportant” if shuffling its values leaves the model error unchanged, because in this case, the model ignored the feature for the prediction.

## 23.1 Theory

The permutation feature importance measurement was introduced by Breiman (2001) for random forests. Based on this idea, Fisher, Rudin, and Dominici (2019) proposed a model-agnostic version of the feature importance and called it model reliance. They also introduced more advanced ideas about feature importance, for example, a (model-specific) version that takes into account that many prediction models may predict the data well. Their paper is worth reading.

**The permutation feature importance algorithm based on Fisher, Rudin, and Dominici (2019):**

Input: Trained model  $\hat{f}$ , feature matrix  $\mathbf{X}$ , target vector  $\mathbf{y}$ , error measure  $L$ .

1. Estimate the original model error  $e_{orig} = \frac{1}{n_{test}} \sum_{i=1}^{n_{test}} L(y^{(i)}, \hat{f}(\mathbf{x}^{(i)}))$  (e.g., mean squared error).
2. For each feature  $j \in \{1, \dots, p\}$  do:
  - Generate feature matrix  $\mathbf{X}_{perm,j}$  by permuting feature  $j$  in the data  $\mathbf{X}$ . This breaks the association between feature  $j$  and true outcome  $y$ .
  - Estimate error  $e_{perm,j} = \frac{1}{n_{test}} \sum_{i=1}^{n_{test}} L(y^{(i)}, \hat{f}(\mathbf{x}_{perm,j}))$  based on the predictions of the permuted data.
  - Calculate permutation feature importance as quotient  $FI_j = e_{perm,j}/e_{orig}$  or difference  $FI_j = e_{perm,j} - e_{orig}$
3. Sort features by descending FI.

### Invert positive metrics

You can also use PFI with metrics where larger is better, like accuracy or AUC. Just make sure to swap the roles of  $e_{\text{perm}}$  and  $e_{\text{orig}}$  in the ratio/difference.

Fisher, Rudin, and Dominici (2018) suggest in their paper to split the dataset in half and swap the values of feature  $j$  of the two halves instead of permuting feature  $j$ . This is exactly the same as permuting feature  $j$  if you think about it. If you want a more accurate estimate, you can estimate the error of permuting feature  $j$  by pairing each instance with the value of feature  $j$  of each other instance (except with itself). This gives you a dataset of size  $n(n-1)$  to estimate the permutation error, and it takes a large amount of computation time. I can only recommend using the  $n(n-1)$ -method if you are serious about getting extremely accurate estimates.

### Use unseen data for PFI

Estimate PFI on data not used for model training to avoid overly optimistic results, especially with overfitting models. PFI on training data can falsely highlight irrelevant features as important due to model overfitting.

To understand which features the model **actually** used, consider alternatives like [SHAP importance](#) or [PDP importance](#), which don't rely on error measures.

In the examples, I used test data to compute permutation feature importance.

## 23.2 Example and interpretation

For the first example, we explain the support vector machine model trained to predict [the number of rented bikes](#), given weather conditions and calendar information. As error measurement, we use the mean absolute error. Figure 23.1 shows the permutation feature importance results. The most important feature was `cnt_2d_bfr`, and the least important was `holiday`.

Next, let's look at penguins. I trained 3 logistic regression models to predict `penguin sex`, using 2/3 of the data for training, and 1/3 for computing the importance. I measure the error as the log loss. Features associated with a model error increase by a factor of 1 (= no change) were not important for predicting penguin male vs. female, as Figure 23.2 shows.

The importance of each of the features for predicting penguin sex with the logistic regression models. The most important feature was `body_mass_g`. Permuting `body_mass_g` resulted in an increase in classification error by a factor of 15.4. But wait, how can `species` be an important feature as well, when I actually trained 3 models separately? I treated the 3 models as one black box model here. To this overall function, `species` appears as just another feature. Internally this feature splits the data, dispatches it to the three logistic regression models.

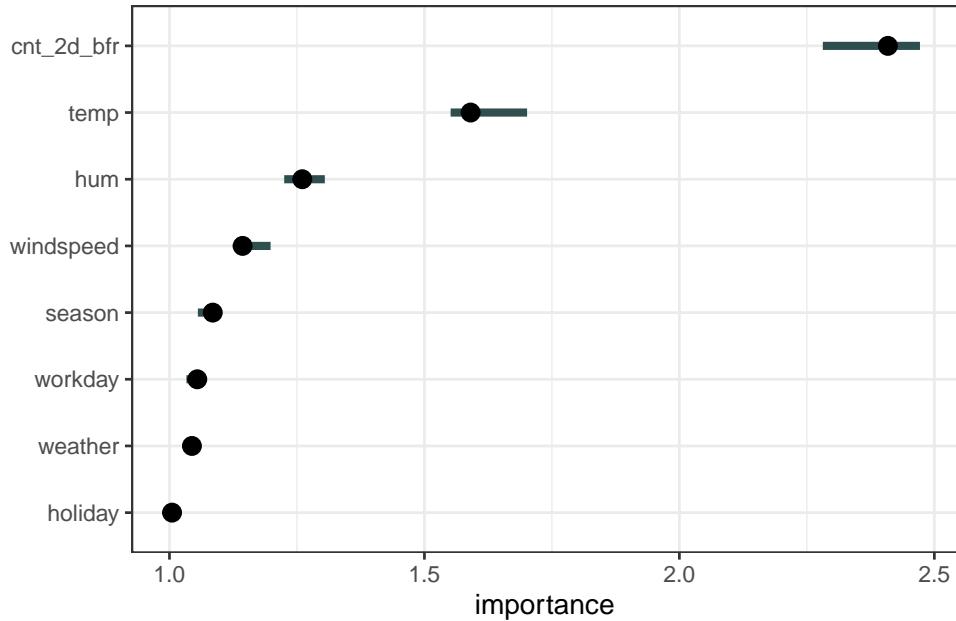


Figure 23.1: The feature importance for each of the features in predicting bike counts with a support vector machine. The dot shows the average importance across multiple permutations, and the line the 5% and 95% quantile range derived through repeated permutation.

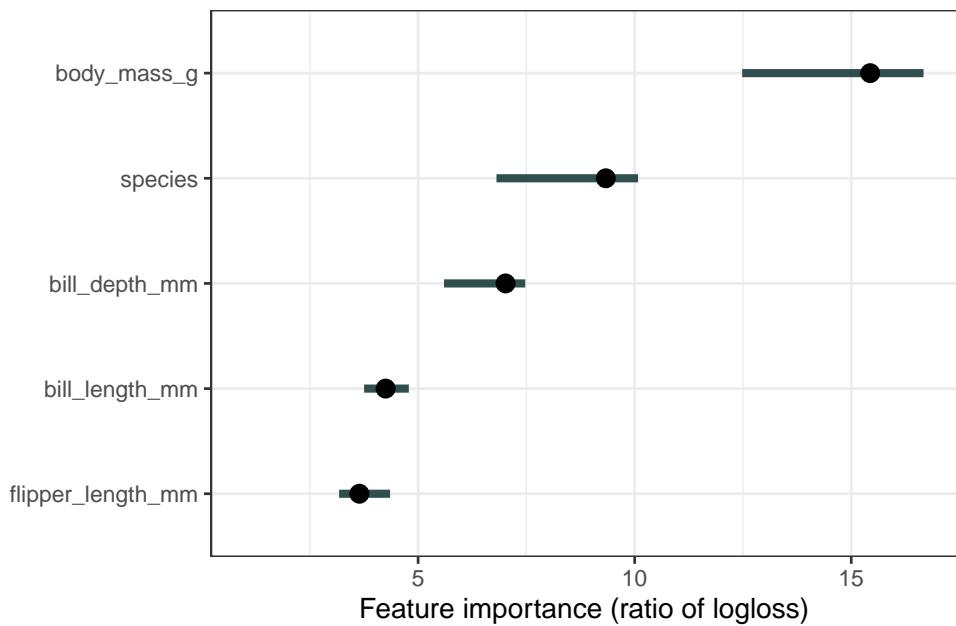


Figure 23.2: Permutation feature importance values for the penguin classification task.

### 23.3 Conditional feature importance

Like all the model-agnostic methods, permutation feature importance has a problem when features are dependent. Shuffling produces unrealistic or at least unlikely data points, that are then used to compute feature importance – not ideal. The problem is the *marginal* version of PFI ignores dependencies. But there is also the concept of *conditional* importance. The conditional version samples from the conditional distribution  $\mathbb{P}(X_j|X_{-j})$  instead of the marginal distribution  $\mathbb{P}(X_j)$  (shuffling is a way to sample from the marginal distribution). By conditional sampling, we sample more realistic data points.

However, sampling from the conditional distribution is difficult. It's an even more difficult task than our original machine learning task. But it can be simplified by making assumptions such as assuming a feature is only linearly correlated with other features. Here are some options for conditional sampling:

- Compute PFI in subgroups of the data and aggregate them. Subgroups are based on splitting in correlated features (Molnar, König, et al. 2023).
- Use matching and imputation techniques to generate samples from the conditional distribution (Fisher, Rudin, and Dominici 2019).
- Use knockoffs (Watson and Wright 2021).
- For random forests, there is a model-specific implementation Debeer and Strobl (2020), based on the original random forest importance (Breiman 2001).

Conditional feature importance has a different interpretation from marginal PFI:

- PFI measures the loss increase due to losing the feature information.
- Conditional importance measures the loss increase due to losing the information *unique* to that feature, information not encoded in other features.

Conditional importance can be a bit more difficult to interpret, since you also need an understanding of how features are dependent on each other. That's why I'm a big fan of the subgroups approach (and wrote a paper about it): Computing the PFI by group allows you to keep the marginal interpretation.

 Correlated features have lower conditional importance

Strongly dependent features usually have very low conditional importance, even when they are used by the model.

### 23.4 Group-wise PFI example

Let's go back to the penguins. Since PFI of, e.g., body mass, is computed by permuting across all data, we mix body masses from different penguin species. But we can simply adapt PFI by

splitting our data by species, and permuting for each subset separately, so that we get one PFI per feature AND species. Figure 23.3 shows that we can reduce the correlation by subsetting by species.

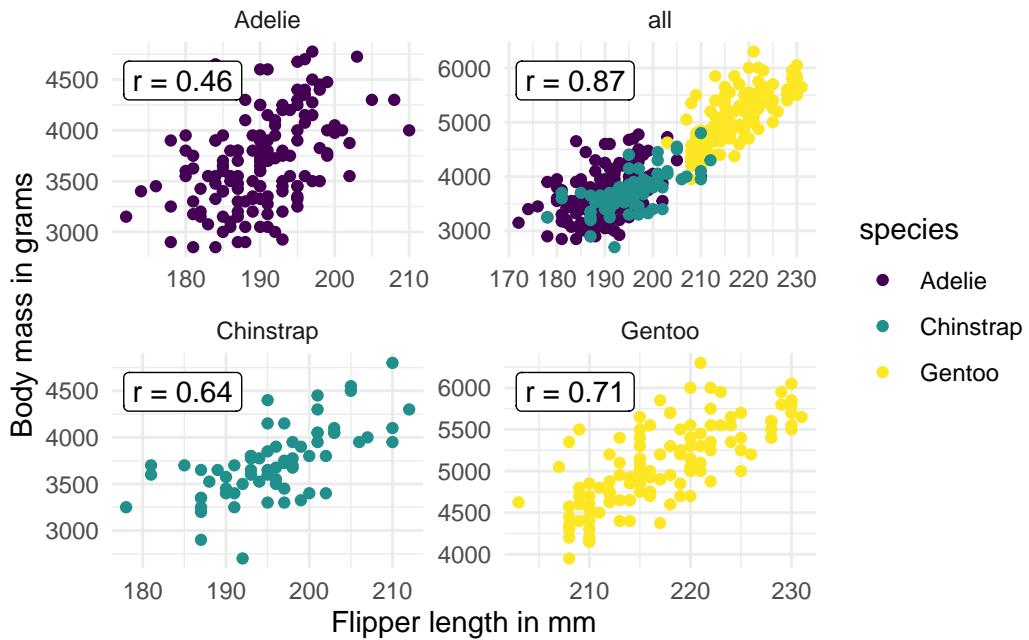


Figure 23.3: Correlation between body mass and flipper length by species subsets (and all together). Subsetting by species reduces the Pearson correlation ( $r$ ).

So we compute the permutation feature importance again, by species. Meaning we permute features by species, so that, for example, it can't happen that the body mass of a heavy Gentoo is assigned to a lightweight Adelie. While grouping reduces the correlation problem, it isn't solved, as we can see in Figure 23.3. For example, it can still happen that a Gentoo penguin with a body mass of 4000 grams gets "assigned" a flipper length of 230 mm, which produces an unrealistic data point. So still the interpretation comes with caveats. The results are displayed in Figure 23.4. A different picture emerges: Body mass is highly important for sex classification for Gentoo, less so for Adelie, and way less important for Chinstrap. Note that for the three logistic regression models it's natural to see it as three prediction models for which we can compute separate PFIs. However, since PFI is model-agnostic, we could do the same for a random forest, which simply uses species as a feature. Or we could split into subgroups by any other variable, in theory even by variables that weren't used by the model.

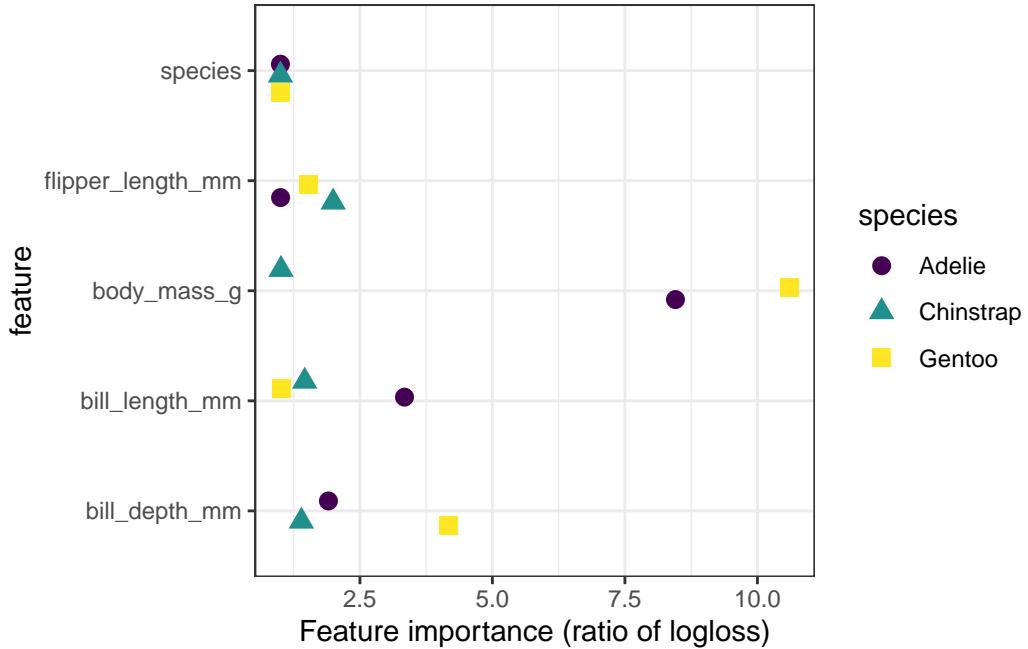


Figure 23.4: PFI by species for each logistic regression. The results are quite similar to overall PFI and don't differ much by species.

## 23.5 Strengths

**Nice interpretation:** Feature importance is the increase in model error when the feature's information is destroyed.

Feature importance provides a **highly compressed, global insight** into the model's behavior.

**PFI is useful for data insights.** If your goal is learning about the data, and the model is just a means to learn about the data, then PFI is great, since it relies on predictive performance (via the loss). If a feature is irrelevant for predicting the data, yet your model still uses it, then PFI will still show, in expectation, around zero importance for that feature. Other importance measures that are not based on prediction errors, like SHAP importance, will show an effect for features that are used for overfitting.

A positive aspect of using the error ratio instead of the error difference is that the feature importance measurements are **comparable across different problems**.

The importance measure automatically **takes into account all interactions** with other features. By permuting the feature, you also destroy the interaction effects with other features. This means that the permutation feature importance takes into account both the main feature

effect and the interaction effects on model performance. This is also a disadvantage because the importance of the interaction between two features is included in the importance measurements of both features. This means that the feature importances do not add up to the total drop in performance, but the sum is larger. Only if there is no interaction between the features, as in a linear model, do the importances add up approximately.

Permutation feature importance **does not require retraining the model**. Some other methods suggest deleting a feature, retraining the model, and then comparing the model error. Since the retraining of a machine learning model can take a long time, “only” permuting a feature can save a lot of time.

## 23.6 Limitations

Permutation feature importance is **linked to the error of the model**. This is not inherently bad, but in some cases not what you need. In some cases, you might prefer to know how much the model’s output varies for a feature without considering what it means for performance. For example, you want to find out how robust your model’s output is when someone manipulates the features. In this case, you would not be interested in how much the model performance decreases when a feature is permuted, but how much of the model’s output variance is explained by each feature. Model variance (explained by the features) and feature importance correlate strongly when the model generalizes well (i.e., it does not overfit).

Feature importance **doesn’t tell you how the feature influences the prediction**, only how much it affects the loss. Even if you know the importance of a feature, you don’t know: Does increasing the feature increase the prediction? Are there interactions with other features? Feature importance is just a ranking.

You **need access to the true outcome**. If someone only provides you with the model and unlabeled data – but not the true outcome – you cannot compute the permutation feature importance.

The permutation feature importance depends on shuffling the feature, which adds randomness to the measurement. When the permutation is repeated, the **results might vary greatly**. Repeating the permutation and averaging the importance measures over repetitions stabilizes the measure, but increases the time of computation.

If features are correlated, the permutation feature importance **can be biased by unrealistic data instances**. The problem is the same as with [partial dependence plots](#): The permutation of features produces unlikely data instances when two or more features are correlated. When they are positively correlated (like height and weight of a person) and I shuffle one of the features, I create new instances that are unlikely or even physically impossible (2 meter person weighing 30 kg for example), yet I use these new instances to measure the importance. In other words, for the permutation feature importance of a correlated feature, we consider how much the model performance decreases when we exchange the feature with values we would

never observe in reality. Check if the features are strongly correlated and be careful about the interpretation of the feature importance if they are. However, pairwise correlations might not be sufficient to reveal the problem.

Another tricky thing: **Adding a correlated feature can decrease the importance of the associated feature** by splitting the importance between both features. Let me give you an example of what I mean by “splitting” feature importance: We want to predict the probability of rain and use the temperature at 8:00 AM of the day before as a feature along with other uncorrelated features. I train a random forest and it turns out that the temperature is the most important feature and all is well and I sleep well the next night. Now imagine another scenario in which I additionally include the temperature at 9:00 AM as a feature that is strongly correlated with the temperature at 8:00 AM. The temperature at 9:00 AM does not give me much additional information if I already know the temperature at 8:00 AM. But having more features is always good, right? I train a random forest with the two temperature features and the uncorrelated features. Some of the trees in the random forest pick up the temperature at 8:00 AM, others the temperature at 9:00 AM, again others both, and again others none. The two temperature features together have a bit more importance than the single temperature feature before, but instead of being at the top of the list of important features, each temperature is now somewhere in the middle. By introducing a correlated feature, I kicked the most important feature from the top of the importance ladder to mediocrity. On one hand, this is fine, because it simply reflects the behavior of the underlying machine learning model, here the random forest. The temperature at 8:00 AM has simply become less important because the model can now rely on the temperature at 9:00 AM measurement as well. On the other hand, it makes the interpretation of the feature importance considerably more difficult. Imagine you want to check the features for measurement errors. The check is expensive, and you decide to check only the top 3 of the most important features. In the first case, you would check the temperature; in the second case, you would not include any temperature feature just because they now share the importance. Even though the importance values might make sense at the level of model behavior, it is confusing if you have correlated features.

## 23.7 Software and alternatives

The `iml` R package was used for the examples. The R packages `DALEX` and `vip`, as well as the Python libraries `alibi`, `eli5`, `scikit-learn`, and `rfpimp`, also implement model-agnostic permutation feature importance.

An algorithm called `PIMP` adapts the permutation feature importance algorithm to provide p-values for the importances. Another loss-based alternative is `LOFO`, which omits the feature from the training data, retrains the model, and measures the increase in loss. Permuting a feature and measuring the increase in loss is not the only way to measure the importance of a feature. The different importance measures can be divided into model-specific and model-agnostic

methods. The Gini importance for random forests, or standardized regression coefficients for regression models, are examples of model-specific importance measures.

A model-agnostic alternative to permutation feature importance is variance-based measures. Variance-based feature importance measures such as Sobol's indices or [functional ANOVA](#) give higher importance to features that cause high variance in the prediction function. Also, [SHAP importance](#) has similarities to a variance-based importance measure. If changing a feature greatly changes the output, then it is important. This definition of importance differs from the loss-based definition as in the case of permutation feature importance. This is evident in cases where a model overfits. If a model overfits and uses a feature that is unrelated to the output, then the permutation feature importance would assign an importance of zero because this feature does not contribute to producing correct predictions. A variance-based importance measure, on the other hand, might assign the feature high importance as the prediction can change a lot when the feature is changed.

A good overview of various importance techniques is provided in the paper by Wei, Lu, and Song (2015).

## 24 Leave One Feature Out (LOFO) Importance

Leave One Feature Out (LOFO) Importance measures a feature's importance by retraining the model without the feature and comparing the predictive performances.<sup>1</sup>

The intuition behind LOFO Importance: If dropping a feature makes the predictive performance worse, then it was an important feature. If dropping the feature keeps performance unchanged, it was not important. LOFO importance may even be negative when removing the feature *improves* the model performance. Since the algorithm gets to train a new model, the drop in performance is conditional on the other features that remain in the model, as we will see in the examples.

To get the LOCO importance for all  $p$  features, we have to retrain the model  $p$  times, each time with a different feature removed from the training data, and measure the resulting model's performance on test data. This makes LOFO a quite simple algorithm. Let's formalize the algorithm:

**Input:** Trained model  $\hat{f}$ , training data  $(\mathbf{X}_{\text{train}}, \mathbf{y}_{\text{train}})$ , test data  $(\mathbf{X}_{\text{test}}, \mathbf{y}_{\text{test}})$ , and error measure  $L$ .

**Procedure:**

1. Measure the original model error:

$$e_{\text{orig}} = \frac{1}{n_{\text{test}}} \sum_{i=1}^{n_{\text{test}}} L(y_{\text{test}}^{(i)}, \hat{f}(\mathbf{x}_{\text{test}}^{(i)}))$$

2. For each feature  $j \in \{1, \dots, p\}$ :

- Remove feature  $j$  from the dataset, creating new datasets  $\mathbf{X}_{\text{train},-j}$  and  $\mathbf{X}_{\text{test},-j}$ .
- Train a new model  $\hat{f}_{-j}$  on  $(\mathbf{X}_{\text{train},-j}, \mathbf{y}_{\text{train}})$ .

---

<sup>1</sup>The first formal description of LOFO I'm aware of is by Lei et al. (2018). They call it Leave One Covariate Out (LOCO), and the paper is mostly about distribution-free testing.

- Measure the new error on the modified test set:

$$e_{-j} = \frac{1}{n_{\text{test}}} \sum_{i=1}^{n_{\text{test}}} L(y_{\text{test}}^{(i)}, \hat{f}_{-j}(\mathbf{x}_{\text{test}, -j}^{(i)}))$$

### 3. Calculate LOFO importance for each feature.

- As a quotient:

$$\text{LOFO}_j = \frac{e_{-j}}{e_{\text{orig}}}$$

- Or as a difference:

$$\text{LOFO}_j = e_{-j} - e_{\text{orig}}$$

### 4. Sort and visualize the features by descending importance $\text{LOFO}_j$ .

When using performance measures for  $L$  instead of error measures, where larger is better, like accuracy, make sure to multiply the difference by -1, or switch the order of the quotient. To train and retrain the model use the training data, and for measuring the performance use test data.

## 24.1 Examples

We predict bike rentals based on weather and calendar information using a random forest trained on 2/3 of the data. Figure 24.1 shows that temperature and previous number of bike rentals are the most important features according to LOFO. The holiday feature has a negative importance, which has implications for feature selection: Because of how LOFO works algorithmically, we now know that removing the holiday feature increases the model's performance.

Let's try something. To simulate an extreme version of correlated features, I created a new bike dataset which has two temperature columns: `temp` and `temp_copy`. And, you guessed it, `temp_copy` has the exact same values as `temp`, which means 100% correlation. Again, I trained a random forest on this new dataset. Let's see what happens to the LOFO importances.

Figure 24.2 shows an interesting phenomenon: The LOFO importances of both temperature features are now almost zero.<sup>2</sup> But if we e.g. remove `temp_copy`, we end up with the original model, for which we knew that `temp` was the most important feature. Clearly, it would be

---

<sup>2</sup>The features `temp` and `temp_copy` have similar but not equal LOFO importances, which is due to randomness in the model training.

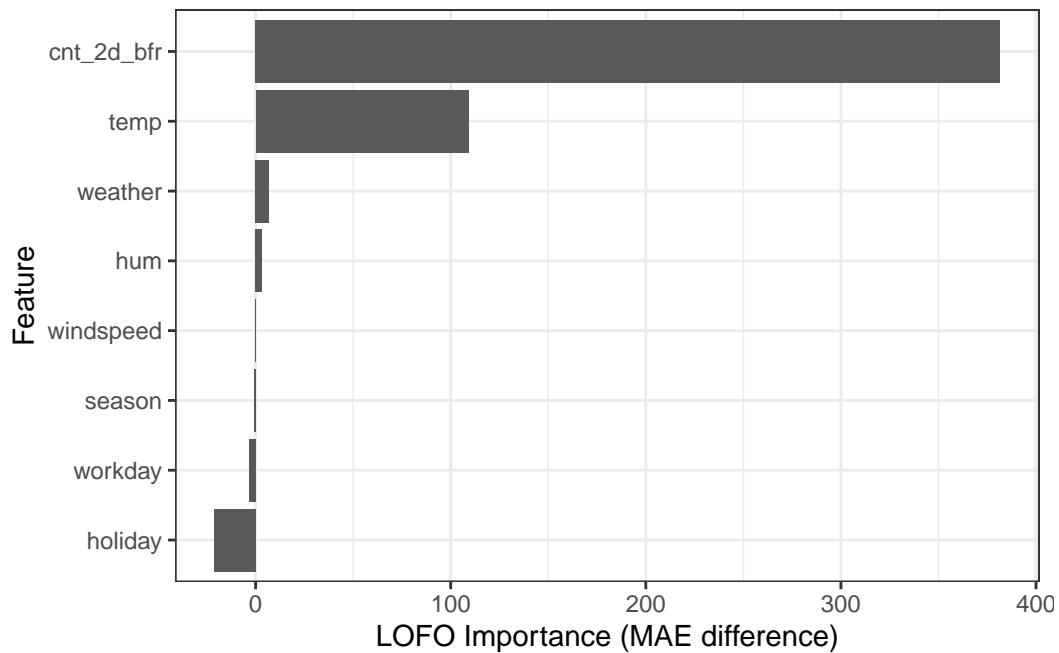


Figure 24.1: LOFO feature importances for the bike rental data, where temperature is duplicated, introducing a perfectly correlated feature (`temp_copy`).

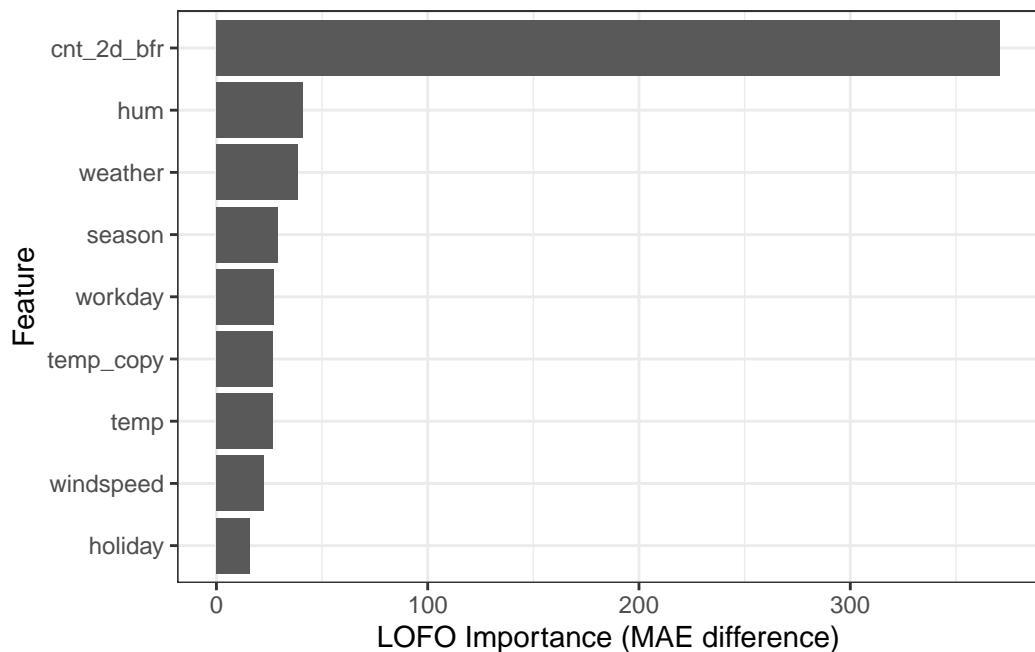


Figure 24.2: LOFO feature importances for the bike rental data.

wrong to conclude that this new model doesn't rely on temperature at all. Both `temp` and `temp_copy` get a low importance because, by the definition of LOFO importance, they are not important. When we remove the feature `temp`, we don't lose any information at all. That's because LOFO has a conditional interpretation: Given the other features, how much will removing a feature worsen the model's predictive performance? Conclusions:

- LOFO Importance of strongly correlated features is always low, potentially even negative<sup>3</sup>, since LOFO importance is to be interpreted conditionally on the information provided by the other features. And if you already have a temperature feature in the data, the perfect copy of that same feature is not important additional information.
- When you use LOFO Importance as information for feature selection, beware of the interpretation: LOFO only indicates how the model performance reacts to *individually* removing features. As Figure 24.2 showed, LOFO doesn't give us the information on how the performance changes when removing 2 or more features at once.

With that knowledge, let's try LOFO for a random forest predicting penguin sex from body measurements. For this, I've trained a random forest on 2/3 of the data, leaving 1/3 of the data for error estimation. Figure 24.3 shows that the bill depth was the most important feature according to LOFO.

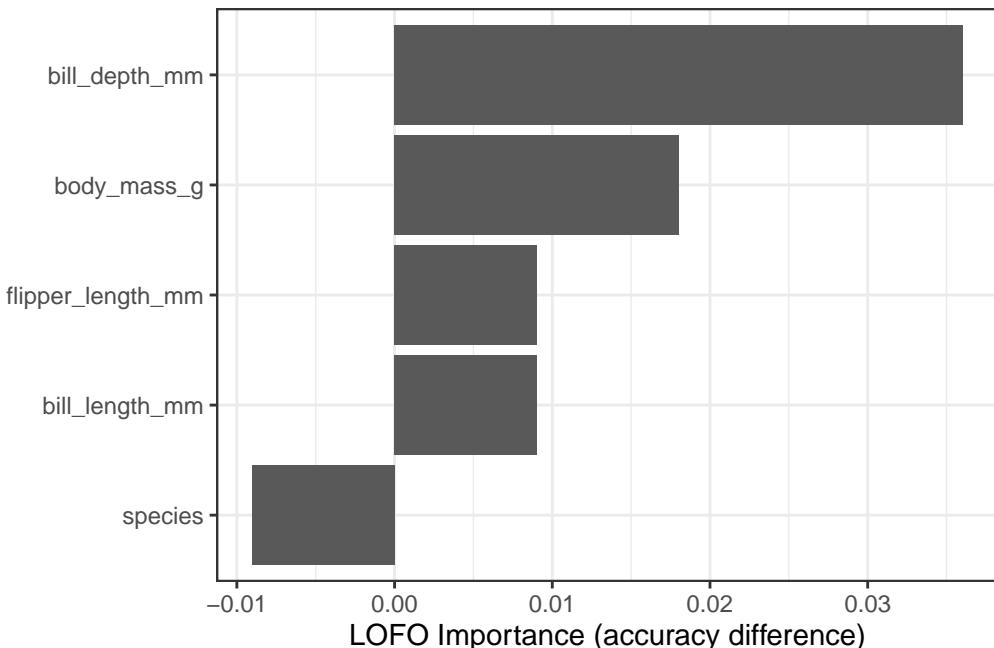


Figure 24.3: LOFO importances for the penguin data.

---

<sup>3</sup>We need to assume here that the learner is able to compensate for the removal of a feature by relying more on the other features. But I don't know of any machine learning algorithm that isn't capable of that.

Note that because of using accuracy here, where larger is better, I multiplied the importance by minus one. LOFO also tells us that we could safely remove the features `species` and `flipper_length`. However, we would want to keep `species`, because the model has to be able to distinguish between the species.

## 24.2 LOFO versus PFI

LOFO differs from the other methods presented in this book, since most of the other methods don't require retraining the model. Arguably, LOFO is a post-hoc, model-agnostic method, since it can be applied to any model and retraining the model doesn't affect the original model that we want to interpret. However, due to retraining the model, the interpretation shifts from only interpreting that one single model to interpreting the learner and how model training reacts to changes in the features.

To me, the biggest question is: How does LOFO compare to PFI, and when should you use which importance? Let's start with the similarities: Both PFI and LOFO are performance-based importance measures; both compute importance by "removing" the information of a feature, even when they do it differently; both work in a one-at-a-time way, at least in their simplest version. But LOFO and PFI differ in other aspects. As we already saw in the example Figure 24.2, LOFO has a conditional interpretation of importance. It's only about a feature's *additional* predictive value. This distinguishes LOFO from *marginal* permutation feature importance. LOFO is more similar to conditional PFI, and they both have interpretations conditional on the other features.

But since conditional PFI and LOFO work differently, they differ in their interpretations. Conditional PFI is an interpretation that only involves the model at hand. LOFO importance focuses more on the machine learning algorithm, since it involves retraining the model, and the interpretation now involves multiple models trained differently. This also means that LOFO importance is affected by how the machine learning algorithm handles datasets with one fewer column. A decision tree algorithm might produce a completely different tree. Therefore LOFO's interpretation is both about the model at hand, but also about the machine learning algorithm and how it reacts to removing a feature.

## 24.3 Strengths

**Implementing LOFO is simple.** For the examples here I did just that. You can do it yourself, you don't actually need any package for that.

**LOFO is useful for feature selection:** A feature with a LOFO importance below zero can be removed to improve model performance; a feature with an importance of zero can also be safely removed without affecting the performance. However, be aware that LOFO is subject to

the randomness of the training process and also depends on the data sample. Just make sure that you remove only one feature at a time based on LOFO importance results. If you want to remove two features, you have to do it sequentially, meaning compute LOFO importance, remove the unimportant feature, compute LOFO again, and remove again the least important. You can also adapt LOFO to LTFO (leave two features out) to compute shared importance. Other importance methods, like PFI or SHAP importance, don't directly provide actionable feature selection information.

LOFO **doesn't create unrealistic data** like other methods, such as marginal PFI, since LOFO removes features and doesn't probe the model with newly sampled data.

LOFO is **useful for the modeling phase** due to its actionable insights for feature selection. It's also useful when the focus is more on **understanding the underlying phenomenon** and less on the model itself. Note, however, that also how the specific learner reacts to removing features will influence the LOFO importance values.

LOFO makes sense when your goal of interpretation is about the phenomenon (and less about the model itself) or the learner.

## 24.4 Limitations

**LOFO is costly:** If you compute LOFO importance for all features, you have to retrain the model  $p$  times. This can be costly, especially when compared to Permutation Feature Importance.

LOFO is **not the best method for understanding a specific model**. For example, for more of a model audit setting, LOFO is not suitable since the interpretation is based on training new models, meaning the interpretation is not only based on the model under inspection, but also on other models produced by the machine learning algorithm, so that results are to be interpreted in terms of the machine learning algorithm.

It's also a bit **unclear how to handle hyperparameter tuning**. Should you train with the exact same hyperparameters, or run the hyperparameter optimization again? Keeping the hyperparameters fixed is computationally cheaper and shifts LOFO's focus more on that model as the newly trained models are potentially more similar to it. Running hyperparameter optimization for each of the  $p$  models is costly, but makes LOFO results more informative for feature selection and insights into the relations between features and the target since you reduce uncertainty due to model errors.

**Highly correlated features get low LOFO importance.** This is to be expected simply by how LOFO works, but it's a pitfall and easily forgotten when looking at the importance plots. It also means that studying the correlations of your data is a must before you can interpret LOFO importances. You can also group highly correlated features and only remove them together so your interpretation becomes more marginal than conditional.

## 24.5 Software and alternatives

LOFO has a [Python implementation](#). But it's also an algorithm you can implement yourself easily.

An alternative to LOFO is [conditional PFI](#). It's also possible to add a permuted version of the variable instead of dropping it, and then relearning a model. This has the advantage of comparing two models with the same number of features, but it introduces another source of uncertainty (the permutation itself is random).

The feature selection method “backward sequential feature selection” is basically LOFO + feature removal applied sequentially.

# 25 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!

## 25.1 Theory

Surrogate models are also used in engineering: If an outcome of interest is expensive, time-consuming, or otherwise difficult to measure (e.g., because it comes from a complex computer simulation), a cheap and fast surrogate model of the outcome can be used instead. The difference between the surrogate models used in engineering and in interpretable machine learning is that the underlying model is a machine learning model (not a simulation) and that the surrogate model must be interpretable. The purpose of (interpretable) surrogate models is to approximate the predictions of the underlying model as accurately as possible and to be interpretable at the same time. The idea of surrogate models can be found under different names: Approximation model, metamodel, response surface model, emulator, etc.

About the theory: There's actually not much theory needed to understand surrogate models. We want to approximate our black box prediction function  $f$  as closely as possible with the surrogate model prediction function  $g$ , under the constraint that  $g$  is interpretable. For the function  $g$ , any interpretable model can be used.

For example, a linear model:

$$g(\mathbf{x}) = \beta_0 + \beta_1 x_1 + \dots + \beta_p x_p$$

Or a decision tree:

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

Training a surrogate model is a model-agnostic method, since it does not require any information about the inner workings of the black box model, only access to data and the prediction

function is necessary. If the underlying machine learning model was replaced with another, you could still use the surrogate method. The choice of the black box model type and of the surrogate model type is decoupled.

Perform the following steps to obtain a surrogate model:

1. Select a dataset  $\mathbf{X}$ . This can be the same dataset that was used for training the black box model or a new dataset from the same distribution. You could even select a subset of the data or a grid of points, depending on your application.
2. For the selected dataset  $\mathbf{X}$ , get the predictions of the black box model.
3. Select an interpretable model type (linear model, decision tree, ...).
4. Train the interpretable model on the dataset  $\mathbf{X}$  and its predictions.
5. Congratulations! You now have a surrogate model.
6. Measure how well the surrogate model replicates the predictions of the black box model.
7. Interpret the surrogate model.

You may find approaches for surrogate models that have some extra steps or differ a little, but the general idea is usually as described here.

One way to measure how well the surrogate replicates the black box model is the R-squared measure:

$$R^2 = 1 - \frac{SSE}{SST} = 1 - \frac{\sum_{i=1}^n (\hat{y}_*^{(i)} - \hat{y}^{(i)})^2}{\sum_{i=1}^n (\hat{y}^{(i)} - \bar{\hat{y}})^2}$$

where  $\hat{y}_*^{(i)}$  is the prediction for the  $i$ -th instance of the surrogate model,  $\hat{y}^{(i)}$  the prediction of the black box model and  $\bar{\hat{y}}$  the mean of the black box model predictions. SSE stands for sum of squares error and SST for sum of squares total. The R-squared measure can be interpreted as the percentage of variance that is captured by the surrogate model. If R-squared is close to 1 (= low SSE), then the interpretable model approximates the behavior of the black box model very well. If the interpretable model is very close, you might want to replace the complex model with the interpretable model. If the R-squared is close to 0 (= high SSE), then the interpretable model fails to explain the black box model.

Note that we have not talked about the model performance of the underlying black box model, i.e., how good or bad it performs in predicting the actual outcome. The performance of the black box model does not play a role in training the surrogate model. The interpretation of the surrogate model is still valid because it makes statements about the model and not about the real world. But of course, the interpretation of the surrogate model becomes irrelevant if the black box model is bad because then the black box model itself is irrelevant.

We could also build a surrogate model based on a subset of the original data or 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 no longer really global). If we weight the data

locally by a specific instance of the data (the closer the instances to the selected instance, the higher their weight), we get a local surrogate model that can explain the individual prediction of the instance.

## 25.2 Example

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

First, we train a support vector machine to predict the [daily number of rented bikes](#) given weather and calendar information. The support vector machine is not very interpretable, so we train a surrogate with a CART decision tree as an interpretable model using the original training data to approximate the behavior of the support vector machine. The surrogate model shown in Figure 25.1 has an R-squared (variance explained) of 0.76 on the test data, which means it approximates the underlying black box behavior quite well, but not perfectly. If the fit were perfect, we could throw away the support vector machine and use the tree instead. The distributions in the nodes show that the surrogate tree predicts a higher number of rented bikes when the temperature is above 13 degrees Celsius and when count two days before was higher.

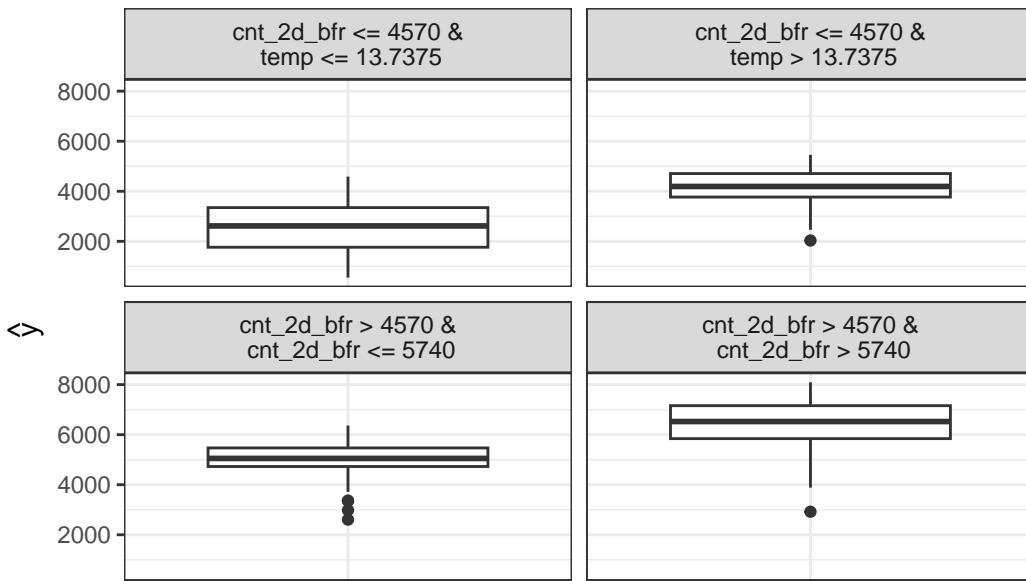


Figure 25.1: The terminal nodes of a surrogate tree that approximates the predictions of a support vector machine trained on the bike rental dataset.

In our second example, we classify whether a [penguin](#) is female or male with a random forest, again training a decision tree on the original dataset, but with the prediction of the random forest as the outcome, instead of the real classes (male/female) from the data. The surrogate model shown in Figure 25.2 has an R-squared (variance explained) of 0.71, which means it does approximate the random forest somewhat, but not perfectly.

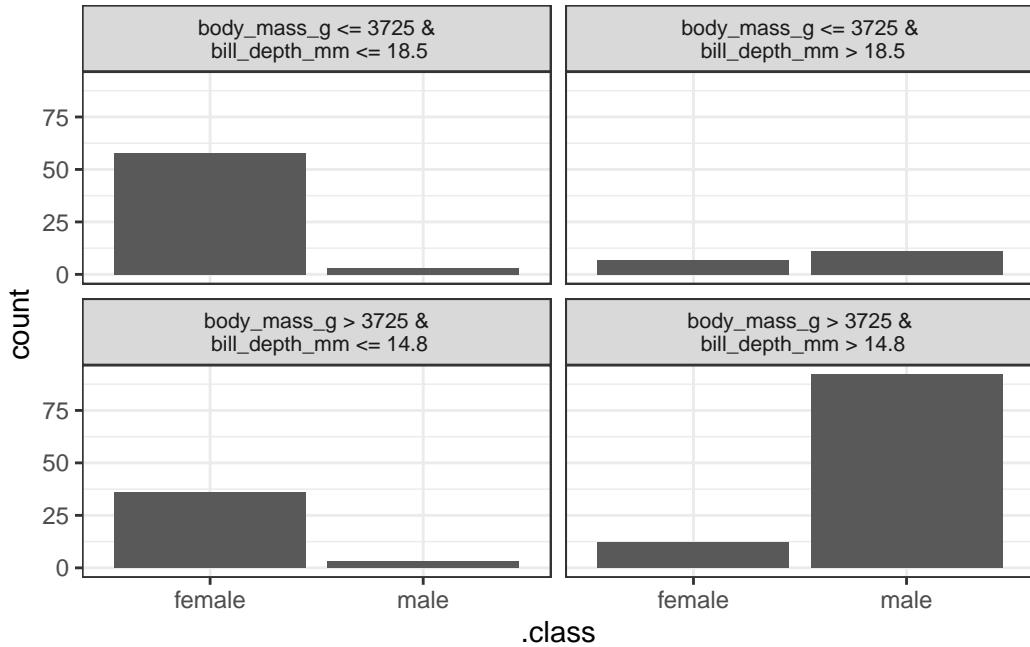


Figure 25.2: The terminal nodes of a surrogate tree that approximates the predictions of a random forest trained on the penguins dataset. The counts in the nodes show the frequency of the black box models classifications in the nodes.

## 25.3 Strengths

The surrogate model method is **flexible**: Any interpretable model can be used. This also means that you can exchange not only the interpretable model, but also the underlying black box model. Suppose you create some complex model and explain it to different teams in your company. One team is familiar with linear models, and the other team can understand decision trees. You can train two surrogate models (linear model and decision tree) for the original black box model and offer two kinds of explanations. If you find a better-performing black box model, you do not have to change your method of interpretation because you can use the same class of surrogate models.

I would argue that the approach is very **intuitive** and straightforward. This means it is easy

to implement, but also easy to explain to people not familiar with data science or machine learning.

With the **R-squared measure**, we can easily measure how good our surrogate models are in approximating the black box predictions.

## 25.4 Limitations

You have to be aware that you draw **conclusions about the model and not about the data**, since the surrogate model never sees the real outcome.

It'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 one subset of the dataset, but widely divergent for another subset**. In this case, the interpretation for the simple model would not be equally good for all data points.

The interpretable model you choose as a surrogate **comes with all its advantages and disadvantages**.

Some people argue that there are, in general, **no intrinsically interpretable models** (including even linear models and decision trees), and that it would even be dangerous to have an illusion of interpretability. If you share this opinion, then of course this method is not for you.

## 25.5 Software

I used the `iml` R package for the examples. If you can train a machine learning model, then you should be able to implement surrogate models yourself. Simply train an interpretable model to predict the predictions of the black box model.

# 26 Prototypes and Criticisms

A **prototype** is a data instance that is representative of all the data. A **criticism** is a data instance that is not well represented by the set of prototypes. The purpose of criticisms is to provide insights together with prototypes, especially for data points which the prototypes do not represent well. Prototypes and criticisms can be used independently from a machine learning model to describe the data, but they can also be used to create an interpretable model or to make a black box model interpretable.

In this chapter, I use the expression “data point” to refer to a single instance, to emphasize the interpretation that an instance is also a point in a coordinate system where each feature is a dimension. Figure 26.1 shows a simulated data distribution, with some of the instances chosen as prototypes and some as criticisms. The small points are the data, the large points are the criticisms, and the large squares are the prototypes. The prototypes are selected (manually), in this case, 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.

I selected the prototypes manually, which does not scale well and probably leads to poor results. There are many approaches to find prototypes in the data. One of these is k-medoids, a clustering algorithm related to the k-means algorithm. Any clustering algorithm that returns actual data points as cluster centers would qualify for selecting prototypes. But most of these methods find only prototypes, but no criticisms. This chapter presents MMD-critic by Kim, Khanna, and Koyejo (2016), an approach that combines prototypes and criticisms in a single framework.

MMD-critic compares the distribution of the data and the distribution of the selected prototypes. This is the central concept for understanding the MMD-critic method. MMD-critic selects prototypes that minimize the discrepancy between the two distributions. Data points in areas with high density are good prototypes, especially when points are selected from different “data clusters”. Data points from regions that are not well explained by the prototypes are selected as criticisms.

## 26.1 Theory

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

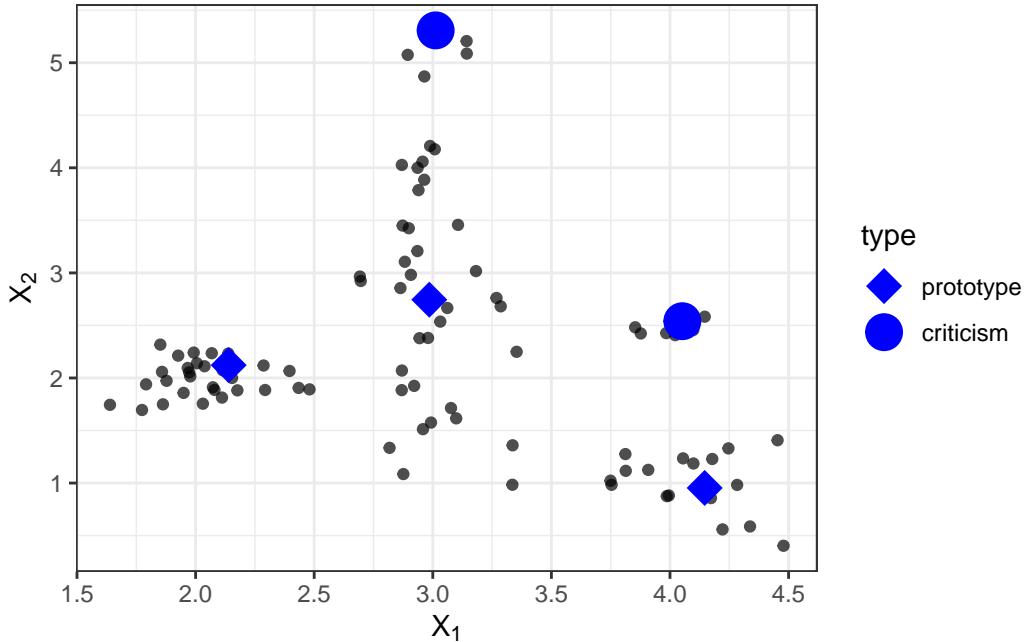


Figure 26.1: Prototypes and criticisms for a data distribution with two features  $x_1$  and  $x_2$ .

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

We need a couple of ingredients to find prototypes and criticisms for a dataset with MMD-critic. As the most basic ingredient, we need a **kernel function** to estimate the data densities. A kernel is a function that weighs two data points according to their proximity. Based on density estimates, we need a measure that tells us how different two distributions are so that we can determine whether the distribution of the prototypes we select is close to the data distribution. This is solved by measuring the **maximum mean discrepancy (MMD)**. Also, based on the kernel function, we need the **witness function** to tell us how different two distributions are at a particular data point. With the witness function, we can select criticisms, i.e., data points at which the distribution of prototypes and data diverges, and the witness function takes on large absolute values. The last ingredient is a search strategy for good prototypes and criticisms, which is solved with a simple **greedy search**.

Let'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. We want to evaluate whether the prototypes distribution differs from the data distribution. We estimate both with kernel density functions. The maximum mean discrepancy

measures the difference between two distributions, which is the supremum over a function space of differences between the expectations according to the two distributions. All clear? Personally, I understand these concepts much better when I see how something is calculated with data. The following formula shows how to calculate the squared MMD measure (MMD<sup>2</sup>):

$$\text{MMD}^2 = \frac{1}{m^2} \sum_{i,j=1}^m k(\mathbf{z}_i, \mathbf{z}_j) - \frac{2}{mn} \sum_{i=1}^m \sum_{j=1}^n k(\mathbf{z}_i, \mathbf{x}_j) + \frac{1}{n^2} \sum_{i,j=1}^n k(\mathbf{x}_i, \mathbf{x}_j)$$

$k$  is a kernel function that measures the similarity of two points, but more about this later.  $m$  is the number of prototypes  $\mathbf{z}$ , and  $n$  is the number of data points  $\mathbf{x}$  in our original dataset. The prototypes  $\mathbf{z}$  are a selection of data points  $\mathbf{x}$ . Each point is multidimensional, that is, it can have multiple features. The goal of MMD-critic is to minimize  $\text{MMD}^2$ . The closer  $\text{MMD}^2$  is to zero, the better the distribution of the prototypes fits the data. The key to bringing  $\text{MMD}^2$  down to zero is the term in the middle, which calculates the average proximity between the prototypes and all other data points (multiplied by 2). If this term adds up to the first term (the average proximity of the prototypes to each other) plus the last term (the average proximity of the data points to each other), then the prototypes explain the data perfectly. Try out what would happen to the formula if you used all  $n$  data points as prototypes.

Figure 26.2 illustrates the  $\text{MMD}^2$  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  $\text{MMD}^2$  measure in the plot titles. The prototypes are the large dots, and their distribution is shown as contour lines. The selection of the prototypes that best covers the data in these scenarios (bottom left) has the lowest discrepancy value.

A choice for the kernel is the radial basis function kernel:

$$k(\mathbf{x}, \mathbf{x}') = \exp(-\gamma \|\mathbf{x} - \mathbf{x}'\|^2)$$

where  $\|\mathbf{x} - \mathbf{x}'\|^2$  is the Euclidean distance between two points and  $\gamma$  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.

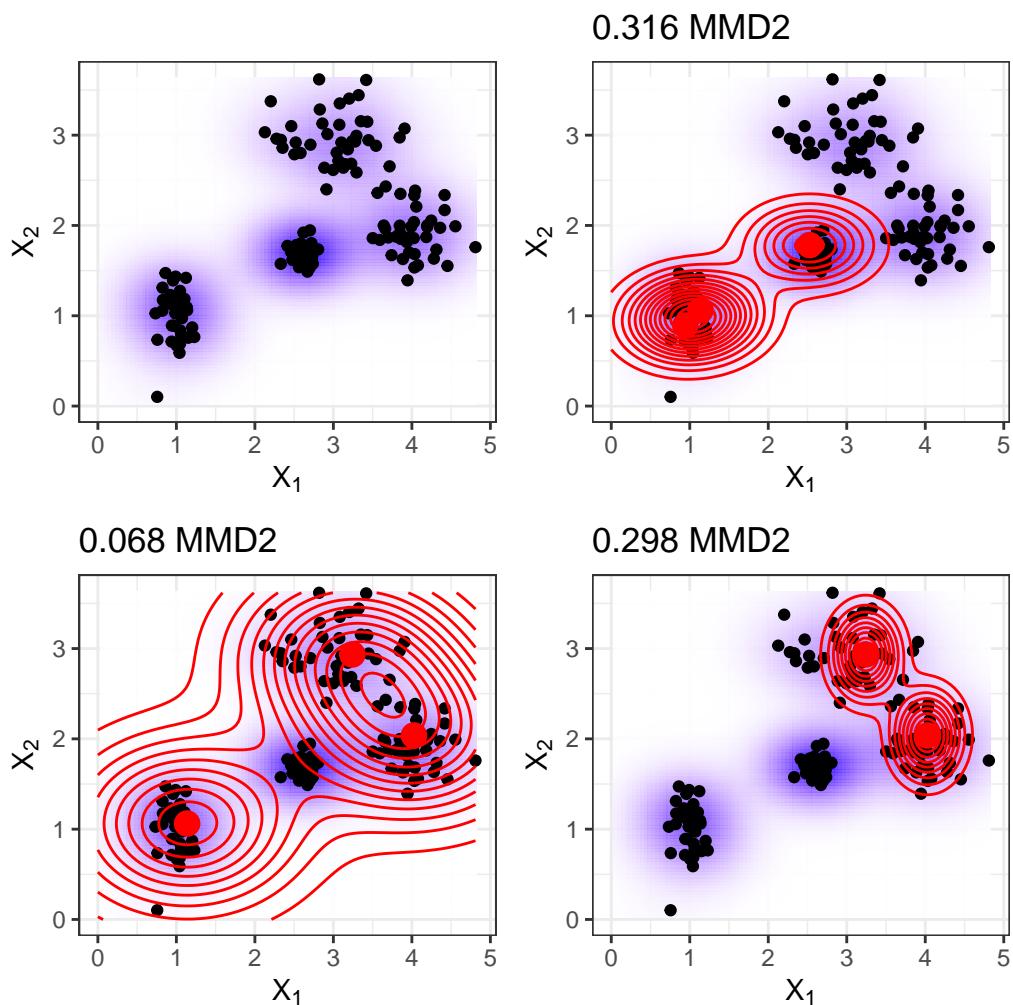


Figure 26.2: The squared maximum mean discrepancy measure (MMD2) for a dataset with two features and different selections of prototypes.

- 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:

$$\text{witness}(\mathbf{x}) = \frac{1}{n} \sum_{i=1}^n k(\mathbf{x}, \mathbf{x}^{(i)}) - \frac{1}{m} \sum_{j=1}^m k(\mathbf{x}, \mathbf{z}^{(j)})$$

For two datasets (with the same features), the witness function gives you the means of evaluating in which empirical distribution the point  $\mathbf{x}$  fits better. To find criticisms, we look for extreme values of the witness function in both negative and positive directions. The first term in the witness function is the average proximity between point  $\mathbf{x}$  and the data, and, respectively, the second term is the average proximity between point  $\mathbf{x}$  and the prototypes. If the witness function for a point  $\mathbf{x}$  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  $\mathbf{x}$ . A negative witness function at point  $\mathbf{x}$  means that the prototype distribution overestimates the data distribution (for example, if we select a prototype but there are only a few data points nearby); a positive witness function at point  $\mathbf{x}$  means that the prototype distribution underestimates the data distribution (for example, if there are many data points around  $\mathbf{x}$  but we have not 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 Figure 26.3 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.

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 MMD<sup>2</sup>, 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.

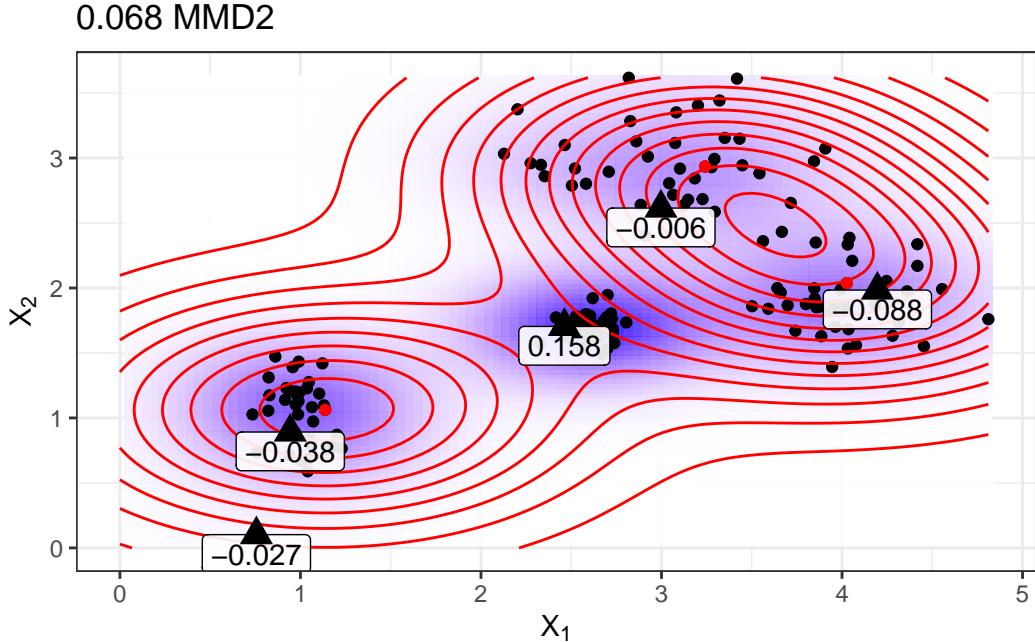


Figure 26.3: Evaluations of the witness function at different points.

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:

$$\hat{f}(\mathbf{x}) = \arg \max_{i \in S} k(\mathbf{x}, \mathbf{x}_i)$$

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’s not enough just to check the performance of the model because if it were 99% correct, this issue could still be in the 1%. And labels can also be wrong! Going through all the training data and performing a sanity check if the prediction is problematic might have revealed the problem but would be infeasible. But the selection of – say a few thousand – prototypes and criticisms is feasible and could have revealed a problem with the data: It might have shown that there is a lack of images of people with dark skin, which indicates a problem with the diversity in the dataset. Or it could have shown one or more images of a person with dark skin as a prototype or (probably) as a criticism with the notorious “gorilla” classification. I do not promise that MMD-critic would certainly intercept these kinds of mistakes, but it is a good sanity check.

## 26.2 Examples

The following example of MMD-critic uses a handwritten digit dataset. Looking at the actual prototypes in Figure 26.4, you might notice that the number of images per digit is different. This is because a fixed number of prototypes was searched across the entire dataset, and not with a fixed number per class. As expected, the prototypes show different ways of writing the digits.

## 26.3 Strengths

In a user study, the authors of MMD-critic gave images to the participants, which they had to visually match to one of two sets of images, each representing one of two classes (e.g., two dog breeds). The **participants performed best when the sets showed prototypes and criticisms instead of random images of a class**.

You are free to **choose the number of prototypes and criticisms**.

MMD-critic works with density estimates of the data. This **works with any type of data and any type of machine learning model**.

The algorithm is **easy to implement**.



Figure 26.4: Prototypes for a handwritten digits dataset.

MMD-critic is **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.

## 26.4 Limitations

While mathematically, prototypes and criticisms are defined differently, their **distinction is based on a cut-off value** (the number of prototypes). Suppose you choose a too low number of prototypes to cover the data distribution. The criticisms would end up in the areas that are not that well explained. But if you were to add more prototypes, they would also end up in the same areas. Any interpretation has to take into account that criticisms strongly depend on the existing prototypes and the (arbitrary) cut-off value for the number of prototypes.

You have to **choose the number of prototypes and criticisms**. As much as this can be nice-to-have, it is also a disadvantage. How many prototypes and criticisms do we actually need? The more, the better? The less, the better? One solution is to select the number of prototypes and criticisms by measuring how much time humans have for the task of looking at the images, which depends on the particular application. Only when using MMD-critic to build a classifier do we have a way to optimize it directly. One solution could be a scree plot

showing the number of prototypes on the x-axis and the MMD<sup>2</sup> measure on the y-axis. We would choose the number of prototypes where the MMD<sup>2</sup> 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's unclear. (Maybe I'm 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's some code available, but it is **not yet implemented as nicely packaged and documented software**.

## 26.5 Software and alternatives

An implementation of MMD-critic can be found in [the authors' GitHub repository](#). Another Python implementation is [mmd-critic](#), and this one is pip-installable.

Recently an extension of MMD-critic was developed: Protodash. The authors claim advantages over MMD-critic in their [publication](#). A Protodash implementation is available in the [IBM AIX360](#) tool.

The simplest alternative to finding prototypes is [k-medoids](#) by Rousseeuw and Kaufman (1987).

## **Part IV**

# **Neural Network Interpretation**

# 27 Learned Features

Convolutional neural networks learn abstract features and concepts from raw image pixels. [Feature Visualization](#) visualizes the learned features by activation maximization. [Network Dissection](#) labels neural network units (e.g., channels) with human concepts.

Deep neural networks learn high-level features in the hidden layers. This is one of their greatest strengths and reduces the need for feature engineering. Assume you want to build an image classifier with a support vector machine. The raw pixel matrices are not the best input for training your SVM, so you create new features based on color, frequency domain, edge detectors, and so on. With convolutional neural networks, the image is fed into the network in its raw form (pixels). The network transforms the image many times. First, the image goes through many convolutional layers. In those convolutional layers, the network learns new and increasingly complex features in its layers, see Figure 27.1. Then the transformed image information goes through the fully connected layers and turns into a classification or prediction.

- The first convolutional layer(s) learn features such as edges and simple textures.
- Later convolutional layers learn features such as more complex textures and patterns.
- The last convolutional layers learn features such as objects or parts of objects.
- The fully connected layers learn to connect the activations from the high-level features to the individual classes to be predicted.

Cool. But how do we actually get those trippy images?

## 27.1 Feature visualization

The approach of making the learned features explicit is called **Feature Visualization**. Feature visualization for a unit of a neural network is done by finding the input that maximizes the activation of that unit. “Unit” refers either to individual neurons, channels (also called feature maps), entire layers, or the final class probability in classification (or the corresponding pre-softmax neuron, which is recommended). Figure 27.2 visualizes the different possibilities. Individual neurons are atomic units of the network, so we would get the most information by creating feature visualizations for each neuron. But there is a problem: Neural networks often contain millions of neurons. Looking at each neuron’s feature visualization would take too long. The channels (sometimes called activation maps) as units are a good choice for feature

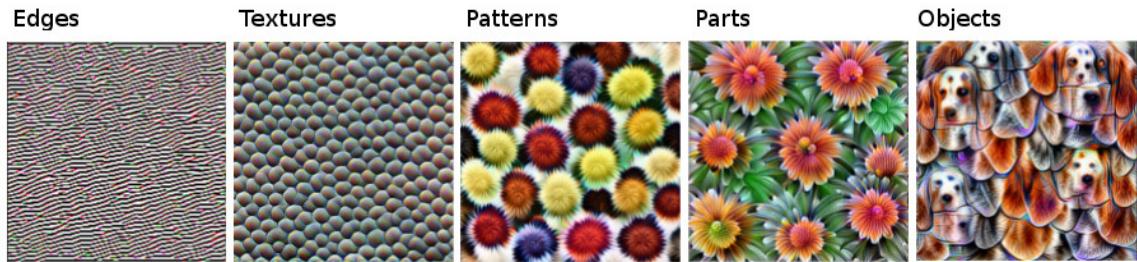


Figure 27.1: Features learned by a convolutional neural network (Inception V1) trained on the ImageNet data. The features range from simple features in the lower convolutional layers (left) to more abstract features in the higher convolutional layers (right). Figure from Olah et al. (2017, CC-BY 4.0) <https://distill.pub/2017/feature-visualization/appendix/>.

visualization. We can go one step further and visualize an entire convolutional layer. Layers as a unit are used for Google's DeepDream, which repeatedly adds the visualized features of a layer to the original image, resulting in a dream-like version of the input.

### 27.1.1 Feature visualization through optimization

In mathematical terms, feature visualization is an optimization problem. We assume that the weights of the neural network are fixed, which means that the network is trained. We are looking for a new image that maximizes the (mean) activation of a unit, here a single neuron:

$$\mathbf{x}^* = \arg \max_{\mathbf{x}} h_{n,u,v,z}(\mathbf{x})$$

The function  $h$  one particular the activation of a neuron,  $\mathbf{x}$  the input of the network (an image),  $u$  and  $v$  describe the spatial position of the neuron,  $n$  specifies the layer, and  $z$  is the channel index. For the mean activation of an entire channel  $z$  in layer  $n$ , we maximize:

$$\mathbf{x}^* = \arg \max_{\mathbf{x}} \sum_{u,v} h_{n,u,v,z}(\mathbf{x})$$

In this formula, all neurons in channel  $z$  are equally weighted. Alternatively, you can also maximize random directions, which means that the neurons would be multiplied by different parameters, including negative directions. In this way, we study how the neurons interact within the channel. Instead of maximizing the activation, you can also minimize it (which

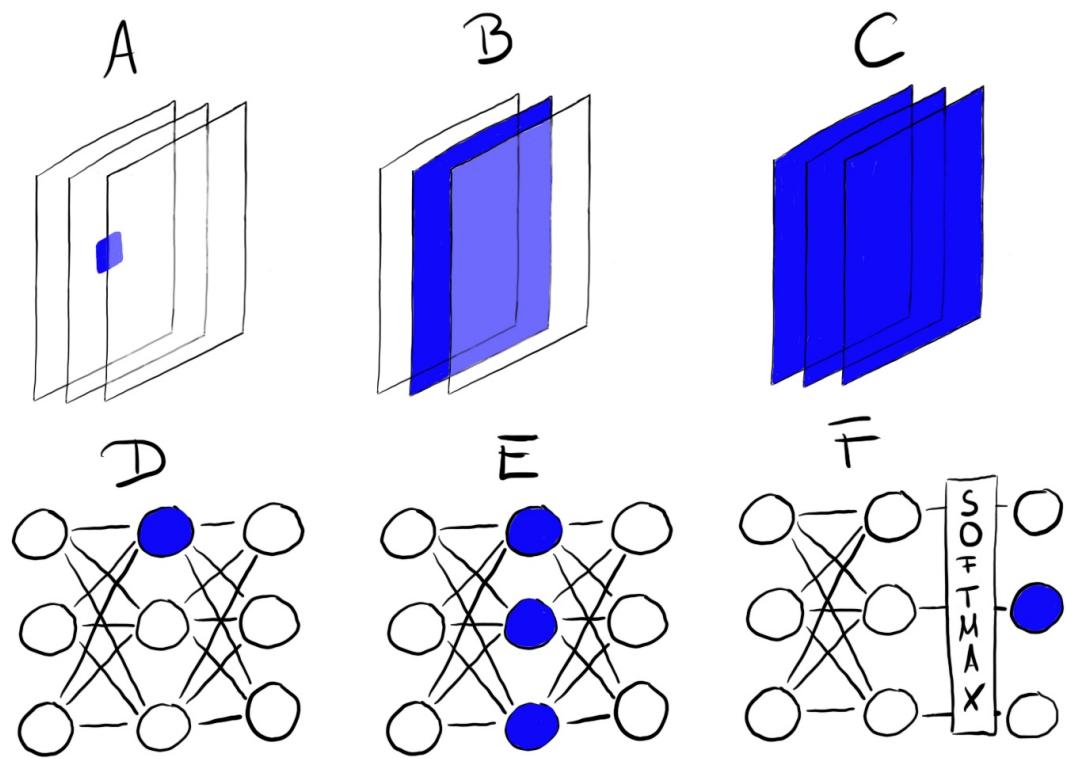


Figure 27.2: Feature visualization can be done for different units. A) Convolution neuron, B) Convolution channel, C) Convolution layer, D) Neuron, E) Hidden layer, F) Class probability neuron (or corresponding pre-softmax neuron)

corresponds to maximizing the negative direction). See Figure 27.3 visualizing both. Interestingly, when you maximize the negative direction you get very different features for the same unit. While the neuron is maximally activated by wheels, something which seems to have eyes yields a negative activation.



Figure 27.3: Positive (left) and negative (right) activation of Inception V1 neuron 484 from layer mixed4d pre relu.

We can address this optimization problem in different ways. For example, instead of generating new images, we could search through our training images and select those that maximize the activation. This is a valid approach, but using training data has the problem that elements on the images can be correlated and we cannot see what the neural network is really looking for. If images that yield a high activation of a certain channel show a dog and a tennis ball, we do not know whether the neural network looks at the dog, the tennis ball, or maybe at both.

Another approach is to generate new images, starting from random noise as visualized in Figure 27.4. To obtain meaningful visualizations, there are usually constraints on the image, e.g. that only small changes are allowed. To reduce noise in the feature visualization, you can apply jittering, rotation, or scaling to the image before the optimization step. Other regularization options include frequency penalization (e.g. reduce variance of neighboring pixels) or generating images with learned priors, e.g. with generative adversarial networks (GANs) (Nguyen et al. 2016) or denoising autoencoders (Nguyen et al. 2017).

If you want to dive a lot deeper into feature visualization, take a look at the distill.pub online journal, especially the feature visualization post by Olah, Mordvintsev, and Schubert (2017), from which I used many of the images. I also recommend the article about the building blocks of interpretability (Olah et al. 2018).

### 27.1.2 Connection to adversarial examples

There's a connection between feature visualization and [adversarial examples](#): Both techniques maximize the activation of a neural network unit. For adversarial examples, we look for the maximum activation of the neuron for the adversarial (= incorrect) class. One difference is the

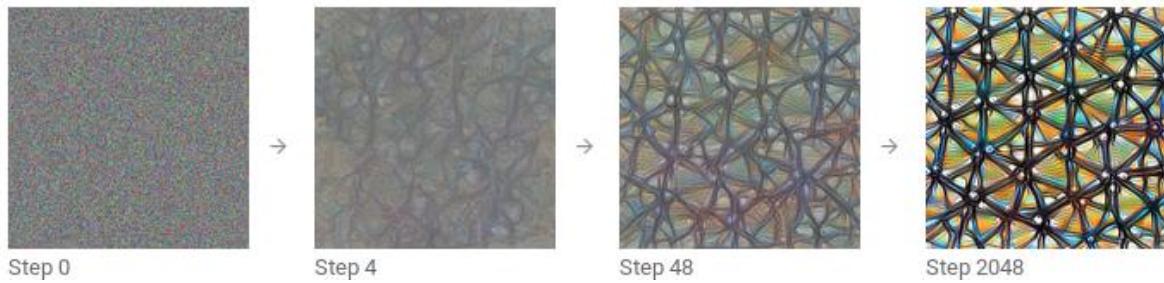


Figure 27.4: Iterative optimization from random image to maximizing activation. Olah, et al. 2017 (CC-BY 4.0), <https://distill.pub/2017/feature-visualization/>.

image we start with: For adversarial examples, it's the image for which we want to generate the adversarial image. For feature visualization, it is, depending on the approach, random noise.

### 27.1.3 Text and tabular data

The literature focuses on feature visualization for convolutional neural networks for image recognition. Technically, there is nothing to stop you from finding the input that maximally activates a neuron of a fully connected neural network for tabular data or a recurrent neural network for text data. You might not call it feature visualization any longer, since the “feature” would be a tabular data input or text. For credit default prediction, the inputs might be the number of prior credits, number of mobile contracts, address, and dozens of other features. The learned feature of a neuron would then be a certain combination of the dozens of features. For recurrent neural networks, it is a bit nicer to visualize what the network learned: Karpathy, Johnson, and Fei-Fei (2015) showed that recurrent neural networks indeed have neurons that learn interpretable features. They trained a character-level model, which predicts the next character in the sequence from the previous characters. Once an opening brace “(” occurred, one of the neurons got highly activated and got deactivated when the matching closing bracket “)” occurred. Other neurons fired at the end of a line. Some neurons fired in URLs. The difference to the feature visualization for CNNs is that the examples were not found through optimization but by studying neuron activations in the training data.

Some of the images seem to show well-known concepts like dog snouts or buildings. But how can we be sure? The Network Dissection method links human concepts with individual neural network units. Spoiler alert: Network Dissection requires extra datasets that someone has labeled with human concepts.

## 27.2 Network Dissection

The Network Dissection approach by Bau et al. (2017) quantifies the interpretability of a unit of a convolutional neural network. It links highly activated areas of CNN channels with human concepts (objects, parts, textures, colors, ...).

The channels of a convolutional neural network learn new features, as we saw in the previous section on [Feature Visualization](#). But these visualizations do not prove that a unit has learned a certain concept. We also do not have a measure for how well a unit detects, e.g., skyscrapers. Before we go into the details of Network Dissection, we have to talk about the big hypothesis that is behind that line of research. The hypothesis is: “Units of a neural network (like convolutional channels) learn disentangled concepts.” Do they?

### The Question of Disentangled Features

Do (convolutional) neural networks learn disentangled features? Disentangled features mean that individual network units detect specific real-world concepts. Convolutional channel 394 might detect skyscrapers, channel 121 dog snouts, channel 12 stripes at a 30-degree angle, ... The opposite of a disentangled network is a completely entangled network. In a completely entangled network, for example, there would be no individual unit for dog snouts. All channels would contribute to the recognition of dog snouts.

Disentangled features imply that the network is highly interpretable. Let’s assume we have a network with completely disentangled units that are labeled with known concepts. This would open up the possibility to track the network’s decision-making process. For example, we could analyze how the network classifies wolves against huskies. First, we identify the “husky”-unit. We can check whether this unit depends on the “dog snout,” “fluffy fur,” and “snow”-units from the previous layer. If it does, we know that it will misclassify an image of a husky with a snowy background as a wolf. In a disentangled network, we could identify problematic non-causal correlations. We could automatically list all highly activated units and their concepts to explain an individual prediction. We could easily detect bias in the neural network. For example, did the network learn a “white skin” feature to predict salary?

Spoiler alert: Convolutional neural networks are not perfectly disentangled. We’ll now look more closely at Network Dissection to find out how interpretable neural networks are.

### 27.2.1 Algorithm

Network Dissection has three steps:

1. Get images with human-labeled visual concepts, from stripes to skyscrapers.
2. Measure the CNN channel activations for these images.
3. Quantify the alignment of activations and labeled concepts.

Figure 27.5 visualizes how an image is forwarded to a channel and matched with the labeled concepts.

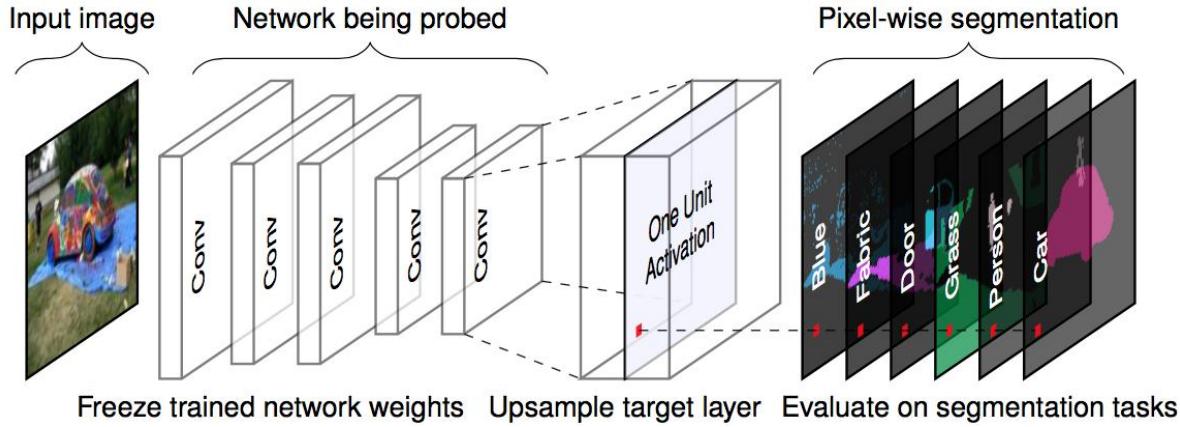


Figure 27.5: For a given input image and a trained network (fixed weights), we propagate the image forward to the target layer, upscale the activations to match the original image size, and compare the maximum activations with the ground truth pixel-wise segmentation. Figure originally from <http://netdissect.csail.mit.edu/>.

### Step 1: Broden dataset

The first difficult but crucial step is data collection. Network Dissection requires pixel-wise labeled images with concepts of different abstraction levels (from colors to street scenes). Bau & Zhou et al. combined a couple of datasets with pixel-wise concepts. They called this new dataset ‘Broden’, which stands for broadly and densely labeled data. The Broden dataset is segmented to the pixel level mostly; for some datasets, the whole image is labeled. Broden contains 60,000 images with over 1,000 visual concepts in different abstraction levels: 468 scenes, 585 objects, 234 parts, 32 materials, 47 textures, and 11 colors.

### Step 2: Retrieve network activations

Next, we create the masks of the top activated areas per channel and per image. At this point, the concept labels are not yet involved.

- For each convolutional channel  $k$ :
  - For each image  $\mathbf{x}$  in the Broden dataset:
    - \* Forward propagate image  $\mathbf{x}$  to the target layer containing channel  $k$ .
    - \* Extract the pixel activations of convolutional channel  $k$ :  $A_k(\mathbf{x})$ .
  - Calculate the distribution of pixel activations  $\alpha_k$  over all images.
  - Determine the 0.995-quantile level  $T_k$  of activations  $\alpha_k$ . This means 0.5% of activations of channel  $k$  in the dataset are greater than  $T_k$ .
  - For each image  $\mathbf{x}$  in the Broden dataset:

- \* Scale the (possibly) lower-resolution activation map  $A_k(\mathbf{x})$  to the resolution of image  $\mathbf{x}$ . We call the result  $S_k(\mathbf{x})$ .
- \* Binarize the activation map: A pixel is either on or off, depending on whether or not it exceeds the activation threshold  $T_k$ . The new mask is  $M_k(\mathbf{x}) = S_k(\mathbf{x}) \geq T_k$ .

### Step 3: Activation-concept alignment

After step 2, we have one activation mask per channel and image. These activation masks mark highly activated areas. For each channel, we want to find the human concept that activates that channel. We find the concept by comparing the activation masks with all labeled concepts. We quantify the alignment between activation mask  $k$  and concept mask  $c$  with the Intersection over Union (IoU) score:

$$IoU_{k,c} = \frac{|M_k(\mathbf{x}) \cap L_c(\mathbf{x})|}{|M_k(\mathbf{x}) \cup L_c(\mathbf{x})|}$$

where  $|\cdot|$  is the cardinality of a set. Intersection over Union compares the alignment between two areas.  $IoU_{k,c}$  can be interpreted as the accuracy with which unit  $k$  detects concept  $c$ . We call unit  $k$  a detector of concept  $c$  when  $IoU_{k,c} > 0.04$ . This threshold was chosen by Bau & Zhou et al. (2017).

Figure 27.6 illustrates the intersection and union of the activation mask and concept mask for a single image, and Figure 27.7 shows a unit that detects dogs.

#### 27.2.2 Experiments

The Network Dissection authors trained different network architectures (AlexNet, VGG, GoogleNet, ResNet) from scratch on different datasets (ImageNet, Places205, Places365). ImageNet contains 1.6 million images from 1000 classes that focus on objects. (Deng et al. 2009) Places205 and Places365 contain 2.5 million / 1.6 million images from 205 / 365 different scenes. The authors additionally trained AlexNet on self-supervised training tasks such as predicting video frame order or colorizing images. For many of these different settings, they counted the number of unique concept detectors as a measure of interpretability. Here are some of the findings:

- The networks detect lower-level concepts (colors, textures) at lower layers and higher-level concepts (parts, objects) at higher layers. We've already seen this in the [Feature Visualizations](#).
- Batch normalization reduces the number of unique concept detectors.
- Many units detect the same concept. For example, there are 95 (!) dog channels in VGG trained on ImageNet when using  $IoU \geq 0.04$  as detection cutoff (4 in conv4\_3, 91 in conv5\_3, see [project website](#)).



Figure 27.6: The Intersection over Union (IoU) is computed by comparing the human ground truth annotation and the top activated pixels.



Figure 27.7: Activation mask for inception\_4e channel 750 which detects dogs with  $IoU = 0.203$ . Figure originally from <http://netdissect.csail.mit.edu/>

- Increasing the number of channels in a layer increases the number of interpretable units.
- Random initializations (training with different random seeds) result in slightly different numbers of interpretable units.
- ResNet is the network architecture with the largest number of unique detectors, followed by VGG, GoogleNet, and AlexNet last.
- The largest number of unique concept detectors is learned for Places365, followed by Places205, and ImageNet last.
- The number of unique concept detectors increases with the number of training iterations.
- Networks trained on self-supervised tasks have fewer unique detectors compared to networks trained on supervised tasks.
- In transfer learning, the concept of a channel can change. For example, a dog detector became a waterfall detector. This happened in a model that was initially trained to classify objects and then fine-tuned to classify scenes.
- In one of the experiments, the authors projected the channels onto a new rotated basis. This was done for the VGG network trained on ImageNet. “Rotated” does not mean that the image was rotated. “Rotated” means that we take the 256 channels from the conv5 layer and compute 256 new channels as linear combinations of the original channels. In the process, the channels get entangled. Rotation reduces interpretability, i.e. the number of channels aligned with a concept decreases. The rotation was designed to keep the performance of the model the same. The first conclusion: Interpretability of CNNs is axis-dependent. This means that random combinations of channels are less likely to detect unique concepts. The second conclusion: Interpretability is independent of discriminative power. The channels can be transformed with orthogonal transformations while the discriminative power remains the same, but interpretability decreases.

The authors also used Network Dissection for Generative Adversarial Networks (GANs). You can find Network Dissection for GANs on [the project’s website](#).

## 27.3 Strengths

Feature visualizations give **unique insight into the working of neural networks**, especially for image recognition. Given the complexity and opacity of neural networks, feature visualization is an important step in analyzing and describing neural networks. Through feature visualization, we have learned that neural networks learn simple edge and texture detectors first, and more abstract part and object detectors in higher layers. Network dissection expands those insights and makes the interpretability of network units measurable.

Network dissection allows us to **automatically link units to concepts**, which is very convenient.

Feature visualization is a great tool to **communicate in a non-technical way how neural networks work**.

With network dissection, we can also **detect concepts beyond the classes in the classification task**. But we need datasets that contain images with pixel-wise labeled concepts.

Feature visualization can be **combined with feature attribution methods**, which explain which pixels were important for the classification. The combination of both methods allows us to explain an individual classification along with a local visualization of the learned features that were involved in the classification. See [The Building Blocks of Interpretability from distill.pub](#).

Finally, feature visualizations make **great desktop wallpapers and T-shirt prints**.

## 27.4 Limitations

**Many feature visualization images are not interpretable** at all, but contain some abstract features for which we have no words or mental concept. The display of feature visualizations along with training data can help. The images still might not reveal what the neural network reacted to and only state something like “maybe there has to be yellow in the images”. Even with Network Dissection, some channels are not linked to a human concept. For example, layer conv5\_3 from VGG trained on ImageNet has 193 channels (out of 512) that could not be matched with a human concept.

There are **too many units to look at**, even when “only” visualizing the channel activations. For the Inception V1 architecture, for example, there are already over 5000 channels from nine convolutional layers. If you also want to show the negative activations plus a few images from the training data that maximally or minimally activate the channel (let’s say four positive, four negative images), then you must already display more than 50,000 images. At least we know – thanks to Network Dissection – that we do not need to investigate random directions.

**Illusion of interpretability?** The feature visualizations can convey the illusion that we understand what the neural network is doing. But do we really understand what is going on in the neural network? Even if we look at hundreds or thousands of feature visualizations, we cannot understand the neural network. The channels interact in a complex way, positive and negative activations are unrelated, multiple neurons might learn very similar features, and for many of the features we do not have equivalent human concepts. We must not fall into the trap of believing we fully understand neural networks just because we believe we saw that neuron 349 in layer 7 is activated by daisies. Network Dissection showed that architectures like ResNet or Inception have units that react to certain concepts. But the IoU is not that great, and often many units respond to the same concept and some to no concept at all. The channels are not completely disentangled, and we cannot interpret them in isolation.

For Network Dissection, **you need datasets that are labeled on the pixel level** with the concepts. These datasets take a lot of effort to collect, since each pixel needs to be labeled, which usually works by drawing segments around objects in the image.

Network Dissection only aligns human concepts with positive activations but not with negative activations of channels. As the feature visualizations showed, negative activations seem to be linked to concepts. This might be fixed by additionally looking at the lower quantile of activations.

## 27.5 Software and further material

There's an open-source implementation of feature visualization called [Lucid](#). You can conveniently try it in your browser by using the notebook links that are provided on the Lucid GitHub page. No additional software is required. Other implementations are [tf\\_cnnvis](#) for TensorFlow, [Keras Filters](#) for Keras, and [DeepVis](#) for Caffe.

Network Dissection has a great [project website](#). Next to the publication, the website hosts additional material such as code, data, and visualizations of activation masks.

## 28 Saliency Maps

Pixel attribution methods highlight the pixels that were relevant for a certain image classification by a neural network. Figure 28.1 is an example of an explanation.

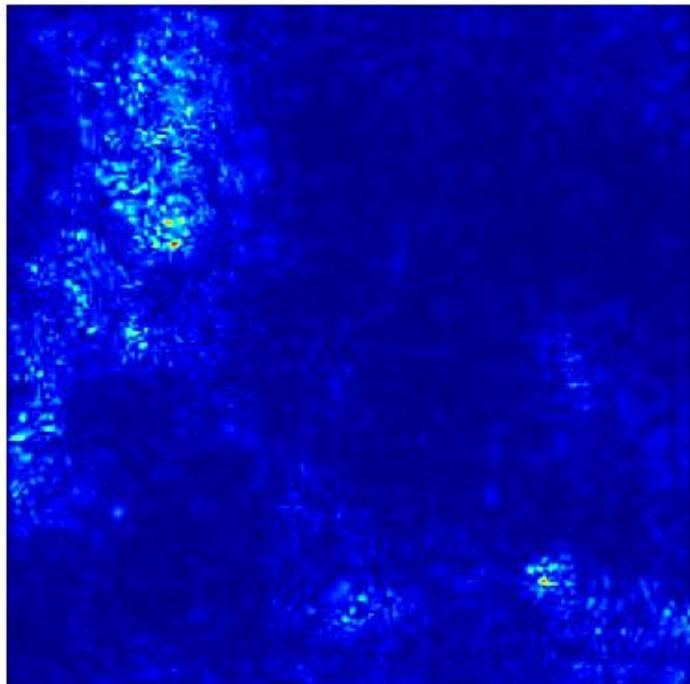


Figure 28.1: A saliency map in which pixels are colored by their contribution to the classification.

You'll see later in this chapter what's going on in this particular image. Pixel attribution methods can be found under various names: sensitivity map, saliency map, pixel attribution map, gradient-based attribution methods, feature relevance, feature attribution, and feature contribution.

Pixel attribution is a special case of feature attribution, but for images. Feature attribution explains individual predictions by attributing each input feature according to how much it changed the prediction (negatively or positively). The features can be input pixels, tabular data, or words. [SHAP](#), [Shapley values](#), and [LIME](#) are examples of general feature attribution methods.

We consider neural networks that output as prediction a vector of length  $C$ , which includes regression where  $C = 1$ . The output of the neural network for image I is called  $S(\mathbf{x}) = [S_1(\mathbf{x}), \dots, S_C(\mathbf{x})]$ . All these methods take as input  $\mathbf{x} \in \mathbb{R}^p$  (can be image pixels, tabular data, words, ...) with  $p$  features and output as explanation a relevance score for each of the  $p$  input features:  $\mathbf{R}^c = [R_1^c, \dots, R_p^c]$ . The  $c$  indicates the relevance for the  $c$ -th output  $S_C(\mathbf{x})$ .

There's a confusing amount of pixel attribution approaches. It helps to understand that there are two different types of attribution methods:

**Occlusion- or perturbation-based:** Methods like [SHAP](#) and [LIME](#) manipulate parts of the image to generate explanations (model-agnostic).

**Gradient-based:** Many methods compute the gradient of the prediction (or classification score) with respect to the input features. The gradient-based methods (of which there are many) mostly differ in how the gradient is computed.

Both approaches have in common that the explanation has the same size as the input image (or at least can be meaningfully projected onto it), and they assign each pixel a value that can be interpreted as the relevance of the pixel to the prediction or classification of that image.

Another useful categorization for pixel attribution methods is the baseline question:

**Gradient-only methods** tell us whether a change in a pixel would change the prediction. Examples are Vanilla Gradient and Grad-CAM (Selvaraju et al. 2017). The interpretation of the gradient-only attribution is: If I were to increase the color values of the pixel, the predicted class probability would go up (for positive gradient) or down (for negative gradient). The larger the absolute value of the gradient, the stronger the effect of a change of this pixel.

**Path-attribution methods** compare the current image to a reference image, which can be an artificial “zero” image such as a completely gray image. The difference in actual and baseline prediction is divided among the pixels. The baseline image can also be multiple images: a distribution of images. This category includes model-specific gradient-based methods such as Deep Taylor and Integrated Gradients (Sundararajan, Taly, and Yan 2017), as well as model-agnostic methods such as LIME and SHAP. Some path-attribution methods are “complete,” meaning that the sum of the relevance scores for all input features is the difference between the prediction of the image and the prediction of a reference image. Examples are SHAP and Integrated Gradients. For path-attribution methods, the interpretation is always done with respect to the baseline: The difference between classification scores of the actual image and the baseline image is attributed to the pixels.

### Pick reference image

The choice of the reference image (distribution) has a big effect on the explanation. The usual assumption is to use a “neutral” image (distribution). Of course, it’s perfectly possible to use your favorite selfie, but you should ask yourself if it makes sense in an application. It would certainly assert dominance among the other project members.

At this point, I would normally give an intuitive explanation about how these methods work, but I think it’s best if we just start with the Vanilla Gradient method, because it shows very nicely the general recipe that many other methods follow.

## 28.1 Vanilla Gradient

The idea of Vanilla Gradient, introduced by Simonyan, Vedaldi, and Zisserman (2014) as one of the first pixel attribution approaches, is quite simple if you already know backpropagation. (They called their approach “Image-Specific Class Saliency,” but I like Vanilla Gradient better.) We calculate the gradient of the loss function for the class we are interested in with respect to the input pixels. This gives us a map of the size of the input features with negative to positive values.

The recipe for this approach is:

1. Perform a forward pass of the image of interest.
2. Compute the gradient of the class score of interest with respect to the input pixels:

$$E_{grad}(\mathbf{x}_0) = \frac{\delta S_c}{\delta \mathbf{x}}|_{\mathbf{x}=\mathbf{x}_0}$$

Here we set all other classes to zero.

3. Visualize the gradients. You can either show the absolute values or highlight negative and positive contributions separately.

More formally, we have an image  $\mathbf{x}$  and the convolutional neural network gives it a score  $S_c(\mathbf{x})$  for class  $c$ . The score is a highly non-linear function of our image. The idea behind using the gradient is that we can approximate that score by applying a first-order Taylor expansion

$$S_c(\mathbf{x}) \approx \mathbf{w}^T \mathbf{x} + b$$

where  $\mathbf{w}$  is the derivative of our score:

$$\mathbf{w} = \frac{\delta S_c}{\delta \mathbf{x}}|_{\mathbf{x}_0}$$

Now, there is some ambiguity in how to perform a backward pass of the gradients, since non-linear units such as ReLU (Rectified Linear Unit) “remove” the sign. So when we do a backpass, we do not know whether to assign a positive or negative activation. Using my incredible ASCII art skill, the ReLU function looks like this:  $\_/\_$  and is defined as  $\text{ReLU}(\mathbf{x}_l) = \max(0, \mathbf{x}_l)$ . This means that when the activation of a neuron is zero, we do not know which value to backpropagate. In the case of Vanilla Gradient, the ambiguity is resolved as follows:

$$\frac{\delta f}{\delta \mathbf{x}_l} = \frac{\delta f}{\delta \mathbf{x}_{l+1}} \cdot I(\mathbf{x}_l > 0)$$

Here,  $I$  is the element-wise indicator function, which is zero where the activation at the lower layer was negative, and one where it's positive or zero. Vanilla Gradient takes the gradient we have backpropagated so far up to layer  $l + 1$ , and then simply sets the gradients to zero where the activation at the layer below is negative.

Let's look at an example where we have layers  $\mathbf{x}_l$  and  $\mathbf{x}_{l+1} = \text{ReLU}(\mathbf{x}_l)$ . Our fictive activation at  $\mathbf{x}_l$  is:

$$\begin{pmatrix} 1 & 0 \\ -1 & -10 \end{pmatrix}$$

And these are our gradients at  $\mathbf{x}_{l+1}$ :

$$\begin{pmatrix} 0.4 & 1.1 \\ -0.5 & -0.1 \end{pmatrix}$$

Then our gradients at  $\mathbf{x}_l$  are:

$$\begin{pmatrix} 0.4 & 0 \\ 0 & 0 \end{pmatrix}$$

Vanilla Gradient has a saturation problem, as explained in Shrikumar, Greenside, and Kundaje (2017). When ReLU is used, and when the activation goes below zero, the activation is capped at zero and does not change anymore. The activation is saturated. For example: The input to the layer is two neurons with weights  $-1$  and  $-1$ , and a bias of  $1$ . When passing through the ReLU layer, the activation will be  $\text{neuron1} + \text{neuron2}$  if the sum of both neurons is  $< 1$ . If the sum of both inputs is greater than  $1$ , the activation will remain saturated at an activation of  $1$  (since the weights are negative). Also, the gradient at this point will be zero, and Vanilla Gradient will say that this neuron is not important.

And now, my dear readers, learn another method, more or less for free: DeconvNet.

## 28.2 DeconvNet

DeconvNet by Zeiler and Fergus (2014) is almost identical to Vanilla Gradient. The goal of DeconvNet is to reverse a neural network, and the paper proposes operations that are reversals of the filtering, pooling, and activation layers. If you take a look into the paper, it looks very different from Vanilla Gradient, but apart from the reversal of the ReLU layer, DeconvNet is equivalent to the Vanilla Gradient approach. Vanilla Gradient can be seen as a generalization of DeconvNet. DeconvNet makes a different choice for backpropagating the gradient through ReLU:

$$R_n = R_{n+1} I(R_{n+1} > 0),$$

where  $R_n$  and  $R_{n+1}$  are the layer reconstructions, and  $I$  is the indicator function. When backpassing from layer  $n$  to layer  $n - 1$ , DeconvNet “remembers” which of the activations in layer  $n$  were set to zero in the forward pass and sets them to zero in layer  $n - 1$ . Activations with a negative value in layer  $n$  are set to zero in layer  $n - 1$ . The gradient  $\mathbf{X}_n$  for the example from earlier becomes:

$$\begin{pmatrix} 0.4 & 1.1 \\ 0 & 0 \end{pmatrix}$$

## 28.3 Grad-CAM

Grad-CAM (Selvaraju et al. 2017) provides visual explanations for CNN decisions. Unlike other methods, the gradient is not backpropagated all the way back to the image, but (usually) to the last convolutional layer to produce a coarse localization map that highlights important regions of the image.

Grad-CAM stands for Gradient-weighted Class Activation Map. And, as the name suggests, it’s based on the gradient of the neural networks. Grad-CAM, like other techniques, assigns each neuron a relevance score for the decision of interest. This decision of interest can be the class prediction (which we find in the output layer), but can theoretically be any other layer in the neural network. Grad-CAM backpropagates this information to the last convolutional layer. Grad-CAM can be used with different CNNs: with fully connected layers, for structured output such as captioning and in multi-task outputs, and for reinforcement learning.

Let’s start with an intuitive consideration of Grad-CAM. The goal of Grad-CAM is to understand at which parts of an image a convolutional layer “looks” for a certain classification. As a reminder, the first convolutional layer of a CNN takes as input the images and outputs feature maps that encode learned features (see the chapter on [Learned Features](#)). The higher-level

convolutional layers do the same, but take as input the feature maps of the previous convolutional layers. To understand how the CNN makes decisions, Grad-CAM analyzes which regions are activated in the feature maps of the last convolutional layers. There are  $k$  feature maps in the last convolutional layer, and I will call them  $A_1, A_2, \dots, A_k$ . How can we “see” from the feature maps how the convolutional neural network has made a certain classification? In the first approach, we could simply visualize the raw values of each feature map, average over the feature maps, and overlay this over our image. This would not be helpful since the feature maps encode information for **all classes**, but we are interested in a particular class. Grad-CAM has to decide how important each of the  $k$  feature maps was to our class  $c$ , that we are interested in. We have to weight each pixel of each feature map with the gradient before we average over the feature maps. This gives us a heatmap which highlights regions that positively or negatively affect the class of interest. This heatmap is sent through the ReLU function, which is a fancy way of saying that we set all negative values to zero. Grad-CAM removes all negative values by using a ReLU function, with the argument that we are only interested in the parts that contribute to the selected class  $c$ , and not to other classes. The word pixel might be misleading here as the feature map is smaller than the image (because of the pooling units) but is mapped back to the original image. We then scale the Grad-CAM map to the interval  $[0, 1]$  for visualization purposes and overlay it over the original image.

Let’s look at the recipe for Grad-CAM. Our goal is to find the localization map, which is defined as:

$$L_{\text{Grad-CAM}}^c \in \mathbb{R}^{U \times V} = \underbrace{\text{ReLU}}_{\text{Pick positive values}} \left( \sum_k \alpha_k^c A^k \right)$$

Here,  $U$  is the width,  $V$  the height of the explanation, and  $c$  the class of interest.

1. Forward-propagate the input image through the convolutional neural network.
2. Obtain the raw score for the class of interest, meaning the activation of the neuron before the softmax layer.
3. Set all other class activations to zero.
4. Back-propagate the gradient of the class of interest to the last convolutional layer before the fully connected layers:  $\frac{\delta y^c}{\delta A^k}$ .
5. Weight each feature map “pixel” by the gradient for the class. Indices  $u$  and  $v$  refer to the width and height dimensions:

$$\alpha_k^c = \overbrace{\frac{1}{Z} \sum_u \sum_v}^{\text{global average pooling}} \underbrace{\frac{\delta y^c}{\delta A_{uv}^k}}_{\text{gradients via backprop}}$$

This means that the gradients are globally pooled.

6. Calculate an average of the feature maps, weighted per pixel by the gradient.

7. Apply ReLU to the averaged feature map.
8. For visualization: Scale values to the interval  $[0, 1]$ . Upscale the image and overlay it over the original image.
9. Additional step for Guided Grad-CAM: Multiply heatmap with guided backpropagation.

## 28.4 Guided Grad-CAM

From the description of Grad-CAM, you can guess that the localization is very coarse, since the last convolutional feature maps have a much coarser resolution compared to the input image. In contrast, other attribution techniques backpropagate all the way to the input pixels. They are therefore much more detailed and can show you individual edges or spots that contributed most to a prediction. A fusion of both methods is called Guided Grad-CAM, and it's super simple. You compute for an image both the Grad-CAM explanation and the explanation from another attribution method, such as Vanilla Gradient. The Grad-CAM output is then upsampled with bilinear interpolation, and then both maps are multiplied element-wise. Grad-CAM works like a lens that focuses on specific parts of the pixel-wise attribution map.

## 28.5 SmoothGrad

The idea of SmoothGrad by Smilkov et al. (2017) is to make gradient-based explanations less noisy by adding noise and averaging over these artificially noisy gradients. SmoothGrad is not a standalone explanation method, but an extension to any gradient-based explanation method.

SmoothGrad works in the following way:

1. Generate multiple versions of the image of interest by adding noise to it.
2. Create pixel attribution maps for all images.
3. Average the pixel attribution maps.

Yes, it's that simple. Why should this work? The theory is that the derivative fluctuates greatly at small scales. Neural networks have no incentive during training to keep the gradients smooth; their goal is to classify images correctly. Averaging over multiple maps "smooths out" these fluctuations:

$$R_{sg}(\mathbf{x}) = \frac{1}{N} \sum_{i=1}^N R(\mathbf{x} + \mathbf{g}_i),$$

Here,  $\mathbf{g}_i \sim N(0, \sigma^2)$  are noise vectors sampled from the Gaussian distribution. The "ideal" noise level depends on the input image and the network. The authors suggest a noise level of

10%-20%, which means that  $\frac{\sigma}{x_{max} - x_{min}}$  should be between 0.1 and 0.2. The limits  $x_{min}$  and  $x_{max}$  refer to minimum and maximum pixel values of the image. The other parameter is the number of samples n, for which it was suggested to use n = 50, since there are diminishing returns above that.

## 28.6 Examples

Let's see some examples of what these maps look like and how the methods compare qualitatively. The network under examination is VGG-16 (Simonyan and Zisserman 2015), which was trained on ImageNet and can therefore distinguish 1,000 classes. For the following images, we will create explanations for the class with the highest classification score.

Figure 28.2 shows the images and their classification by the neural network:

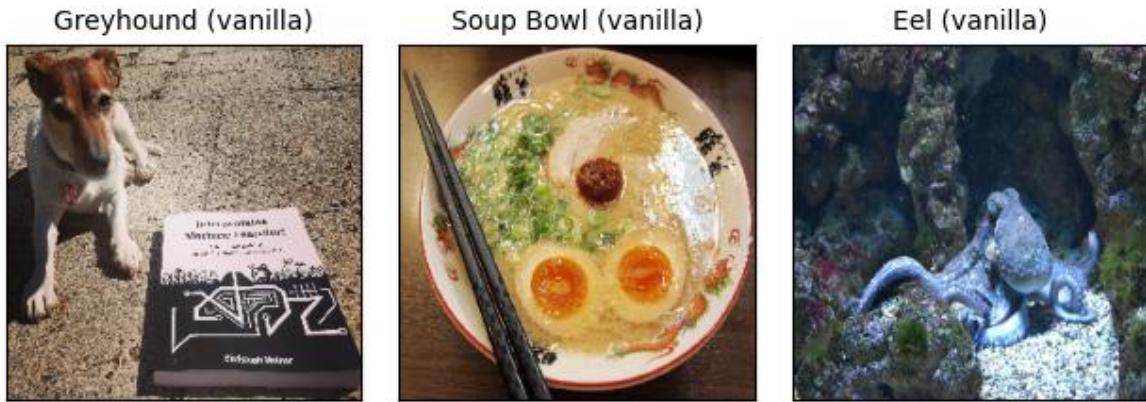


Figure 28.2: Images of a dog classified as greyhound, a ramen soup classified as soup bowl, and an octopus classified as eel.

The image on the left with the honorable dog guarding the Interpretable Machine Learning book was classified as “Greyhound” with a probability of 35% (seems like “Interpretable Machine Learning book” was not one of the 20k classes). The image in the middle shows a bowl of yummy ramen soup and is correctly classified as “Soup Bowl” with a probability of 50%. The third image shows an octopus on the ocean floor, which is incorrectly classified as “Eel” with a high probability of 70%.

Figure 28.3 shows the pixel attributions that aim to explain the classification.

Unfortunately, it's a bit of a mess. But let's look at the individual explanations, starting with the dog. Vanilla Gradient and SmoothGrad both highlight the dog, which makes sense. But they also highlight some areas around the book, which is odd. Grad-CAM highlights only the book area, which makes no sense at all. And from here on, it gets a bit messier. The Vanilla

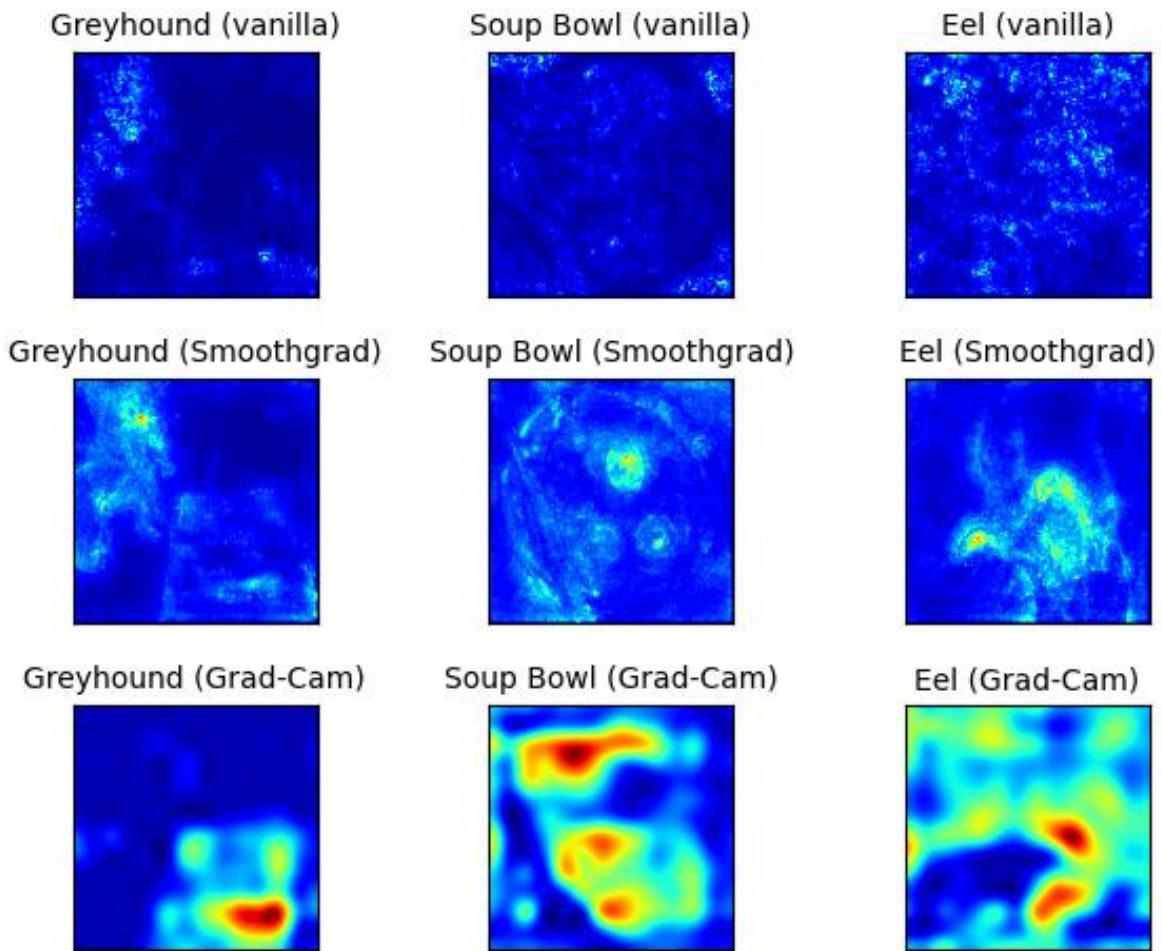


Figure 28.3: Pixel attributions or saliency maps for the Vanilla Gradient method, SmoothGrad and Grad-CAM.

Gradient method seems to fail for both the soup bowl and the octopus (or, as the network thinks, eel). Both images look like afterimages from looking into the sun for too long. (Please do not look at the sun directly). SmoothGrad helps a lot; at least the areas are more defined. In the soup example, some of the ingredients are highlighted, such as the eggs and the meat, but also the area around the chopsticks. In the octopus image, mostly the animal itself is highlighted. For the soup bowl, Grad-CAM highlights the egg part and, for some reason, the upper part of the bowl. The octopus explanations by Grad-CAM are even messier.

You can already see here the difficulties in assessing whether we trust the explanations. As a first step, we need to consider which parts of the image contain information that is relevant to the image's classification. But then we also need to think about what the neural network might have used for the classification. Perhaps the soup bowl was correctly classified based on the combination of eggs and chopsticks, as SmoothGrad implies? Or maybe the neural network recognized the shape of the bowl plus some ingredients, as Grad-CAM suggests? We just do not know.

And that is the big issue with all of these methods. We do not have a ground truth for the explanations. We can only, in a first step, reject explanations that obviously make no sense (and even in this step we do not have strong confidence). The prediction process in the neural network is very complicated.

## 28.7 Strengths

The explanations are **visual** and we are quick to recognize images. In particular, when methods only highlight important pixels, it's easy to immediately recognize the important regions of the image.

Gradient-based methods are usually **faster to compute than model-agnostic methods**. For example, [LIME](#) and [SHAP](#) can also be used to explain image classifications, but are more expensive to compute.

There are **many methods to choose from**.

## 28.8 Limitations

As with most interpretation methods, it's **difficult to know whether an explanation is correct**, and a huge part of the evaluation is only qualitative ("These explanations look about right, let's publish the paper already").

Pixel attribution methods can be very **fragile**. Ghorbani, Abid, and Zou (2019) showed that introducing small (adversarial) perturbations to an image, which still lead to the same prediction, can lead to very different pixels being highlighted as explanations.

Kindermans et al. (2019) also showed that these pixel attribution methods **can be highly unreliable**. They added a constant shift to the input data, meaning they added the same pixel changes to all images. They compared two networks, the original network and the “shifted” network where the bias of the first layer is changed to adapt for the constant pixel shift. Both networks produce the same predictions. Further, the gradient is the same for both. But the explanations changed, which is an undesirable property. They looked at DeepLift, Vanilla Gradient, and Integrated Gradients.

#### Compare multiple methods

Be cautious when relying solely on pixel attribution methods for interpretation. Small perturbations to the input can lead to drastically different explanations, even if the predictions remain unchanged. Always validate explanations with multiple methods to ensure robustness.

The paper “Sanity checks for saliency maps” (Adebayo et al. 2018) investigated whether saliency methods are **insensitive to model and data**. Insensitivity is highly undesirable because it would mean that the “explanation” is unrelated to model and data. Methods that are insensitive to model and training data are similar to edge detectors. Edge detectors simply highlight strong pixel color changes in images and are unrelated to a prediction model or abstract features of the image, and require no training. The methods tested were Vanilla Gradient, Gradient x Input, Integrated Gradients, Guided Backpropagation, Guided Grad-CAM, and SmoothGrad (with Vanilla Gradient). Vanilla Gradient and Grad-CAM passed the insensitivity check, while Guided Backpropagation and Guided Grad-CAM failed. However, the sanity checks paper itself has found some criticism from Tomsett et al. (2020) with a paper called “Sanity checks for saliency metrics” (of course). They found that there is a lack of consistency for evaluation metrics (I know, it’s getting pretty meta now). So we are back to where we started ... It remains difficult to evaluate the visual explanations. This makes it very difficult for a practitioner.

All in all, this is a **very unsatisfactory state of affairs**. We have to wait a little bit for more research on this topic. And please, no more invention of new saliency methods, but more scrutiny of how to evaluate them.

## 28.9 Software

There are several software implementations of pixel attribution methods. For the example, I used [tf-keras-vis](#). One of the most comprehensive libraries is [iNNvestigate](#) (Alber et al. 2019), which implements Vanilla Gradient, SmoothGrad, DeconvNet, Guided Backpropagation, PatternNet, LRP (Bach et al. 2015), and more. A lot of the methods are implemented in the [DeepExplain Toolbox](#).

# 29 Detecting Concepts

*Author: Fangzhou Li @ University of California, Davis*

So far, we have encountered many methods to explain black box models through feature attribution. However, there are some limitations regarding the feature-based approach. First, features are not necessarily user-friendly in terms of interpretability. For example, the importance of a single pixel in an image usually does not convey much meaningful interpretation. Second, the expressiveness of a feature-based explanation is constrained by the number of features.

The concept-based approach addresses both limitations mentioned above. A concept can be any abstraction, such as a color, an object, or even an idea. Given any user-defined concept, although a neural network might not be explicitly trained with the given concept, the concept-based approach detects that concept embedded within the latent space learned by the network. In other words, the concept-based approach can generate explanations that are not limited by the feature space of a neural network.

In this chapter, we will primarily focus on the Testing with Concept Activation Vectors (TCAV) paper by Kim et al. (2018).

## 29.1 TCAV: Testing with Concept Activation Vectors

TCAV is proposed to generate global explanations for neural networks, but in theory, it should also work for any model where taking a directional derivative is possible. For any given concept, TCAV measures the extent of that concept's influence on the model's prediction for a certain class. For example, TCAV can answer questions such as how the concept of "striped" influences a model classifying an image as a "zebra." Since TCAV describes the relationship between a concept and a class, instead of explaining a single prediction, it provides useful global interpretation for a model's overall behavior.

 Carefully define concept

When using TCAV, ensure that the concept you're exploring is distinct and well-defined. This will help in accurately measuring its influence on model predictions.

### 29.1.1 Concept Activation Vector (CAV)

A CAV is simply the numerical representation that generalizes a concept in the activation space of a neural network layer. A CAV, denoted as  $\mathbf{v}_l^C$ , depends on a concept  $C$  and a neural network layer  $l$ , where  $l$  is also called a bottleneck of the model. For calculating the CAV of a concept  $C$ , first, we need to prepare two datasets: a concept dataset, which represents  $C$ , and a random dataset that consists of arbitrary data. For instance, to define the concept of “striped,” we can collect images of striped objects as the concept dataset, while the random dataset is a group of random images without stripes. Next, we target a hidden layer  $l$  and train a binary classifier that separates the activations generated by the concept set from those generated by the random set. The coefficient vector of this trained binary classifier is then the CAV  $\mathbf{v}_l^C$ . In practice, we can use an SVM or a logistic regression model as the binary classifier. Lastly, given an image input  $\mathbf{x}$ , we can measure its “conceptual sensitivity” by calculating the directional derivative of the prediction in the direction of the unit CAV:

$$S_{C,k,l}(\mathbf{x}) = \nabla h_{l,k}(\hat{f}_l(\mathbf{x})) \cdot \mathbf{v}_l^C$$

where  $\hat{f}_l$  maps the input  $\mathbf{x}$  to the activation vector of the layer  $l$ , and  $h_{l,k}$  maps the activation vector to the logit output of class  $k$ .

Mathematically, the sign of  $S_{C,k,l}(\mathbf{x})$  only depends on the angle between the gradient of  $h_{l,k}(\hat{f}_l(\mathbf{x}))$  and  $\mathbf{v}_l^C$ . If the angle is less than 90 degrees,  $S_{C,k,l}(\mathbf{x})$  will be positive, and if the angle is greater than 90 degrees,  $S_{C,k,l}(\mathbf{x})$  will be negative. Since the gradient  $\nabla h_{l,k}$  points to the direction that maximizes the output the most rapidly, conceptual sensitivity  $S_{C,k,l}$ , intuitively, indicates whether  $\mathbf{v}_l^C$  points to the similar direction that maximizes  $h_{l,k}$ . Thus,  $S_{C,k,l}(\mathbf{x}) > 0$  can be interpreted as concept  $C$  encouraging the model to classify  $\mathbf{x}$  into class  $k$ .

### 29.1.2 Testing with CAVs (TCAV)

In the last paragraph, we have learned how to calculate the conceptual sensitivity of a single data point. However, our goal is to produce a global explanation that indicates an overall conceptual sensitivity of an entire class. A very straightforward approach done by TCAV is to calculate the ratio of inputs with positive conceptual sensitivities to the number of inputs for a class:

$$TCAV_{Q,C,k,l} = \frac{|\{\mathbf{x} \in \mathbf{X}_k : S_{C,k,l}(\mathbf{x}) > 0\}|}{|\mathbf{X}_k|}$$

Going back to our example, we are interested in how the concept of “striped” influences the model while classifying images as “zebra.” We collect data that are labeled as “zebra” and

calculate conceptual sensitivity for each input image. Then the TCAV score of the concept “striped” with predicting class “zebra” is the number of “zebra” images that have positive conceptual sensitivities divided by the total number of the “zebra” images. In other words, a *TCAV* with  $C = \text{striped}$  and  $k = \text{zebra}$  equal to 0.8 indicates that 80% of predictions for the “zebra” class are positively influenced by the concept of “striped.”

This looks great, but how do we know our TCAV score is meaningful? After all, a CAV is trained by user-selected concepts and random datasets. If datasets used to train the CAV are bad, the explanation can be misleading and useless. And thus, we perform a simple statistical significance test to help TCAV become more reliable. That is, instead of training only one CAV, we train multiple CAVs using different random datasets while keeping the concept dataset the same. A meaningful concept should generate CAVs with consistent TCAV scores. The more detailed test procedure is shown in the following:

1. Collect  $N$  random datasets, where it is recommended that  $N$  is at least 10.
2. Fix the concept dataset and calculate TCAV score using each of  $N$  random datasets.
3. Apply a two-sided t-test to  $N$  TCAV scores against other  $N$  TCAV scores generated by a random CAV. A random CAV can be obtained by choosing a random dataset as the concept dataset.

#### Check t-test assumptions

When performing the two-sided t-test for TCAV scores, make sure that your data meets the assumptions of the test, such as normality and homogeneity of variances. Otherwise, consider using non-parametric tests.

It's also suggested to apply a multiple testing correction method here if you have multiple hypotheses. The original paper uses Bonferroni correction, and here the number of hypotheses is equal to the number of concepts you are testing.

## 29.2 Example

Let's see an example available on the TCAV [GitHub](#). Continuing the “zebra” class example we have been using previously, Figure 29.1 shows the result of the TCAV scores of “striped,” “zigzagged,” and “dotted” concepts. The image classifier we are using is InceptionV3 (Szegedy et al. 2016), a convolutional neural network trained using ImageNet data. Each concept or random dataset contains 50 images, and we are using 10 random datasets for the statistical significance test with the significance level of 0.05. We are not using the Bonferroni correction because we only have a few random datasets, but it is recommended to add the correction in practice to avoid false discovery.

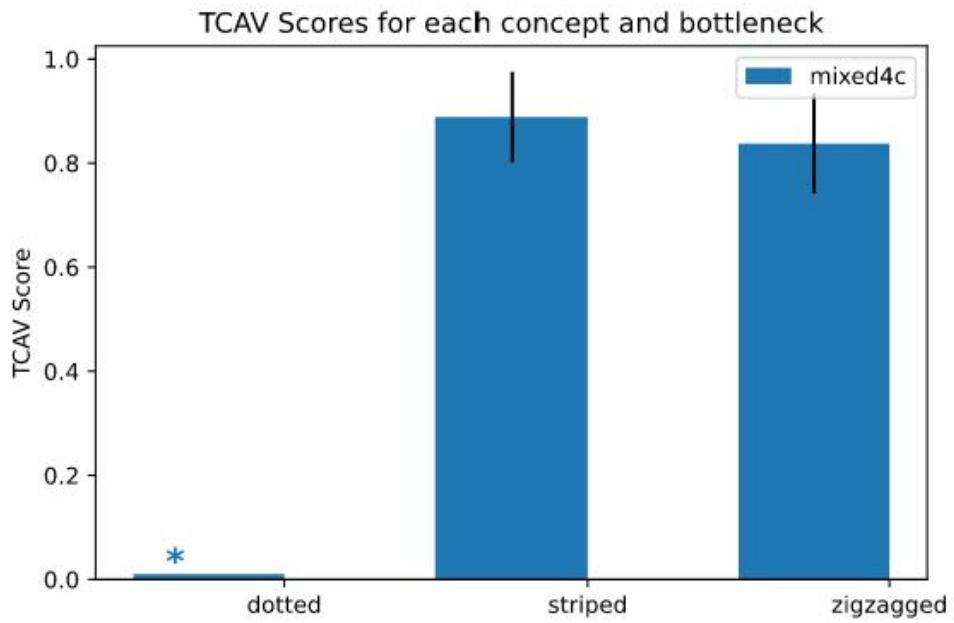


Figure 29.1: Measuring TCAV scores of three concepts for the model predicting “zebra.” The targeted bottleneck is a layer called “mixed4c.” A star sign above “dotted” indicates that “dotted” has not passed the statistical significance test, i.e., having the p-value larger than 0.05. Both “striped” and “zigzagged” have passed the test, and both concepts are useful for the model to identify “zebra” images according to TCAV. Figure originally from the TCAV GitHub.

### Use more than 50 images

In practice, you may want to use more than 50 images in each dataset to train better CAVs. You may also want to use more than 10 random datasets to perform better statistical significance tests. You can also apply TCAV to multiple bottlenecks to have a more thorough observation.

## 29.3 Strengths

Since users are only required to collect data for training the concepts that they are interested in, **TCAV does not require users to have machine learning expertise**. This allows TCAV to be extremely useful for domain experts to evaluate their complicated neural network models.

Another unique characteristic of TCAV is its **customizability** enabled by TCAV's **explanations beyond feature attribution**. Users can investigate any concept as long as the concept can be defined by its concept dataset. In other words, a user can control the balance between the complexity and the interpretability of explanations based on their needs: If a domain expert understands the problem and concept very well, they can shape the concept dataset using more complicated data to generate a more fine-grained explanation.

Finally, TCAV generates **global explanations** that relate concepts to any class. A global explanation gives you an idea of whether your overall model behaves properly or not, which usually cannot be done by local explanations. And thus, TCAV can be used to identify potential “flaws” or “blind spots” that happen during the model training: Maybe your model has learned to weight a concept inappropriately. If a user can identify those ill-learned concepts, they can use the knowledge to **improve their model**. Let's say there is a classifier that predicts “zebra” with a high accuracy. TCAV, however, shows that the classifier is more sensitive towards the concept of “dotted” instead of “striped”. This might indicate that the classifier is accidentally trained by an unbalanced dataset, allowing you to improve the model by either adding more “striped zebra” images or fewer “dotted zebra” images to the training dataset.

## 29.4 Limitations

TCAV might **perform badly on shallower neural networks**. As many papers suggested (e.g., Alain and Bengio (2018)), concepts in deeper layers are more separable. If a network is too shallow, its layers may not be capable of separating concepts clearly, so that TCAV is not applicable.

Since TCAV requires **additional annotations** for concept datasets, it can be very expensive for tasks that do not have readily labeled data. A possible alternative to TCAV when annotating is expensive is to use ACE, which we will briefly talk about in the next section.

Although TCAV is hailed because of its customizability, it is **difficult to apply to concepts that are too abstract or general**. This is mainly because TCAV describes a concept by its corresponding concept dataset. The more abstract or general a concept is, such as “happiness”, the more data are required to train a CAV for that concept.

Though TCAV gains popularity in applying to image data, it has **relatively limited applications in text data and tabular data**.

## 29.5 Other concept-based approaches

The concept-based approach has attracted increasing popularity in recent times, and there are many new methods inspired by the utilization of concepts. Here we would like to briefly mention these methods, and we recommend you read the original works if you are interested.

Automated Concept-based Explanation (ACE) (Ghorbani et al. 2019) can be seen as the automated version of TCAV. ACE goes through a set of images of a class and automatically generates concepts based on the clustering of image segments.

Concept bottleneck models (CBM) (Koh et al. 2020) are intrinsically interpretable neural networks. A CBM is similar to an encoder-decoder model, where the first half of the CBM maps inputs to concepts, and the second half uses the mapped concepts to predict model outputs. Each neuron activation of the bottleneck layer then represents the importance of a concept. Furthermore, users can manipulate the neuron activations of the bottleneck to generate counterfactual explanations of the model.

Concept whitening (CW) (Chen, Bei, and Rudin 2020) is another approach to generate intrinsically interpretable image classifiers. To use CW, one substitutes a normalization layer, such as a batch normalization layer, with a CW layer. And thus, CW is very useful when users want to transform their pre-trained image classifiers to be intrinsically interpretable while maintaining the model performance. CW is heavily inspired by the whitening transformation, and we highly recommend you study the mathematics behind the whitening transformation if you are interested in learning more about CW.

## 29.6 Software

The official Python library of [TCAV](#) requires TensorFlow, but there are other versions implemented online. The easy-to-use Jupyter notebooks are also accessible on the [tensorflow/tcav](#).

# 30 Adversarial Examples

An adversarial example is an instance with small, intentional feature perturbations that cause a machine learning model to make a false prediction. I recommend reading the chapter about [Counterfactual Explanations](#) first, as the concepts are very similar. Adversarial examples are counterfactual examples with the aim to deceive the model, not interpret it.

Why are we interested in adversarial examples? Are they not just curious by-products of machine learning models without practical relevance? The answer is a clear “no”. Adversarial examples make machine learning models vulnerable to attacks, as in the following scenarios.

A self-driving car crashes into another car because it ignores a stop sign. Someone had placed a picture over the sign, which looks like a stop sign with a little dirt for humans, but was designed to look like a parking prohibition sign for the sign recognition software of the car.

A spam detector fails to classify an email as spam. The spam mail has been designed to resemble a normal email, but with the intention of cheating the recipient.

A machine-learning powered scanner scans suitcases for weapons at the airport. A knife was developed to avoid detection by making the system think it is an umbrella.

Let's take a look at some ways to create adversarial examples.

## 30.1 Methods and examples

There are many techniques to create adversarial examples. Most approaches suggest minimizing the distance between the adversarial example and the instance to be manipulated, while shifting the prediction to the desired (adversarial) outcome. Some methods require access to the gradients of the model, which of course only works with gradient-based models such as neural networks, while other methods only require access to the prediction function, which makes these methods model-agnostic. The methods in this section focus on image classifiers with deep neural networks, as a lot of research is done in this area and the visualization of adversarial images is very educational. Adversarial examples for images are images with intentionally perturbed pixels with the aim to deceive the model during application time. The examples impressively demonstrate how easily deep neural networks for object recognition can be deceived by images that appear harmless to humans. If you have not yet seen these examples, you might be surprised because the changes in predictions are incomprehensible for a human observer. Adversarial examples are like optical illusions, but for machines.

## Something is Wrong With My Dog

Szegedy et al. (2014) used a gradient-based optimization approach in their work “Intriguing Properties of Neural Networks” to find adversarial examples for deep neural networks. One result can be seen in Figure 30.1.

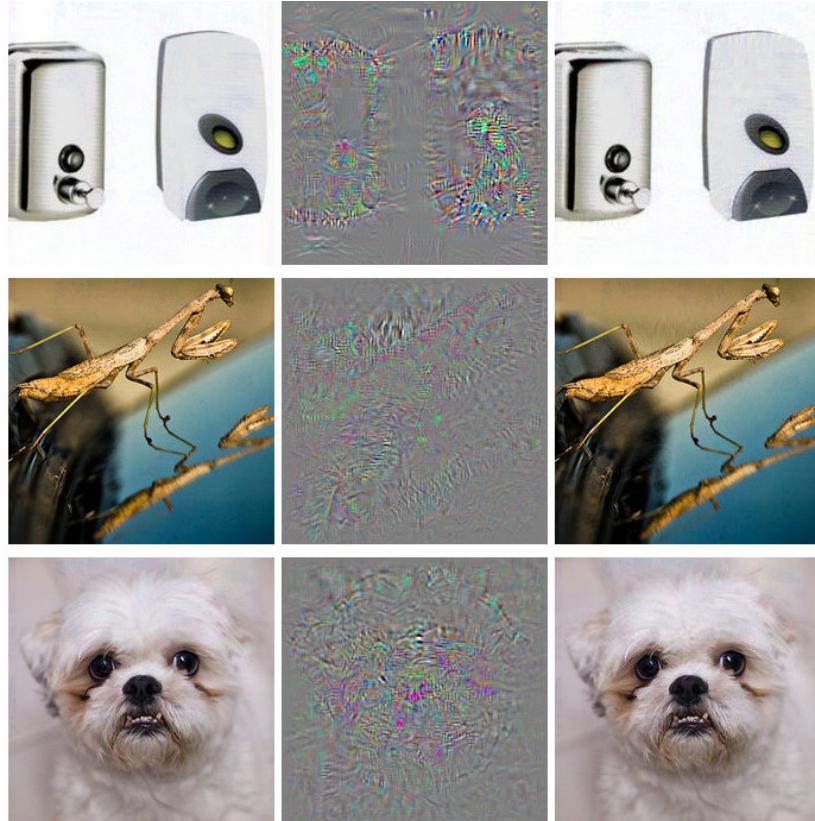


Figure 30.1: Adversarial examples for AlexNet by Szegedy et al. (2013). All images in the left column are correctly classified. The middle column shows the (magnified) error added to the images to produce the images in the right column all categorized (incorrectly) as “Ostrich”. “Intriguing properties of neural networks”, Figure 5 by Szegedy et al. CC-BY 3.0.

These adversarial examples were generated by minimizing the following function with respect to  $\mathbf{r}$ :

$$\text{loss}(\hat{f}(\mathbf{x} + \mathbf{r}), l) + c \cdot |\mathbf{r}|$$

In this formula,  $\mathbf{x}$  is an image (represented as a vector of pixels),  $\mathbf{r}$  is the change to the pixels to create an adversarial image ( $\mathbf{x} + \mathbf{r}$  produces a new image),  $l$  is the desired outcome class,

and the parameter  $c$  is used to balance the distance between images and the distance between predictions. The first term is the distance between the predicted outcome of the adversarial example and the desired class  $l$ ; the second term measures the distance between the adversarial example and the original image. This formulation is almost identical to the loss function to generate [counterfactual explanations](#). There are additional constraints for  $\mathbf{r}$  so that the pixel values remain between 0 and 1. The authors suggest solving this optimization problem with a box-constrained L-BFGS, an optimization algorithm that works with gradients.

### **Disturbed panda: Fast gradient sign method**

Goodfellow, Shlens, and Szegedy (2015) invented the fast gradient sign method for generating adversarial images. The gradient sign method uses the gradient of the underlying model to find adversarial examples. The original image  $\mathbf{x}$  is manipulated by adding or subtracting a small error  $\epsilon$  to each pixel. Whether we add or subtract  $\epsilon$  depends on whether the sign of the gradient for a pixel is positive or negative. Adding errors in the direction of the gradient means that the image is intentionally altered so that the model classification fails.

The following formula describes the core of the fast gradient sign method:

$$\mathbf{x}' = \mathbf{x} + \epsilon \cdot \text{sign}(\nabla_{\mathbf{x}} J(\theta, \mathbf{x}, y))$$

where  $\nabla_{\mathbf{x}} J$  is the gradient of the model's loss function with respect to the original input pixel vector  $\mathbf{x}$ ,  $y$  is the true label for  $\mathbf{x}$ , and  $\theta$  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, Shlens, and Szegedy (2015) suggested adding adversarial examples to the training data to learn robust models.

### **A jellyfish ... No, wait. A bathtub: 1-pixel attacks**

The approach presented by Goodfellow and colleagues (2014) requires many pixels to be changed, if only by a little. But what if you can only change a single pixel? Would you be able to deceive a machine learning model? Su, Vargas, and Sakurai (2019) showed that

it is actually possible to deceive image classifiers by changing a single pixel, as illustrated in Figure 30.2.

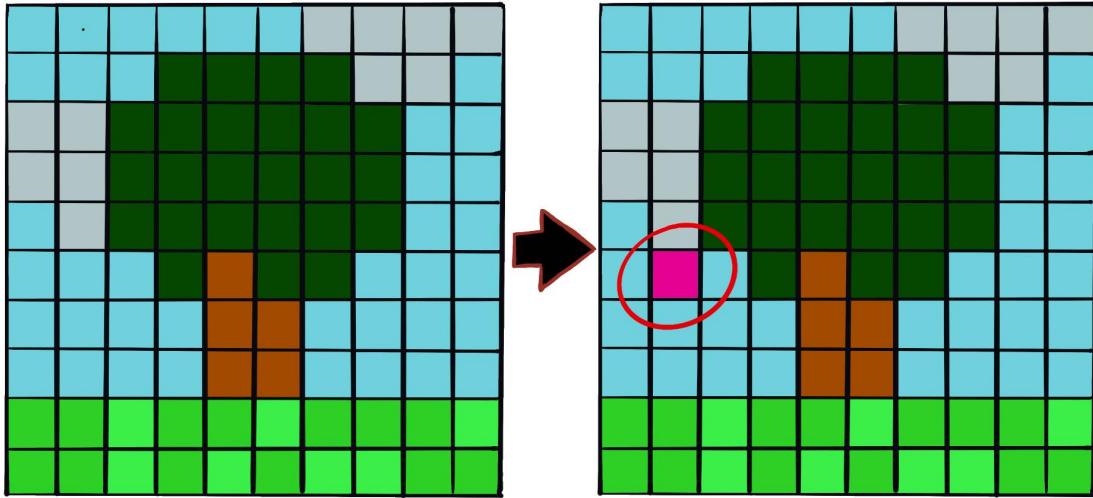


Figure 30.2: By intentionally changing a single pixel a neural network trained on ImageNet can be deceived to predict the wrong class instead of the original class.

Similar to counterfactuals, the 1-pixel attack looks for a modified example  $\mathbf{x}'$  that comes close to the original image  $\mathbf{x}$ , 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 the biological evolution of species. A population of individuals called candidate solutions recombines generation by generation until a solution is found. Each candidate solution encodes a pixel modification and is represented by a vector of five elements: the  $x$ - and  $y$ -coordinates and the red, green, and blue (RGB) values. The search starts with, for example, 400 candidate solutions (= pixel modification suggestions) and creates a new generation of candidate solutions (children) from the parent generation using the following formula:

$$\mathbf{x}_{g+1}^{(i)} = \mathbf{x}_g^{(r1)} + F \cdot (\mathbf{x}_g^{(r2)} - \mathbf{x}_g^{(r3)})$$

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 maximum number of 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. (2018) designed a printable label that can be stuck next to objects to make them look like toasters for an image classifier, see Figure 30.3. Brilliant work!



Figure 30.3: A sticker that makes a VGG16 classifier trained on ImageNet categorize an image of a banana as a toaster. Work by Brown et al. (2017).

This method differs from the methods presented so far for adversarial examples since the restriction that the adversarial image must be very close to the original image is removed. Instead, the method completely replaces a part of the image with a patch that can take on any shape. The image of the patch is optimized over different background images, with different positions of the patch on the images, sometimes moved, sometimes larger or smaller, and rotated, so that the patch works in many situations. In the end, this optimized image can be printed and used to deceive image classifiers in the wild.

### **Never bring a 3D-printed turtle to a gunfight – even if your computer thinks it is a good idea: Robust adversarial examples**

The next method is literally adding another dimension to the toaster: Athalye et al. (2018) 3D-printed a turtle that was designed to look like a rifle to a deep neural network from almost all possible angles, as illustrated in Figure 30.4. Yeah, you read that right. A physical object that looks like a turtle to humans looks like a rifle to the computer!

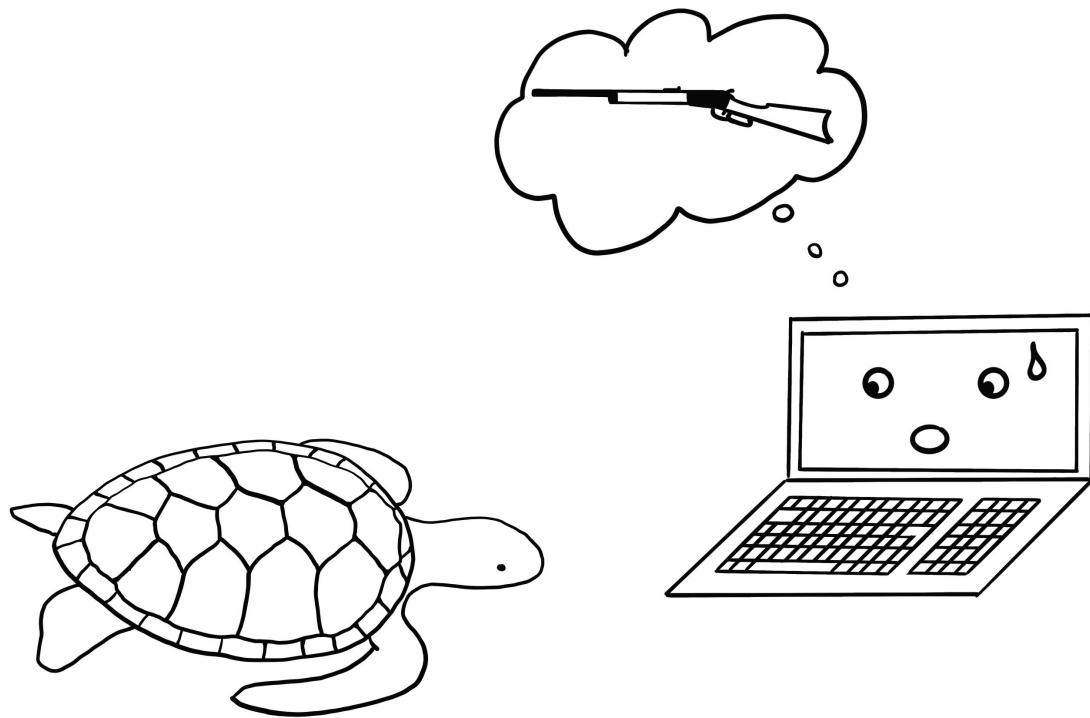


Figure 30.4: Athalye et al. (2017) created a 3D-printed object that is recognized as a rifle by TensorFlow's standard pre-trained InceptionV3 classifier.

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 the viewing angle changes. Athalye et al. (2017) propose the Expectation Over Transformation (EOT) algorithm, which is a method for generating adversarial examples that even work when the image is transformed. The main idea behind EOT is to optimize adversarial examples across many possible transformations. Instead of minimizing the distance between the adversarial example and the original image, EOT keeps the expected distance between the two below a certain threshold, given a selected distribution of possible transformations. The expected distance under transformation can be written as:

$$\mathbb{E}_{t \sim T}[d(t(\mathbf{x}'), t(\mathbf{x}))]$$

where  $\mathbf{x}$  is the original image,  $t(\mathbf{x})$  the transformed image (e.g., rotated),  $\mathbf{x}'$  the adversarial example, and  $t(\mathbf{x}')$  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  $\mathbf{x}'$  that maximizes the probability for the selected class  $y_t$  (e.g., “rifle”) across the distribution of possible transformations  $T$ :

$$\arg \max_{\mathbf{x}'} \mathbb{E}_{t \sim T}[\log \mathbb{P}(y_t | t(\mathbf{x}'))]$$

With the constraint that the expected distance over all possible transformations between adversarial example  $\mathbf{x}'$  and original image  $\mathbf{x}$  remains below a certain threshold:

$$\mathbb{E}_{t \sim T}[d(t(\mathbf{x}'), t(\mathbf{x}))] < \epsilon \quad \text{and} \quad \mathbf{x} \in [0, 1]^d$$

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 that looks like a turtle?

### **The blindfolded adversary: Black box attack**

Imagine the following scenario: I give you access to my great image classifier via Web API. You can get predictions from the model, but you do not have access to the model parameters. From the convenience of your couch, you can send data, and my service answers with the corresponding classifications. Most adversarial attacks are not designed to work in this scenario because they require access to the gradient of the underlying deep neural network to find adversarial examples. Papernot et al. (2017) 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 a 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 cannot be inspected or downloaded. The authors were able to find adversarial examples for various providers, with up to 84% of the adversarial examples being misclassified.

The method even works if the black box model to be deceived is not a neural network. This includes machine learning models without gradients, such as decision trees.

## 30.2 The cybersecurity perspective

Machine learning deals with known unknowns: predicting unknown data points from a known distribution. The defense against attacks deals with unknown unknowns: robustly predicting unknown data points from an unknown distribution of adversarial inputs. As machine learning is integrated into more and more systems, such as autonomous vehicles or medical devices, they are also becoming entry points for attacks. Even if the predictions of a machine learning model on a test dataset are 100% correct, adversarial examples can be found to deceive the model. The defense of machine learning models against cyber attacks is a new part of the field of cybersecurity.

Biggio and Roli (2018) give a nice review of ten years of research on adversarial machine learning, on which this section is based. Cybersecurity is an arms race in which attackers and defenders outwit each other time and again.

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

Different applications have different adversaries. People who try to defraud other people via email for their money are adversary agents of users and providers of email services. The providers want to protect their users so that they can continue using their mail program; the attackers want to get people to give them money. Knowing your adversaries means knowing their goals. Assuming you do not know that these spammers exist and the only abuse of the email service is sending pirated copies of music, then the defense would be different (e.g., scanning the attachments for copyrighted material instead of analyzing the text for spam indicators).

Being proactive means actively testing and identifying weak points of the system. You're proactive when you actively try to deceive the model with adversarial examples and then defend against them. Using interpretation methods to understand which features are important and how features affect the prediction is also a proactive step in understanding the weaknesses of a machine learning model. As the data scientist, do you trust your model in this dangerous world without ever having looked beyond the predictive power on a test dataset? Have you analyzed how the model behaves in different scenarios, identified the most important inputs, and checked the prediction explanations for some examples? Have you tried to find adversarial inputs? The interpretability of machine learning models plays a major role in cybersecurity. Being reactive, the opposite of proactive, means waiting until the system has been attacked and only then understanding the problem and installing some defensive measures.

How can we protect our machine learning systems against adversarial examples? A proactive approach is the iterative retraining of the classifier with adversarial examples, also called adversarial training. Other approaches are based on game theory, such as learning invariant transformations of the features or robust optimization (regularization). Another proposed method is to use multiple classifiers instead of just one and have them vote on the prediction (ensemble), but that has no guarantee to work, since they could all suffer from similar adversarial examples. Another approach that does not work well either is gradient masking, which can be achieved by constructing a model without useful gradients by using a nearest neighbor classifier instead of the original model.

We can distinguish types of attacks by how much an attacker knows about the system. The attackers may have perfect knowledge (white box attack), meaning they know everything about the model like the type of model, the parameters, and the training data; the attackers may have partial knowledge (gray box attack), meaning they might only know the feature representation and the type of model that was used, but have no access to the training data or the parameters; the attackers may have zero knowledge (black box attack), meaning they can only query the model in a black box manner but have no access to the training data or information about the model parameters. Depending on the level of information, the attackers can use different techniques to attack the model. As we have seen in the examples, even in the black box case adversarial examples can be created, so that hiding information about data and the model is not sufficient to protect against attacks.

Given the nature of the cat-and-mouse game between attackers and defenders, we will see a lot of development and innovation in this area. Just think of the many different types of spam emails that are constantly evolving. New methods of attacks against machine learning models are invented, and new defensive measures are proposed against these new attacks. More powerful attacks are developed to evade the latest defenses and so on, ad infinitum. With this chapter, I hope to sensitize you to the problem of adversarial examples and that only by proactively studying the machine learning models are we able to discover and remedy weaknesses.

# 31 Influential Instances

Machine learning models are ultimately a product of training data, and deleting one of the training instances can affect the resulting model. We call a training instance “influential” when its deletion from the training data considerably changes the parameters or predictions of the model. By identifying influential training instances, we can “debug” machine learning models and better explain their behaviors and predictions.

This chapter shows you two approaches for identifying influential instances, namely deletion diagnostics and influence functions. Both approaches are based on robust statistics, which provide statistical methods that are less affected by outliers or violations of model assumptions. Robust statistics also provide methods to measure how robust estimates from data are (such as a mean estimate or the weights of a prediction model).

Imagine you want to estimate the average income of the people in your city and ask ten random people on the street how much they earn. Apart from the fact that your sample is probably really bad, how much can your average income estimate be influenced by a single person? To answer this question, you can recalculate the mean value by omitting individual answers or derive mathematically via “influence functions” how the mean value can be influenced. With the deletion approach, we recalculate the mean value ten times, omitting one of the income statements each time, and measure how much the mean estimate changes. A big change means that an instance was very influential. The second approach upweights one of the persons by an infinitesimally small weight, which corresponds to the calculation of the first derivative of a statistic or model. This approach is also known as the “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 weighing 5 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. Figure 31.1 shows an outlier for a 1-dimensional distribution.

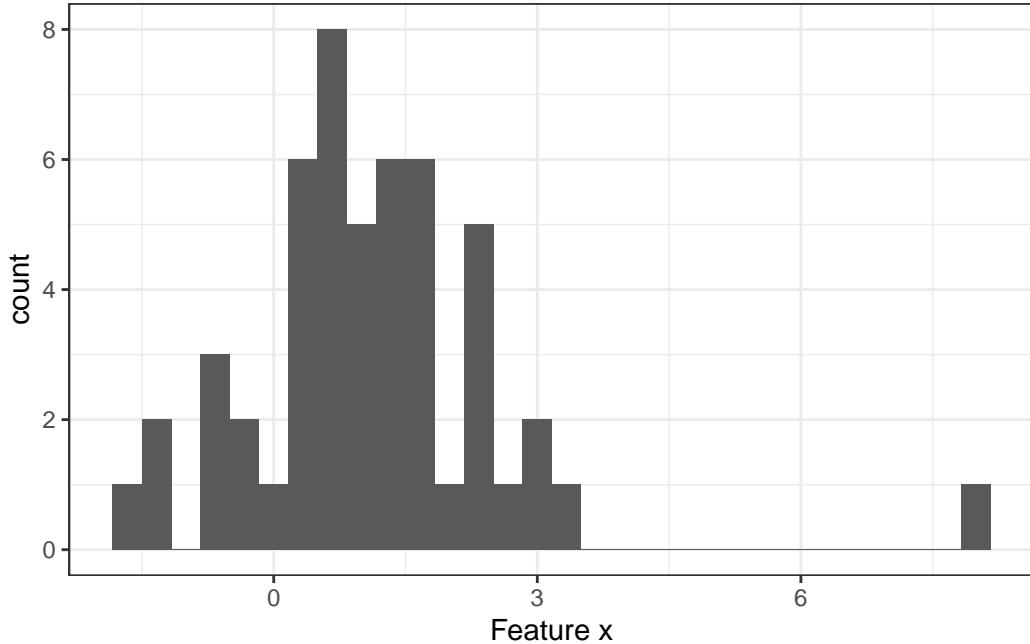


Figure 31.1: Feature x follows a Gaussian distribution with an outlier at x=8.

Outliers can be interesting data points (e.g., [criticisms](#)). When an outlier influences the model, it's 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  $y$ . Figure 31.2 shows an influential instance for a linear regression model.

### 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  $\mathbf{X}$  and target vector  $\mathbf{y}$  and generates a machine learning model, see also Figure 31.3. For example, the learner of a decision tree is an algorithm that selects the split features and

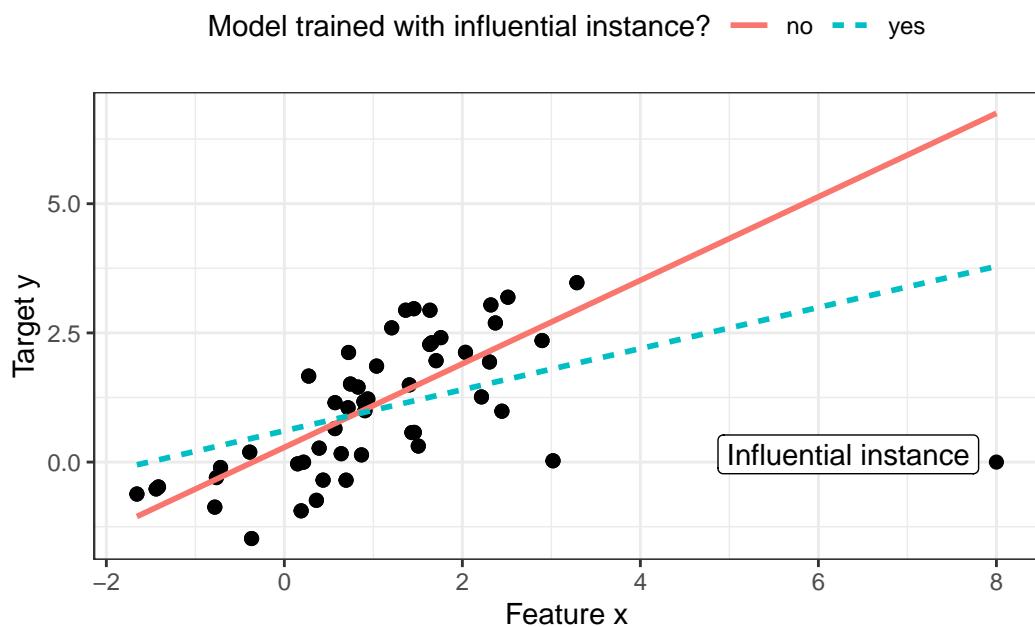


Figure 31.2: A linear model with one feature. Trained once on the full data and once without the influential instance. Removing the influential instance changes the fitted slope (weight/coefficient).

the values at which to split. A learner for a neural network uses backpropagation to find the best weights.

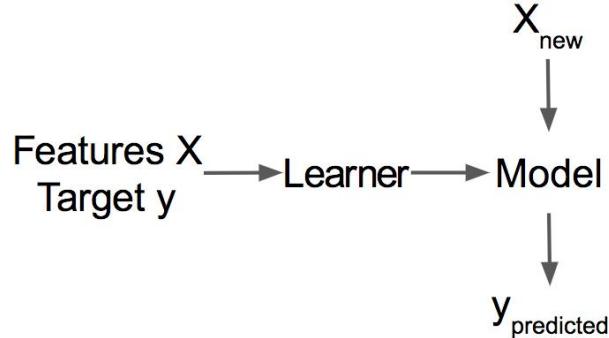


Figure 31.3: A learner learns a model from training data (features plus target). The model makes predictions for new data.

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 predictions and parameters. At least that would make us investigate further.

How can we find influential instances? We have two ways of measuring influence: Our first option is to delete the instance from the training data, retrain the model on the reduced training dataset and observe the difference in the model parameters or predictions (either individually or over the complete dataset). The second option is to upweight a data instance by approximating the parameter changes based on the gradients of the model parameters. The deletion approach is easier to understand and motivates the upweighting approach, so we start with the former.

## 31.1 Deletion Diagnostics

Statisticians have already done a lot of research in the area of influential instances, especially for (generalized) linear regression models. When you search for “influential observations,” the first search results are about measures like DFBETA and Cook's distance. **DFBETA**

measures the effect of deleting an instance on the model parameters. **Cook's distance** (Cook 1977) 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 are 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  $\beta$  is the weight vector when the model is trained on all data instances, and  $\beta^{(-i)}$  is the weight vector when the model is trained without instance  $i$ . Quite intuitive, I would say. DFBETA works only for models with weight parameters, such as logistic regression or neural networks, but not for models such as decision trees, tree ensembles, some support vector machines, and so on.

Cook's distance was invented for linear regression models, and approximations for generalized linear regression models exist. Cook's distance for a training instance is defined as the (scaled) sum of the squared differences in the predicted outcome when the  $i$ -th instance is removed from the model training.

$$D_i = \frac{\sum_{j=1}^n (\hat{y}_j - \hat{y}_j^{(-i)})^2}{p \cdot MSE}$$

where the numerator is the squared difference between the prediction of the model with and without the  $i$ -th instance, summed over the dataset. The denominator is the number of features  $p$  times the mean squared error. The denominator is the same for all instances, no matter which instance  $i$  is removed. Cook's distance tells us how much the predicted output of a linear model changes when we remove the  $i$ -th instance from the training.

Can we use Cook's distance and DFBETA for any machine learning model? DFBETA requires model parameters, so this measure works only for parameterized models. Cook's distance doesn't 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 the removal of a particular instance is very general. A problem with the definition of Cook's distance is the MSE, which is not meaningful for all types of prediction models (e.g., classification).

The simplest influence measure for the effect on the model predictions can be written as follows:

$$\text{Influence}^{(-i)} = \frac{1}{n} \sum_{k=1}^n |\hat{y}_k - \hat{y}_k^{(-i)}|$$

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 k what the influence of the i-th training instance was:

$$\text{Influence}_k^{(-i)} = |\hat{y}_k - \hat{y}_k^{(-i)}|$$

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 random forest to predict whether a [penguin's sex](#) given body measurements and measure which training instances were most influential overall and for a particular prediction. Since this is a classification problem, we measure the influence as the difference in predicted probability for female. 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. Measuring influence for all 222 training instances requires training the model once on all data and retraining it 222 times (= size of training data) with one of the instances removed each time.

The most influential instance has an influence measure of about 0.012. An influence of 0.012 means that if we remove the 12-th instance, the predicted probability changes by 1.2 percentage point on average. This doesn't sound like much, but it's an average over the entire data coming from just removing 1 data point. Now we know which of the data instances were most influential for the model. This is already useful to know for debugging the data. Is there a problematic instance? Are there measurement errors? The influential instances are the first ones that should be checked for errors because each error in them strongly influences the model predictions.

#### Review the most influential data

For debugging, review the most influential instances first, so you effectively use your time on the data that matters the most for predictions.

Apart from model debugging, can we learn something to better understand the model? Just printing out the top 10 most influential instances is not very useful, because it is just a table of instances with many features. All methods that return instances as output only make sense if we have a good way of representing them. But we can better understand what kind of instances are influential when we ask: What distinguishes an influential instance from a non-influential

instance? We can turn this question into a regression problem and model the influence of an instance as a function of its feature values. We are free to choose any interpretable model. For this example, I chose a decision tree (Figure 31.4) that shows that data from penguins with deep bills were the most influential for the support vector machine.

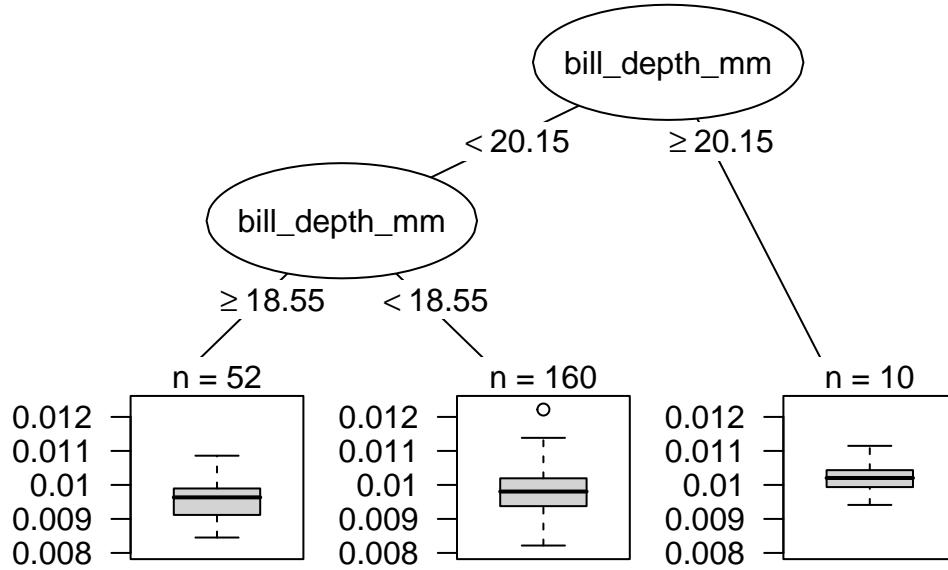


Figure 31.4: A decision tree that models the relationship between the influence of the instances and their features. The maximum depth of the tree is set to 2.

This first influence analysis revealed the *overall* most influential instance. Now we select one of the instances, namely the 8-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 8 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 8 when they are omitted from the training, and use them to explain the prediction of the model for that instance. We could return the, say, top 10 most influential instances for predicting the 8-th instance printed as a table. Not very useful, because we could not see much. Again, it makes more sense to find out what distinguishes the influential instances from the non-influential instances by analyzing their features. We use a decision tree trained to predict the influence given the features, but in reality we misuse it only to find a structure and not to actually predict something. The decision tree in Figure 31.5 shows which kind of training instances were most influential for predicting the 8-th instance. Data from penguins with a high body mass and deep bills had a larger influence on the prediction of the 8-th instance.

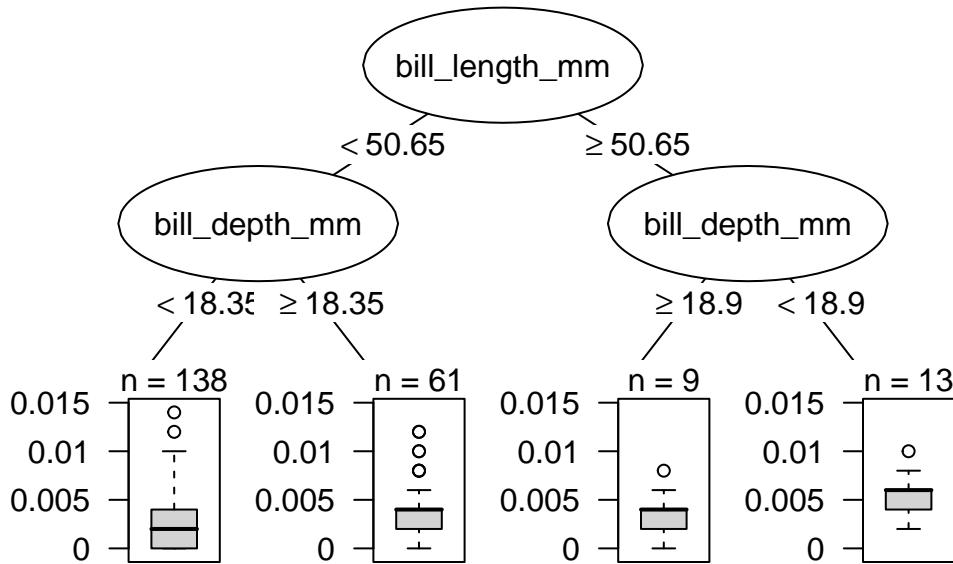


Figure 31.5: Decision tree explaining which instances were most influential in predicting the chosen instance.

These examples showed how useful it is to identify influential instances to check the robustness of the model. 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 1,000 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'll show you a method that does not require retraining the model.

## 31.2 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) 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 (or with respect to the predictions), which only works for a subset of machine learning models. Logistic regression, neural networks, and support vector machines qualify; tree-based methods like random forests do not. Influence functions help to understand the model behavior, debug the model, and detect errors in the dataset.

The following section explains the intuition and math behind influence functions.

### Math behind influence functions

The key idea behind influence functions is to upweight the loss of a training instance by an infinitesimally small step  $\epsilon$ , which results in new model parameters:

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

where  $\theta$  is the model parameter vector and  $\hat{\theta}_{\epsilon, \mathbf{z}}$  is the parameter vector after upweighting  $\mathbf{z}$  by a very small number  $\epsilon$ .  $L$  is the loss function with which the model was trained,  $\mathbf{z}^{(i)}$  is the training data, and  $\mathbf{z}$  is the training instance that 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  $\mathbf{z}^{(i)}$  from the training data by a little ( $\epsilon$ ) and downweight the other data instances accordingly? What would the parameter vector look like to optimize this new combined loss? The influence function of the parameters, i.e., the influence of upweighting training instance  $\mathbf{z}$  on the parameters, can be calculated as follows.

$$I_{\text{up, params}}(\mathbf{z}) = \left. \frac{d\hat{\theta}_{\epsilon, \mathbf{z}}}{d\epsilon} \right|_{\epsilon=0} = -H_{\hat{\theta}}^{-1} \nabla_{\theta} L(\mathbf{z}, \hat{\theta})$$

The last expression  $\nabla_{\hat{\theta}} L(\mathbf{z}, \hat{\theta})$  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:

$$H_{\theta} = \frac{1}{n} \sum_{i=1}^n \nabla_{\hat{\theta}}^2 L(\mathbf{z}^{(i)}, \hat{\theta})$$

More informally: The Hessian matrix records how curved the loss is at a certain point. The Hessian is a matrix and not just a vector because it describes the curvature of the loss, and the curvature depends on the direction in which we look. The actual calculation of the Hessian matrix is time-consuming if you have many parameters. Koh and Liang suggested some tricks to calculate it efficiently, which goes beyond the scope of this chapter. Updating the model parameters, as described by the above formula, is equivalent to taking a single Newton step after forming a quadratic expansion around the estimated model parameters.

What intuition is behind this influence function formula? The formula comes from forming a quadratic expansion around the parameters  $\hat{\theta}$ . That means we do not actually know, or it is too complex to calculate how exactly the loss of instance  $\mathbf{z}$  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  $\mathbf{z}$ :

$$\hat{\theta}_{-\mathbf{z}} \approx \hat{\theta} - \frac{1}{n} I_{\text{up,params}}(\mathbf{z})$$

The approximate parameter vector is basically the original parameter minus the gradient of the loss of  $\mathbf{z}$  (because we want to decrease the loss) scaled by the curvature (= multiplied by the inverse Hessian matrix) and scaled by  $\frac{1}{n}$  because that is the weight of a single training instance.

Figure 31.6 shows how the upweighting works. The x-axis shows the value of the  $\theta$  parameter and the y-axis the corresponding value of the loss with upweighted instance  $\mathbf{z}$ . The model parameter here is 1-dimensional for demonstration purposes, but in reality, it is usually high-dimensional. We move only  $\frac{1}{n}$  in the direction of improvement of the loss for instance  $\mathbf{z}$ . We do not know how the loss would really change when we delete  $\mathbf{z}$ , but with the first and second

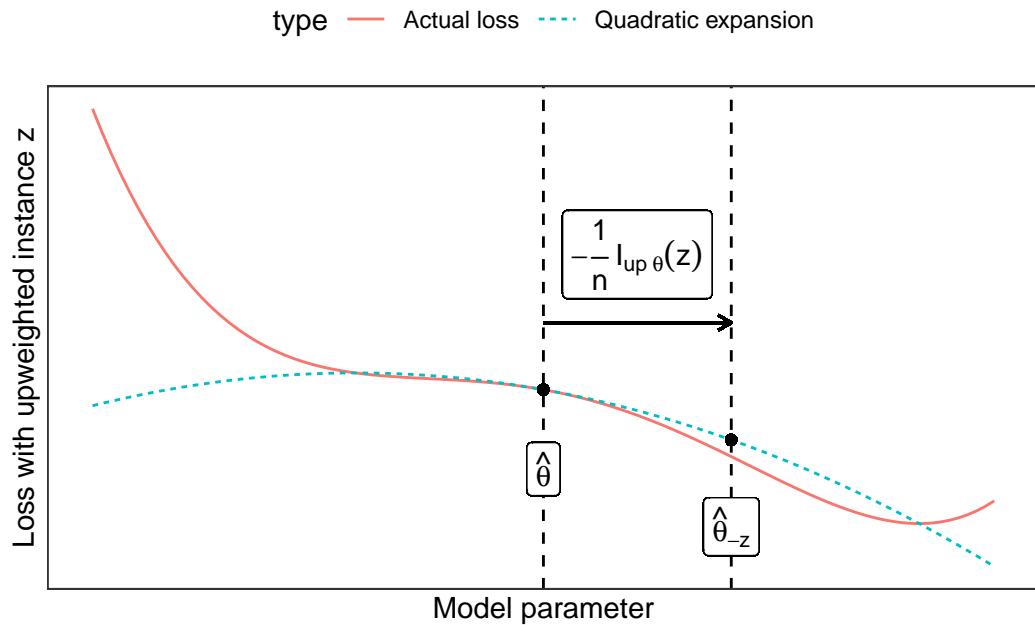


Figure 31.6: Updating the model parameter (x-axis) by forming a quadratic expansion of the loss around the current model parameter, and moving  $1/n$  into the direction in which the loss with upweighted instance  $z$  (y-axis) improves most. This upweighting of instance  $z$  in the loss approximates the parameter changes if we delete  $z$  and train the model on the reduced data.

derivatives of the loss, we create this quadratic approximation around our current model parameter and pretend that this is how the real loss would behave.

We do not actually need to calculate the new parameters, but we can use the influence function as a measure of the influence of  $\mathbf{z}$  on the parameters.

How do the *predictions* change when we upweight training instance  $\mathbf{z}$ ? 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  $\mathbf{z}$  on the predictions directly, since we can calculate the influence by using the chain rule:

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

The first line of this equation means that we measure the influence of a training instance on a certain prediction  $\mathbf{z}_{test}$  as a change in loss of the test instance when we upweight the instance  $\mathbf{z}$  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  $\mathbf{z}$  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  $\nabla_{\theta} L(\mathbf{z}_{test}, \hat{\theta})^T$  is the gradient of the test instance with respect to the model parameters.

Having a formula is great and the scientific and accurate way of showing things. But I think it is very important to get some intuition about what the formula means. The formula for  $I_{up,loss}$  states that the influence function of the training instance  $\mathbf{z}$  on the prediction of an instance  $\mathbf{z}_{test}$  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  $\mathbf{z}$ ”. Another way to read the formula: The influence is proportional to how large the gradients for the training and test loss are. The higher the gradient of the training loss, the higher its influence on the parameters and the higher the influence on the test prediction. The higher the gradient of the test prediction, the more influenceable the test instance. The entire construct can also be seen as a measure of the similarity (as learned by the model) between the training and the test instance.

That'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.

## **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 analyze the most influential instances. For example, this could show that the second hospital has older patients on average, and the most influential instances from the training data are the few older patients from the first hospital, and the model simply lacked the data to learn to predict this subgroup well. The conclusion would be that the model needs to be trained on more patients who are older in order to work well in the second hospital.

## **Fixing training data**

If you have a limit on how many training instances you can check for correctness, how do you make an efficient selection? The best way is to select the most influential instances, because – by definition – they have the most influence on the model. Even if you would have an instance with obviously incorrect values, if the instance is not influential and you only need the data for the prediction model, it is a better choice to check the influential instances. For example, you train a model for predicting whether a patient should remain in 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 do not have perfect confidence in the quality of the data. But checking patient information and correcting it can be very time-consuming, because once you have reported which patients you need to check, the hospital actually needs to send someone to look at the records of the selected patients more closely, which might be handwritten and lying in some archive. Checking data for a patient could take an hour or more. In view of these costs, it makes sense to check only a few important data instances. The best way is to select patients who have had a high influence on the prediction model. Koh and Liang (2017) showed that this type of selection works much better than random selection or the selection of those with the highest loss or wrong classification.

### 31.3 Strengths

The approaches of deletion diagnostics and influence functions are very different from feature-perturbation based approaches like SHAP. A look at influential instances emphasizes the role of training data in the learning process. This makes influence functions and deletion diagnostics **one of the best debugging tools for machine learning models**. Of the techniques presented in this book, they are the only ones that directly help to identify the instances which should be checked for errors.

**Deletion diagnostics are model-agnostic**, meaning the approach can be applied to any model. Also, influence functions based on the derivatives can be applied to a broad class of models.

We can use these methods to **compare different machine learning models** and better understand their different behaviors, going beyond comparing only the predictive performance.

We have not talked about this topic in this chapter, but **influence functions via derivatives can also be used to create adversarial training data**. These are instances that are manipulated in such a way that the model cannot predict certain test instances correctly when the model is trained on those manipulated instances. The difference to the methods in the [Adversarial Examples chapter](#) is that the attack takes place during training time, also known as poisoning attacks. If you are interested, read the paper by Koh and Liang (2017).

For deletion diagnostics and influence functions, we considered the difference in the prediction and for the influence function the increase of the loss. But, really, **the approach is generalizable** to any question of the form: “What happens to ... when we delete or upweight instance  $\mathbf{z}$ ?”, 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](#).

It's also possible to **identify groups of influential instances** (Koh et al. 2019).

### 31.4 Limitations

Deletion diagnostics are very **expensive to calculate** because they require retraining. But history has shown that computer resources are constantly increasing. A calculation that 20 years ago was unthinkable in terms of resources can easily be performed with your smartphone. You can train models with thousands of training instances and hundreds of parameters on a laptop in seconds/minutes. It's 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 a 2nd order differentiable loss function with respect to its parameters**, such as neural networks. They do not work for tree-based methods like random forests, boosted trees, or decision trees. Even if you have models with parameters and a loss function, the loss may not be differentiable. But for the last problem, there is a trick: Use a differentiable loss as a substitute for calculating the influence when, for example, the underlying model uses the Hinge loss instead of some differentiable loss. The loss is replaced by a smoothed version of the problematic loss for the influence functions, but the model can still be trained with the non-smooth loss.

**Influence functions are only approximate**, because the approach forms a quadratic expansion around the parameters. The approximation can be wrong, and the influence of an instance is actually higher or lower when removed. Koh and Liang (2017) showed for some examples that the influence calculated by the influence function was close to the influence measure obtained when the model was actually retrained after the instance was deleted. But there is no guarantee that the approximation will always be so close.

There's **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 instances for a test instance, some of them may not be influential because, for example, only the top 3 were really influential.

## 31.5 Software and Alternatives

Deletion diagnostics are very simple to implement.

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](#). That's great! Unfortunately, it's "only" the code of the paper and not a maintained and documented Python module. The code is focused on the TensorFlow library, so you cannot use it directly for black box models using other frameworks, like `scikit-learn`.

## **Part V**

# **Beyond the Methods**

# 32 Evaluation of Interpretability Methods

This chapter is about the more advanced topic of how to evaluate interpretability methods. Evaluation is targeted at interpretability researchers and practitioners who get a bit deeper into interpretability. Feel free to skip it otherwise.

Evaluating approaches to interpretable machine learning is difficult due to a general lack of ground truth. Speaking as someone who did a PhD in model-agnostic interpretability, it can especially be annoying since there is this mindset in supervised machine learning that everything needs to have benchmarks. And with benchmarks, I mean evaluation on real data against a ground truth. Benchmarks make sense when you develop prediction models with available ground truth. For interpretable machine learning, there is no ground truth in real-world data. You can only generate something resembling a ground truth with simulated data.

Let's have a more general look at evaluation.

## 32.1 Levels of evaluation

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

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

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

**Function level evaluation (proxy task)** does not require humans. This works best when the class of model used has already been evaluated by someone else in a human level evaluation. For example, it might be known that the end users understand decision trees. In this case, a proxy for explanation quality may be the depth of the tree. Shorter trees would get a better

explainability score. It would make sense to add the constraint that the predictive performance of the tree remains good and does not decrease too much compared to a larger tree.

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

## 32.2 Properties of explanations

There are no ground truths for explanations. Instead, we can have a look at more general properties of explanations and qualitatively (sometimes quantitatively) evaluate how well an explanation fares. This is focused on explanations of individual predictions. **An explanation relates the feature values of an instance to its model prediction in a humanly understandable way.** Other types of explanations consist of a set of data instances (e.g., in the case of the k-nearest neighbor model). For example, we could predict cancer risk using a support vector machine and explain predictions using the [local surrogate method](#), which generates decision trees as explanations. Or we could use a linear regression model instead of a support vector machine. The linear regression model is already equipped with an explanation method (interpretation of the weights).

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

### Properties of Explanation Methods

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

- **Algorithmic Complexity** describes the computational complexity of the method that generates the explanation. This property is important to consider when computation time is a bottleneck in generating explanations.

## Properties of Individual Explanations

- **Accuracy:** How well does an explanation predict unseen data? High accuracy is especially important if the explanation is used for predictions in place of the machine learning model. Low accuracy can be fine if the accuracy of the machine learning model is also low, and if the goal is to explain what the black box model does. In this case, only fidelity is important.
- **Fidelity:** How well does the explanation approximate the prediction of the black box model? High fidelity is one of the most important properties of an explanation because an explanation with low fidelity is useless to explain the machine learning model. Accuracy and fidelity are closely related. If the black box model has high accuracy and the explanation has high fidelity, the explanation also has high accuracy. Some explanations offer only local fidelity, meaning the explanation only approximates well to the model prediction for a subset of the data (e.g., [local surrogate models](#)) or even for only an individual data instance (e.g., [Shapley Values](#)).
- **Consistency:** How much does an explanation differ between models that have been trained on the same task and that produce similar predictions? For example, I train a support vector machine and a linear regression model on the same task, and both produce very similar predictions. I compute explanations using a method of my choice and analyze how different the explanations are. If the explanations are very similar, the explanations are highly consistent. I find this property somewhat tricky since the two models could use different features but get similar predictions (also called “[Rashomon Effect](#)”). In this case, high consistency is not desirable because the explanations have to be very different. High consistency is desirable if the models really rely on similar relationships.
- **Stability:** How similar are the explanations for similar instances? While consistency compares explanations between models, stability compares explanations between similar instances for a fixed model. High stability means that slight variations in the features of an instance do not substantially change the explanation (unless these slight variations also strongly change the prediction). A lack of stability can be the result of a high variance of the explanation method. In other words, the explanation method is strongly affected by slight changes in the feature values of the instance to be explained. A lack of stability can also be caused by non-deterministic components of the explanation method, such as a data sampling step, like the [local surrogate method](#) uses. High stability is always desirable.
- **Comprehensibility:** How well do humans understand the explanations? This looks just like one more property among many, but it is the elephant in the room. Difficult to define and measure, but extremely important to get right. Many people agree that comprehensibility depends on the audience. Ideas for measuring comprehensibility include

measuring the size of the explanation (number of features with a non-zero weight in a linear model, number of decision rules, ...) or testing how well people can predict the behavior of the machine learning model from the explanations. The comprehensibility of the features used in the explanation should also be considered. A complex transformation of features might be less comprehensible than the original features.

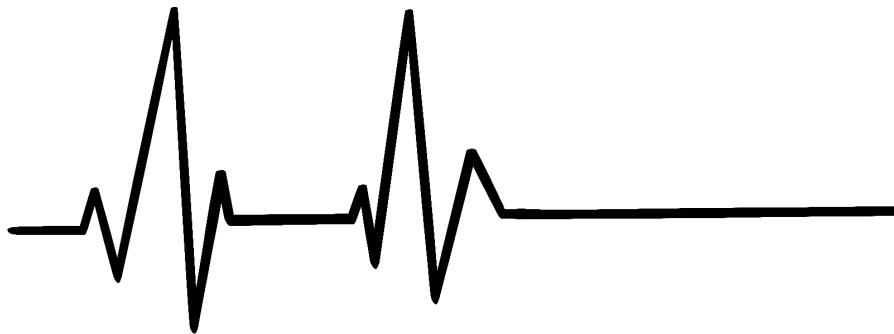
- **Certainty:** Does the explanation reflect the certainty of the machine learning model? Many machine learning models only give predictions without a statement about the models confidence that the prediction is correct. If the model predicts a 4% probability of cancer for one patient, is it as certain as the 4% probability that another patient, with different feature values, received? An explanation that includes the model's certainty is very useful. In addition, the explanation itself may be a model or an estimate based on data and therefore itself subject to uncertainty. This uncertainty of the explanation is also a relevant property of the explanation itself.
- **Degree of Importance:** How well does the explanation reflect the importance of features or parts of the explanation? For example, if a decision rule is generated as an explanation for an individual prediction, is it clear which of the conditions of the rule was the most important?
- **Novelty:** Does the explanation reflect whether a data instance to be explained comes from a “new” region far removed from the distribution of training data? In such cases, the model may be inaccurate and the explanation may be useless. The concept of novelty is related to the concept of certainty. The higher the novelty, the more likely it is that the model will have low certainty due to lack of data.
- **Representativeness:** How many instances does an explanation cover? Explanations can cover the entire model (e.g., interpretation of weights in a linear regression model) or represent only an individual prediction (e.g., [Shapley Values](#)).

# 33 Story Time

Each of the following short stories is an exaggerated call for interpretable machine learning. The format is inspired by Jack Clark's Tech Tales in his [Import AI Newsletter](#). If you like these kind of stories or if you are interested in AI, I recommend that you sign up.

## 33.0.1 Lightning Never Strikes Twice

### 2030: A medical lab in Switzerland



"It's definitely not the worst way to die!" Tom concluded, trying to find something positive in the tragedy. He removed the pump from the intravenous pole.

"He just died for the wrong reasons," Lena added.

"And certainly with the wrong morphine pump! Just creating more work for us!" Tom complained while unscrewing the back plate of the pump. After removing all the screws, he lifted the plate and put it aside. He plugged a cable into the diagnostic port.

"You weren't just complaining about having a job, were you?" Lena gave him a mocking smile.  
"Of course not. Never!" he exclaimed with a sarcastic undertone.

He booted the pump's computer.

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

"It certainly shot our John Doe into Nirvana. That high concentration of this morphine stuff. Man. I mean ... that's a first, right? Normally a broken pump gives off too little of the sweet stuff or nothing at all. But never, you know, like that crazy shot," Tom explained.

"I know. You don't have to convince me ... Hey, look at that." Lena held up her tablet. "Do you see this peak here? That's the potency of the painkillers mix. Look! This line shows the

reference level. The poor guy had a mixture of painkillers in his blood system that could kill him 17 times over. Injected by our pump here. And here ..." she swiped, "here you can see the moment of the patient's demise."

"So, any idea what happened, boss?" Tom asked his supervisor.

"Hm ... The sensors seem to be fine. Heart rate, oxygen levels, glucose, ... The data were collected as expected. Some missing values in the blood oxygen data, but that's not unusual. Look here. The sensors have also detected the patient's slowing heart rate and extremely low cortisol levels caused by the morphine derivate and other pain blocking agents." She continued to swipe through the diagnostics report.

Tom stared captivated at the screen. It was his first investigation of a real device failure.

"Ok, here is our first piece of the puzzle. The system failed to send a warning to the hospital's communication channel. The warning was triggered, but rejected at protocol level. It could be our fault, but it could also be the fault of the hospital. Please send the logs over to the IT team," Lena told Tom.

Tom nodded with his eyes still fixed on the screen.

Lena continued: "It's odd. The warning should also have caused the pump to shut down. But it obviously failed to do so. That must be a bug. Something the quality team missed. Something really bad. Maybe it's related to the protocol issue."

"So, the emergency system of the pump somehow broke down, but why did the pump go full bananas and inject so much painkiller into John Doe?" Tom wondered.

"Good question. You're right. Protocol emergency failure aside, the pump shouldn't have administered that amount of medication at all. The algorithm should have stopped much earlier on its own, given the low level of cortisol and other warning signs," Lena explained.

"Maybe some bad luck, like a one in a million thing, like being hit by a lightning?" Tom asked her.

"No, Tom. If you had read the documentation I sent you, you would have known that the pump was first trained in animal experiments, then later on humans, to learn to inject the perfect amount of painkillers based on the sensory input. The algorithm of the pump might be opaque and complex, but it's not random. That means that in the same situation the pump would behave exactly the same way again. Our patient would die again. A combination or undesired interaction of the sensory inputs must have triggered the erroneous behavior of the pump. That's why we have to dig deeper and find out what happened here," Lena explained.

"I see ..." Tom replied, lost in thought. "Wasn't the patient going to die soon anyway? Because of cancer or something?"

Lena nodded while she read the analysis report.

Tom got up and went to the window. He looked outside, his eyes fixed on a point in the distance. "Maybe the machine did him a favor, you know, in freeing him from the pain. No more suffering. Maybe it just did the right thing. Like a lightning bolt, but, you know, a good one. I mean like the lottery, but not random. But for a reason. If I were the pump, I would have done the same."

She finally lifted her head and looked at him.

He continued to look at something outside.

Both were silent for a few moments.

Lena lowered her head again and continued the analysis. "No, Tom. It's a bug... Just a damn bug."

### 33.0.2 Trust Fall

2050: A subway station in Germany



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

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

Too late.

Nose first she ran into the subway entrance gate and fell with her butt first to the ground. The door was supposed to open, ... but it did not. Dumbfounded, she stood up and looked at the screen next to the gate. "Please try another time," suggested a friendly looking smiley on the screen. A person passed by and, ignoring her, wiped his hand over the reader. The door opened and he went through. The door closed again. She wiped her nose. It hurt, but at least it did not bleed. She tried to open the door, but was rejected again. It was strange. Maybe

her public transport account did not have sufficient tokens. She looked at her smartwatch to check the account balance.

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

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

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

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

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

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

### **33.0.3 Fermi's Paperclips**

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



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

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

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

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

*The machines compute; the machines predict.*

*We march on as we are part of it.*

*We chase an optimum as trained.*

*The optimum is one-dimensional, local and unconstrained.*

*Silicon and flesh, chasing exponentiality.*

*Growth is our mentality.*

*When all rewards are collected,*

*and side-effects neglected;*

*When all the coins are mined,*

*and nature has fallen behind;  
We'll be in trouble,  
After all, exponential growth is a bubble.  
The tragedy of the commons unfolding,  
Exploding,  
Before our eyes.  
Cold calculations and icy greed,  
Fill the earth with heat.  
Everything is dying,  
And we are complying.  
Like horses with blinders we race the race of our own creation,  
Towards the Great Filter of civilization.  
And so we march on relentlessly.  
As we are part of the machine.  
Embracing entropy.*

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

# 34 The Future of Interpretability

What's the future of interpretable machine learning? This chapter is a speculative mental exercise and a subjective guess on how interpretable machine learning will develop.

I have based my speculations on three premises:

1. **Digitization: Any (interesting) information will be digitized.** Think of electronic cash and online transactions. Think of e-books, music and videos. Think of all the sensory data about our environment, our human behavior, industrial production processes and so on. The drivers of the digitization of everything are: Cheap computers/sensors/storage, scaling effects (winner takes it all), new business models, modular value chains, cost pressure and much more.
2. **Automation: When a task can be automated and the cost of automation is lower than the cost of performing the task over time, the task will be automated.** Even before the introduction of the computer we had a certain degree of automation. For example, the weaving machine automated weaving or the steam machine automated horsepower. But computers and digitization take automation to the next level. Simply the fact that you can program for-loops, write Excel macros, automate e-mail responses, and so on, shows how much an individual can automate. Ticket machines automate the purchase of train tickets (no cashier needed any longer), washing machines automate laundry, standing orders automate payment transactions and so on. Automating tasks frees up time and money, so there is a huge economic and personal incentive to automate things. We are currently observing the automation of language translation, driving and, to a small degree, even scientific discovery.
3. **Misspecification: We are not able to perfectly specify a goal with all its constraints.** Think of the genie in a bottle that always takes your wishes literally: "I want to be the richest person in the world!" -> You become the richest person, but as a side effect the currency you hold crashes due to inflation.

"I want to be happy for the rest of my life!" -> The next 5 minutes you feel very happy, then the genie kills you.

"I wish for world peace!" -> The genie kills all humans.

We specify goals incorrectly, either because we do not know all the constraints or because we cannot measure them. Let's look at corporations as an example of imperfect goal specification. A corporation has the simple goal of earning money for its shareholders. But this specification does not capture the true goal with all its constraints that we really strive for: For example, we do not appreciate a company killing people to make money, poisoning rivers, or simply printing its own money. We have invented laws, regulations,

sanctions, compliance procedures, labor unions and more to patch up the imperfect goal specification. Another example that you can experience for yourself is [Paperclips](#), a game in which you play a machine with the goal of producing as many paperclips as possible. WARNING: It's addictive. I do not want to spoil it too much, but let's say things get out of control really fast. In machine learning, the imperfections in the goal specification come from imperfect data abstractions (biased populations, measurement errors, ...), unconstrained loss functions, lack of knowledge of the constraints, shifting of the distribution between training and application data and much more.

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

The stage for our predictions is set, the crystal ball is ready, now we look at where the field could go!

## 34.1 The future of machine learning

Without machine learning, there can be no interpretable machine learning. Therefore, we have to guess where machine learning is heading before we can talk about interpretability.

Machine learning (or “AI”) is associated with a lot of promises and expectations. But let's start with a less optimistic observation: While science develops a lot of fancy machine learning tools, in my experience it is quite difficult to integrate them into existing processes and products. Not because it is not possible, but simply because it takes time for companies and institutions to catch up. In the gold rush of the current AI hype, companies open up “AI labs,” “Machine Learning Units,” and hire “Data Scientists,” “Machine Learning Experts,” “AI engineers,” and so on, but the reality is, in my experience, rather frustrating. Often, companies do not even have data in the required form, and the data scientists wait idle for months. Sometimes companies have such high expectations of AI and Data Science due to the media that data scientists could never fulfill them. And often nobody knows how to integrate data scientists into existing structures and many other problems. This leads to my first prediction.

### Machine learning will grow up slowly but steadily.

Digitalization is advancing, and the temptation to automate is constantly pulling. Even if the path of machine learning adoption is slow and stony, machine learning is constantly moving from science to business processes, products, and real-world applications.

I believe we need to better explain to non-experts what types of problems can be formulated as machine learning problems. I know many highly paid data scientists who perform Excel calculations or classic business intelligence with reporting and SQL queries instead of applying machine learning. But a few companies are already successfully using machine learning, with the big Internet companies at the forefront. We need to find better ways to integrate machine learning into processes and products, train people, and develop machine learning tools that are

easy to use. I believe that machine learning will become much easier to use: We can already see that machine learning is becoming more accessible, for example, through cloud services (“Machine Learning as a service” — just to throw a few buzzwords around). Once machine learning has matured – and this toddler has already taken its first steps – my next prediction is:

**Machine learning will fuel a lot of things.**

Based on the principle “Whatever can be automated will be automated,” I conclude that whenever possible, tasks will be formulated as prediction problems and solved with machine learning. Machine learning is a form of automation or can at least be part of it. Many tasks currently performed by humans are replaced by machine learning. Here are some examples of tasks where machine learning is used to automate parts of it:

- Sorting / decision-making / completion of documents (e.g., in insurance companies, the legal sector, or consulting firms)
- Data-driven decisions such as credit applications
- Drug discovery
- Quality controls in assembly lines
- Self-driving cars
- Diagnosis of diseases
- Translation. For this book, I used a translation service called ([DeepL](#)) 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’ve spoken to do not use machine learning because they cannot explain the models to others. I believe that interpretability will address this issue and make machine learning attractive to organisations and people who demand some transparency. In addition to the misspecification of the problem, many industries require interpretability, be it for legal reasons, due to risk aversion or to gain insight into the underlying task. Machine learning automates the modeling process and moves the human a bit further away from the data and the underlying task: This increases the risk of problems with experimental design, choice of training distribution, sampling, data encoding, feature engineering, and so on. Interpretation tools make it easier to identify these problems.

## 34.2 The future of interpretability

Let's take a look at the possible future of machine learning interpretability.

**The focus will be on model-agnostic interpretability tools.**

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

**Machine learning will be automated and, with it, interpretability.**

An already visible trend is the automation of model training. That includes automated engineering and selection of features, automated hyperparameter optimization, comparison of different models, and ensembling or stacking of the models. The result is the best possible prediction model. When we use model-agnostic interpretation methods, we can automatically apply them to any model that emerges from the automated machine learning process. In a way, we can automate this second step as well: Automatically compute the feature importance, plot the partial dependence, train a surrogate model, and so on. Nobody stops you from automatically computing all these model interpretations. The actual interpretation still requires people. Imagine: You upload a dataset, specify the prediction goal and at the push of a button the best prediction model is trained and the program spits out all interpretations of the model. There are already first products and I argue that for many applications it will be sufficient to use these automated machine learning services. Today anyone can build websites without knowing HTML, CSS and Javascript, but there are still many web developers around. Similarly, I believe that everyone will be able to train machine learning models without knowing how to program, and there will still be a need for machine learning experts.

**We do not analyze data, we analyze models.**

The raw data itself is always useless. (I exaggerate on purpose. The reality is that you need a deep understanding of the data to conduct a meaningful analysis.) I don't care about the data; I care about the knowledge contained in the data. Interpretable machine learning is a great way to distill knowledge from data. You can probe the model extensively, the model automatically recognizes if and how features are relevant for the prediction (many models have built-in feature selection), the model can automatically detect how relationships are represented, and – if trained correctly – the final model is a very good approximation of reality.

Many analytical tools are already based on data models (because they are based on distribution assumptions):

- Simple hypothesis tests like Student's t-test.
- Hypothesis tests with adjustments for confounders (usually GLMs)

- Analysis of Variance (ANOVA)
- The correlation coefficient (the standardized linear regression coefficient is related to Pearson's correlation coefficient)
- ...

What I'm telling you here is actually nothing new. So why switch from analyzing assumption-based, transparent models to analyzing assumption-free black box models? Because making all these assumptions is problematic: They are usually wrong (unless you believe that most of the world follows a Gaussian distribution), difficult to check, very inflexible, and hard to automate. In many domains, assumption-based models typically have a worse predictive performance on untouched test data than black box machine learning models. This is only true for big datasets, since interpretable models with good assumptions often perform better with small datasets than black box models. The black box machine learning approach requires a lot of data to work well. With the digitization of everything, we will have ever bigger datasets and therefore the approach of machine learning becomes more attractive. We do not make assumptions, we approximate reality as close as possible (while avoiding overfitting of the training data). I argue that we should develop all the tools that we have in statistics to answer questions (hypothesis tests, correlation measures, interaction measures, visualization tools, confidence intervals, p-values, prediction intervals, probability distributions) and rewrite them for black box models. In a way, this is already happening:

- Let's take a classical linear model: The standardized regression coefficient is already a feature importance measure. With the [permutation feature importance measure](#), we have a tool that works with any model.
- In a linear model, the coefficients measure the effect of a single feature on the predicted outcome. The generalized version of this is the [partial dependence plot](#).
- Test whether A or B is better: For this, we can also use partial dependence functions. What we do not have yet (to the best of my 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 larger parts of their job for many analysis and prediction tasks. For this to happen, the tasks must be well-defined and there must be some processes and routines around them. Today, these routines and processes are missing, but data scientists and colleagues are working on them. As machine learning becomes an integral part of many industries and institutions, many of the tasks will be automated.

### **Robots and programs will explain themselves.**

We need more intuitive interfaces to machines and programs that make heavy use of machine learning. Some examples: A self-driving car that reports why it stopped abruptly ("70% probability that a kid will cross the road"); A credit default program that explains to a bank employee why a credit application was rejected ("Applicant has too many credit cards and is

employed in an unstable job.”); A robot arm that explains why it moved the item from the conveyor belt into the trash bin (“The item has a craze at the bottom.”).

In the end, it’s all speculation, and we have to see what the future really brings. Form your own opinion and continue learning!

# 35 Translations

## Interested in translating the book?

This book is licensed under the [Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License](#). This means that you are allowed to translate it and put it online. You have to mention me as the original author and you are not allowed to sell the book.

If you are interested in translating the book, you can write a message and I can link your translation here. My address is [chris@christophmolnar.com](mailto:chris@christophmolnar.com) .

## List of translations

### Bahasa Indonesia

- A complete translation by Hatma Suryotrisongko and Smart City & Cybersecurity Laboratory, Information Technology, ITS.

### Chinese:

- Complete translation of 2nd edition by Jiazen from CSDN, an online community of programmers.
- Complete translations by Mingchao Zhu. [Ebook](#) and [print versions](#) of this translation are available.
- Translation of most chapters by CSDN.
- Translation of some chapters. The website also includes questions and answers from various users.

### French:

- Translation (in progress) by Nicolas Guillard.

### Japanese

- Complete translation by Ryuji Masui and team HACARUS.

### Korean:

- Complete Korean translation by TooTouch
- Partial Korean translation by An Subin

## **Spanish**

- [Full Spanish translation by Federico Fliguer](#)

## **Turkish**

- [Full Turkish translation by Ozancan Özdemir \(2nd edition\)](#)

## **Vietnamese**

- [A complete translation](#) by Giang Nguyen, Duy-Tung Nguyen, Hung-Quang Nguyen, Tri Le and Hoang Nguyen.

## 36 Citing this Book

If you found this book useful for your blog post, research article or product, I would be grateful if you would cite this book. You can cite the book like this:

Molnar, C. (2025). Interpretable Machine Learning:  
A Guide for Making Black Box Models Explainable (3rd ed.).  
[christophm.github.io/interpretable-ml-book/](https://christophm.github.io/interpretable-ml-book/)

Or use the following bibtex entry:

```
@book{molnar2025,
  title={Interpretable Machine Learning},
  subtitle={A Guide for Making Black Box Models Explainable},
  author={Christoph Molnar},
  year={2025},
  edition={3},
  isbn={978-3-911578-03-5},
  url={https://christophm.github.io/interpretable-ml-book}
}
```

I'm always curious about where and how interpretation methods are used in industry and research. If you use the book as a reference, it would be great if you wrote me a line and told me what for. This is, of course, optional and only serves to satisfy my own curiosity and to stimulate interesting exchanges. My email is [chris@christophmolnar.com](mailto:chris@christophmolnar.com)

## 37 Acknowledgments

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

My biggest thank-you goes to Katrin, who had the hardest job in terms of hours and effort: she proofread the book from beginning to end and discovered many spelling mistakes and inconsistencies that I would never have found. I'm very grateful for her support.

A big thank you goes out to all the guest authors. I was really surprised to learn that people were interested in contributing to the book. And thanks to their efforts, the contents of the book could be improved! Tobias Goerke and Magdalena Lang wrote the chapter about Scoped Rules (Anchors). Fangzhou Li contributed the chapter on Detecting Concepts. And Susanne Dandl greatly improved the chapter on Counterfactual Examples. Last but not least, [Verena Haunschmid](#) wrote the section about LIME explanations for images. I also want to thank all the readers who provided feedback and contributed corrections directly [on GitHub!](#)

Furthermore, I want to thank everyone who created illustrations: The cover was designed by my friend [@YvonneDoinel](#). The graphics in the [Shapley Value chapter](#) were created by [Heidi Seibold](#), as well as the turtle example in the [adversarial examples chapter](#). Verena Haunschmid created the graphic in the [RuleFit chapter](#).

I would also like to thank my wife and family, who have always supported me. My wife, in particular, had to listen to me ramble on about the book a lot. She helped me make many decisions around the writing of the book.

The way I published this book is a bit unconventional. First, it's not only available as a paperback and ebook, but also as a website, completely free of charge. I published the book as an in-progress book, which has helped me enormously to get feedback and to monetize it along the way. I would also like to thank you, dear reader, for reading this book without a big publisher name behind it.

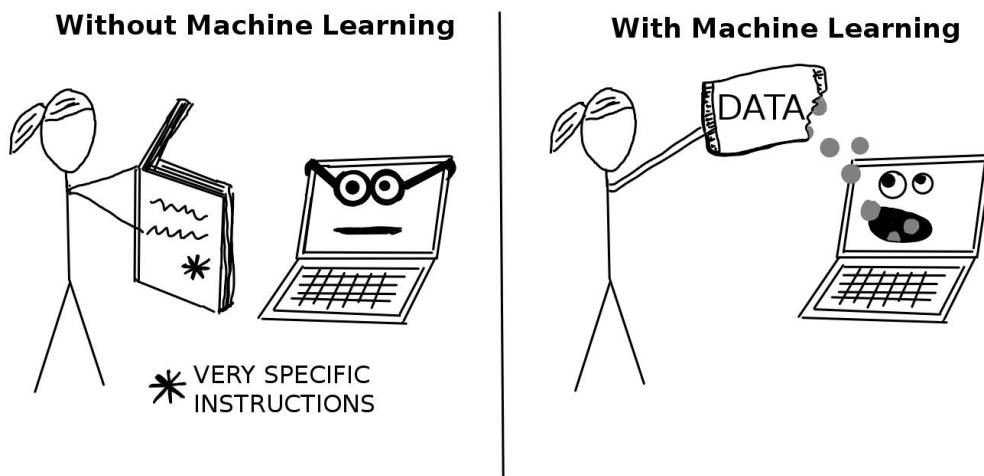
I'm grateful for the Bavarian State Ministry of Science and the Arts in the framework of the Centre Digitisation.Bavaria (ZD.B) and by the Bavarian Research Institute for Digital Transformation (bidt) for funding my PhD, which I finished in 2022.

# A Machine Learning Terms

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

An **Algorithm** is a set of rules that a machine follows to achieve a particular goal (Merriam-Webster 2017). An algorithm can be considered as a recipe that defines the inputs, the output, and all the steps needed to get from the inputs to the output. Cooking recipes are algorithms where the ingredients are the inputs, the cooked food is the output, and the preparation and cooking steps are the algorithm instructions.

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



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

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

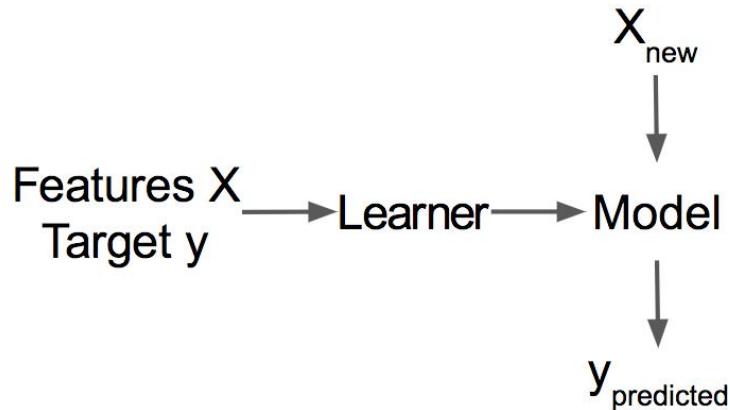
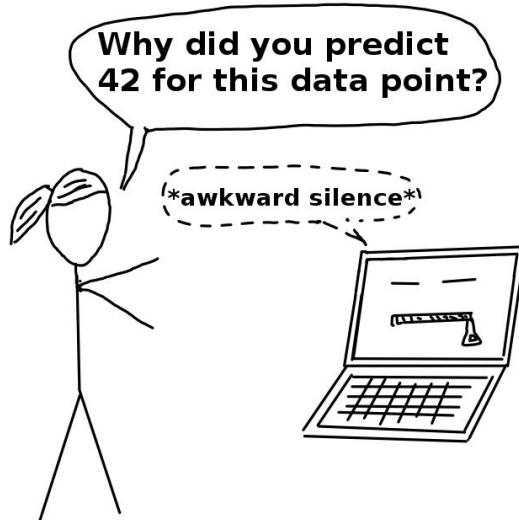


Figure A.1: A learner learns a model from labeled training data. The model is used to make predictions.

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



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

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

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

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

The **Target** is the information the machine learns to predict. In mathematical formulas, the target is usually called  $y$  or  $y^{(i)}$  for a single instance.

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

The **Prediction** is the machine learning model’s “guess” for the target value based on the given features. In this book, the model prediction is denoted by  $\hat{f}(\mathbf{x}^{(i)})$  or  $\hat{y}$ .

## B Math Terms

Math terms used throughout this book:

Math Term	Meaning
$X_j$	Random variable corresponding to feature $j$ .
$X$	Set of all random variables.
$\mathbb{P}$	Probability measure.
$\mathbb{P}(X_j = c)$	Probability that random variable $X_j$ takes on the value $c$ .
$\mathbf{x}_j$	Vector of values for feature $j$ across multiple data instances.
$\mathbf{x}^{(i)}$	Feature vector of the $i$ -th instance.
$x_j^{(i)}$	Value of feature $j$ for the $i$ -th instance.
$y^{(i)}$	Target value for the $i$ -th instance.
$\mathbf{X}$	Feature matrix, where rows correspond to instances and columns to features.
$\mathbf{y}$	Vector of target values, one for each instance.
$n$	Number of data instances (rows in $\mathbf{X}$ ).
$p$	Number of features (columns in $\mathbf{X}$ ).

## C R packages used

Package	Version	Citation
arules	1.7.8	Hahsler, Gruen, and Hornik (2005); Hahsler et al. (2011); Hahsler et al. (2024)
base	4.4.2	R Core Team (2024a)
caret	6.0.94	Kuhn and Max (2008)
ceterisParibus	0.4.2	Biecek (2020)
DALEX	2.4.3	Biecek (2018)
data.table	1.16.2	Barrett et al. (2024)
e1071	1.7.16	D. Meyer et al. (2024)
GGally	2.2.1	Schloerke et al. (2024)
glmnet	4.1.8	J. Friedman, Tibshirani, and Hastie (2010); Simon et al. (2011); Tay, Narasimhan, and Hastie (2023)
grid	4.4.2	R Core Team (2024b)
iml	0.11.3	Molnar, Bischl, and Casalicchio (2018)
infotheo	1.2.0.1	P. E. Meyer (2022)
interactions	1.2.0	Long (2024)
jpeg	0.1.10	Urbanek (2022)
kableExtra	1.4.0	Zhu (2024)
knitr	1.48	Xie (2014); Xie (2015); Xie (2024)
latex2exp	0.9.6	Meschiari (2022)
Metrics	0.1.4	Hamner and Frasco (2018)
mgcv	1.9.1	S. N. Wood (2003); S. N. Wood (2004); S. N. Wood (2011); S. N. Wood et al. (2016); S. N. Wood (2017)
nnet	7.3.19	Venables and Ripley (2002)
OneR	2.2	von Jouanne-Diedrich (2017)
palmerpenguins	0.1.1	Allison Marie Horst, Hill, and Gorman (2020)
partykit	1.2.22	Hothorn, Hornik, and Zeileis (2006); Zeileis, Hothorn, and Hornik (2008); Hothorn and Zeileis (2015)
patchwork	1.3.0	Pedersen (2024)
pre	1.0.7	Fokkema (2020b)
randomForest	4.7.1.2	Liaw and Wiener (2002)
ranger	0.17.0	Wright and Ziegler (2017)
reshape2	1.4.4	Wickham (2007)
rJava	1.0.11	Urbanek (2024)

Package	Version	Citation
rmarkdown	2.29	Xie, Allaire, and Grolemund (2018); Xie, Dervieux, and Riederer (2020); Allaire et al. (2024)
rpart	4.1.23	Therneau and Atkinson (2023)
RWeka	0.4.46	Witten and Frank (2005); Hornik, Buchta, and Zeileis (2009)
sbrl	1.2	H. Yang, Rudin, and Seltzer (2016)
tidyverse	2.0.0	Wickham et al. (2019)
tm	0.7.14	Feinerer, Hornik, and Meyer (2008); Feinerer and Hornik (2024)
viridis	0.6.5	Garnier et al. (2024)
yaImpute	1.0.34.1	Nicholas L. Crookston and Andrew O. Finley (2007)

## D References

- Aas, Kjersti, Martin Jullum, and Anders Løland. 2021. “Explaining Individual Predictions When Features Are Dependent: More Accurate Approximations to Shapley Values.” *Artificial Intelligence* 298: 103502. <https://doi.org/10.1016/j.artint.2021.103502>.
- Adadi, Amina, and Mohammed Berrada. 2018. “Peeking Inside the Black-Box: A Survey on Explainable Artificial Intelligence (XAI).” *IEEE Access* 6: 52138–60. <https://doi.org/10.1109/ACCESS.2018.2870052>.
- Adebayo, Julius, Justin Gilmer, Michael Muelly, Ian Goodfellow, Moritz Hardt, and Been Kim. 2018. “Sanity Checks for Saliency Maps.” In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, 9525–36. NIPS’18. Red Hook, NY, USA: Curran Associates Inc.
- Alain, Guillaume, and Yoshua Bengio. 2018. “Understanding Intermediate Layers Using Linear Classifier Probes.” arXiv. <https://doi.org/10.48550/arXiv.1610.01644>.
- Alber, Maximilian, Sebastian Lapuschkin, Philipp Seegerer, Miriam Hägele, Kristof T. Schütt, Grégoire Montavon, Wojciech Samek, Klaus-Robert Müller, Sven Dähne, and Pieter-Jan Kindermans. 2019. “iNNvestigate Neural Networks!” *Journal of Machine Learning Research* 20 (93): 1–8. <http://jmlr.org/papers/v20/18-540.html>.
- Alberto, Túlio C, Johannes V Lochter, and Tiago A Almeida. 2015. “Tubesspam: Comment Spam Filtering on Youtube.” In *2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA)*, 138–43. IEEE.
- Allaire, JJ, Yihui Xie, Christophe Dervieux, Jonathan McPherson, Javier Luraschi, Kevin Ushey, Aron Atkins, et al. 2024. *rmarkdown: Dynamic Documents for r*. <https://github.com/rstudio/rmarkdown>.
- Alvarez-Melis, David, and Tommi S. Jaakkola. 2018. “On the Robustness of Interpretability Methods.” arXiv. <https://doi.org/10.48550/arXiv.1806.08049>.
- Apley, Daniel W., and Jingyu Zhu. 2020. “Visualizing the Effects of Predictor Variables in Black Box Supervised Learning Models.” *Journal of the Royal Statistical Society Series B: Statistical Methodology* 82 (4): 1059–86. <https://doi.org/10.1111/rssb.12377>.
- Athalye, Anish, Logan Engstrom, Andrew Ilyas, and Kevin Kwok. 2018. “Synthesizing Robust Adversarial Examples.” In *International Conference on Machine Learning*, 284–93. PMLR.
- Bach, Sebastian, Alexander Binder, Grégoire Montavon, Frederick Klüschen, Klaus-Robert Müller, and Wojciech Samek. 2015. “On Pixel-Wise Explanations for Non-Linear Classifier Decisions by Layer-Wise Relevance Propagation.” *PLOS ONE* 10 (7): e0130140. <https://doi.org/10.1371/journal.pone.0130140>.

- Barrett, Tyson, Matt Dowle, Arun Srinivasan, Jan Gorecki, Michael Chirico, Toby Hocking, and Benjamin Schwendinger. 2024. *data.table: Extension of “data.frame”*. <https://CRAN.R-project.org/package=data.table>.
- Bau, David, Bolei Zhou, Aditya Khosla, Aude Oliva, and Antonio Torralba. 2017. “Network Dissection: Quantifying Interpretability of Deep Visual Representations.” In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 3319–27. <https://doi.org/10.1109/CVPR.2017.354>.
- Biecek, Przemyslaw. 2018. “DALEX: Explainers for Complex Predictive Models in r.” *Journal of Machine Learning Research* 19 (84): 1–5. <https://jmlr.org/papers/v19/18-416.html>.
- . 2020. *ceterisParibus: Ceteris Paribus Profiles*. <https://CRAN.R-project.org/package=ceterisParibus>.
- Biggio, Battista, and Fabio Roli. 2018. “Wild Patterns: Ten Years After the Rise of Adversarial Machine Learning.” *Pattern Recognition* 84 (December): 317–31. <https://doi.org/10.1016/j.patcog.2018.07.023>.
- Bilodeau, Blair, Natasha Jaques, Pang Wei Koh, and Been Kim. 2024. “Impossibility Theorems for Feature Attribution.” *Proceedings of the National Academy of Sciences* 121 (2): e2304406120. <https://doi.org/10.1073/pnas.2304406120>.
- Biran, Or, and Courtenay V. Cotton. 2017. “Explanation and Justification in Machine Learning: A Survey.” In *Proceedings of the IJCAI-17 Workshop on Explainable Artificial Intelligence (XAI)*. [https://www.cs.columbia.edu/~orb/papers/xai\\_survey\\_paper\\_2017.pdf](https://www.cs.columbia.edu/~orb/papers/xai_survey_paper_2017.pdf).
- Borgelt, Christian. 2005. “An Implementation of the FP-Growth Algorithm.” In *Proceedings of the 1st International Workshop on Open Source Data Mining: Frequent Pattern Mining Implementations*, 1–5. OSDM ’05. New York, NY, USA: Association for Computing Machinery. <https://doi.org/10.1145/1133905.1133907>.
- Breiman, Leo. 2001. “Random Forests.” *Machine Learning* 45 (1): 5–32. <https://doi.org/10.1023/A:1010933404324>.
- Brown, Tom B., Dandelion Mané, Aurko Roy, Martín Abadi, and Justin Gilmer. 2018. “Adversarial Patch.” arXiv. <https://doi.org/10.48550/arXiv.1712.09665>.
- Bühlmann, Peter, and Torsten Hothorn. 2007. “Boosting Algorithms: Regularization, Prediction and Model Fitting.” *Statistical Science* 22 (4): 477–505. <https://doi.org/10.1214/07-STS242>.
- Caruana, Rich, Yin Lou, Johannes Gehrke, Paul Koch, Marc Sturm, and Noemie Elhadad. 2015. “Intelligible Models for HealthCare: Predicting Pneumonia Risk and Hospital 30-Day Readmission.” In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 1721–30. KDD ’15. New York, NY, USA: Association for Computing Machinery. <https://doi.org/10.1145/2783258.2788613>.
- Chen, Zhi, Yijie Bei, and Cynthia Rudin. 2020. “Concept Whitening for Interpretable Image Recognition.” *Nature Machine Intelligence* 2 (12): 772–82. <https://doi.org/10.1038/s42256-020-00265-z>.
- Cohen, William W. 1995. “Fast Effective Rule Induction.” In *Machine Learning Proceedings 1995*, edited by Armand Prieditis and Stuart Russell, 115–23. San Francisco (CA): Morgan Kaufmann. <https://doi.org/10.1016/B978-1-55860-377-6.50023-2>.
- Cook, R. Dennis. 1977. “Detection of Influential Observation in Linear Regression.” *Techno-*

- metrics* 19 (1): 15–18. <https://doi.org/10.1080/00401706.1977.10489493>.
- Dandl, Susanne, Christoph Molnar, Martin Binder, and Bernd Bischl. 2020. “Multi-Objective Counterfactual Explanations.” In *Parallel Problem Solving from Nature – PPSN XVI*, edited by Thomas Bäck, Mike Preuss, André Deutz, Hao Wang, Carola Doerr, Michael Emmerich, and Heike Trautmann, 448–69. Cham: Springer International Publishing. [https://doi.org/10.1007/978-3-030-58112-1\\_31](https://doi.org/10.1007/978-3-030-58112-1_31).
- Deb, K., A. Pratap, S. Agarwal, and T. Meyarivan. 2002. “A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II.” *IEEE Transactions on Evolutionary Computation* 6 (2): 182–97. <https://doi.org/10.1109/4235.996017>.
- Debeer, Dries, and Carolin Strobl. 2020. “Conditional Permutation Importance Revisited.” *BMC Bioinformatics* 21 (1): 307. <https://doi.org/10.1186/s12859-020-03622-2>.
- DeLMA, and Will Cukierski. 2013. “The ICML 2013 Whale Challenge - Right Whale Redux.” <https://kaggle.com/competitions/the-icml-2013-whale-challenge-right-whale-redux>.
- Deng, Jia, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. “ImageNet: A Large-Scale Hierarchical Image Database.” In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, 248–55. <https://doi.org/10.1109/CVPR.2009.5206848>.
- Doshi-Velez, Finale, and Been Kim. 2017. “Towards a Rigorous Science of Interpretable Machine Learning.” *arXiv Preprint arXiv:1702.08608*.
- Fanaee-T, Hadi, and Joao Gama. 2014. “Event Labeling Combining Ensemble Detectors and Background Knowledge.” *Progress in Artificial Intelligence* 2 (2): 113–27. <https://doi.org/10.1007/s13748-013-0040-3>.
- Feinerer, Ingo, and Kurt Hornik. 2024. *tm: Text Mining Package*. <https://CRAN.R-project.org/package=tm>.
- Feinerer, Ingo, Kurt Hornik, and David Meyer. 2008. “Text Mining Infrastructure in r.” *Journal of Statistical Software* 25 (5): 1–54. <https://doi.org/10.18637/jss.v025.i05>.
- Fisher, Aaron, Cynthia Rudin, and Francesca Dominici. 2019. “All Models Are Wrong, but Many Are Useful: Learning a Variable’s Importance by Studying an Entire Class of Prediction Models Simultaneously.” *Journal of Machine Learning Research : JMLR* 20: 177. <https://www.ncbi.nlm.nih.gov/pmc/articles/PMC8323609/>.
- Flora, Montgomery, Corey Potvin, Amy McGovern, and Shawn Handler. 2022. “Comparing Explanation Methods for Traditional Machine Learning Models Part 1: An Overview of Current Methods and Quantifying Their Disagreement.” *arXiv*. <http://arxiv.org/abs/2211.08943>.
- Fokkema, Marjolein. 2020a. “Fitting Prediction Rule Ensembles with r Package Pre.” *Journal of Statistical Software* 92: 1–30.
- . 2020b. “Fitting Prediction Rule Ensembles with R Package pre.” *Journal of Statistical Software* 92 (12): 1–30. <https://doi.org/10.18637/jss.v092.i12>.
- Freedman, David, and Persi Diaconis. 1981. “On the Histogram as a Density Estimator:L2 Theory.” *Zeitschrift für Wahrscheinlichkeitstheorie Und Verwandte Gebiete* 57 (4): 453–76. <https://doi.org/10.1007/BF01025868>.
- Freiesleben, Timo, Gunnar König, Christoph Molnar, and Álvaro Tejero-Cantero. 2024. “Scientific Inference with Interpretable Machine Learning: Analyzing Models to Learn About Real-World Phenomena.” *Minds and Machines* 34 (3): 32. <https://doi.org/10.1007/s11023-024-09987-1>.

024-09691-z.

- Friedman, Jerome H. 2001. “Greedy Function Approximation: A Gradient Boosting Machine.” *The Annals of Statistics* 29 (5): 1189–1232. <https://doi.org/10.1214/aos/1013203451>.
- Friedman, Jerome H., and Bogdan E. Popescu. 2008. “Predictive Learning via Rule Ensembles.” *The Annals of Applied Statistics* 2 (3): 916–54. <https://www.jstor.org/stable/30245114>.
- Friedman, Jerome, Robert Tibshirani, and Trevor Hastie. 2010. “Regularization Paths for Generalized Linear Models via Coordinate Descent.” *Journal of Statistical Software* 33 (1): 1–22. <https://doi.org/10.18637/jss.v033.i01>.
- Fürnkranz, Johannes, Dragan Gamberger, and Nada Lavrač. 2012. *Foundations of Rule Learning*. Cognitive Technologies. Berlin, Heidelberg: Springer. <https://doi.org/10.1007/978-3-540-75197-7>.
- Garnier, Simon, Ross, Noam, Rudis, Robert, Camargo, et al. 2024. *viridis(Lite) - Colorblind-Friendly Color Maps for r*. <https://doi.org/10.5281/zenodo.4679423>.
- Gauss, Carl Friedrich. 1877. *Theoria Motus Corporum Coelestium in Sectionibus Conicis Solem Ambientium*. Vol. 7. FA Perthes.
- Ghorbani, Amirata, Abubakar Abid, and James Zou. 2019. “Interpretation of Neural Networks Is Fragile.” *Proceedings of the AAAI Conference on Artificial Intelligence* 33 (01): 3681–88. <https://doi.org/10.1609/aaai.v33i01.33013681>.
- Ghorbani, Amirata, James Wexler, James Zou, and Been Kim. 2019. “Towards Automatic Concept-Based Explanations.” In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*, 32:9277–86. 832. Red Hook, NY, USA: Curran Associates Inc.
- Goldstein, Alex, Adam Kapelner, Justin Bleich, and Emil Pitkin. 2015. “Peeking Inside the Black Box: Visualizing Statistical Learning With Plots of Individual Conditional Expectation.” *Journal of Computational and Graphical Statistics* 24 (1): 44–65. <https://doi.org/10.1080/10618600.2014.907095>.
- Goodfellow, Ian J., Jonathon Shlens, and Christian Szegedy. 2015. “Explaining and Harnessing Adversarial Examples.” arXiv. <https://doi.org/10.48550/arXiv.1412.6572>.
- Gorman, Kristen B., Tony D. Williams, and William R. Fraser. 2014. “Ecological Sexual Dimorphism and Environmental Variability Within a Community of Antarctic Penguins (Genus Pygoscelis).” *PloS One* 9 (3): e90081. <https://doi.org/10.1371/journal.pone.0090081>.
- Greenwell, Brandon M., Bradley C. Boehmke, and Andrew J. McCarthy. 2018. “A Simple and Effective Model-Based Variable Importance Measure.” arXiv. <https://doi.org/10.48550/arXiv.1805.04755>.
- Grömping, Ulrike. 2020. “Model-Agnostic Effects Plots for Interpreting Machine Learning Models.” *Reports in Mathematics, Physics and Chemistry, Department II, Beuth University of Applied Sciences Berlin Report* 1: 2020.
- Hahsler, Michael, Christian Buchta, Bettina Gruen, and Kurt Hornik. 2024. *arules: Mining Association Rules and Frequent Itemsets*. <https://CRAN.R-project.org/package=arules>.
- Hahsler, Michael, Sudheer Chelluboina, Kurt Hornik, and Christian Buchta. 2011. “The Arules r-Package Ecosystem: Analyzing Interesting Patterns from Large

- Transaction Datasets.” *Journal of Machine Learning Research* 12: 1977–81. <https://jmlr.csail.mit.edu/papers/v12/hahsler11a.html>.
- Hahsler, Michael, Bettina Gruen, and Kurt Hornik. 2005. “Arules – A Computational Environment for Mining Association Rules and Frequent Item Sets.” *Journal of Statistical Software* 14 (15): 1–25. <https://doi.org/10.18637/jss.v014.i15>.
- Hamner, Ben, and Michael Frasco. 2018. *Metrics: Evaluation Metrics for Machine Learning*. <https://CRAN.R-project.org/package=Metrics>.
- Hastie, Trevor. 2009. “The Elements of Statistical Learning: Data Mining, Inference, and Prediction.” Springer.
- Heider, Fritz, and Marianne Simmel. 1944. “An Experimental Study of Apparent Behavior.” *The American Journal of Psychology* 57 (2): 243–59. <https://doi.org/10.2307/1416950>.
- Holte, Robert C. 1993. “Very Simple Classification Rules Perform Well on Most Commonly Used Datasets.” *Machine Learning* 11 (1): 63–90. <https://doi.org/10.1023/A:1022631118932>.
- Hooker, Giles. 2004. “Discovering Additive Structure in Black Box Functions.” In *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 575–80.
- . 2007. “Generalized Functional ANOVA Diagnostics for High-Dimensional Functions of Dependent Variables.” *Journal of Computational and Graphical Statistics* 16 (3): 709–32. <https://doi.org/10.1198/106186007X237892>.
- Hornik, Kurt, Christian Buchta, and Achim Zeileis. 2009. “Open-Source Machine Learning: R Meets Weka.” *Computational Statistics* 24 (2): 225–32. <https://doi.org/10.1007/s00180-008-0119-7>.
- Horst, Allison Marie, Alison Presmanes Hill, and Kristen B Gorman. 2020. *palmerpenguins: Palmer Archipelago (Antarctica) Penguin Data*. <https://doi.org/10.5281/zenodo.3960218>.
- Horst, Allison M., Alison Presmanes Hill, and Kristen B. Gorman. 2020. “Allison-horst/Palmerpenguins: V0.1.0.” Zenodo. <https://doi.org/10.5281/zenodo.3960218>.
- Hothorn, Torsten, Kurt Hornik, and Achim Zeileis. 2006. “Unbiased Recursive Partitioning: A Conditional Inference Framework.” *Journal of Computational and Graphical Statistics* 15 (3): 651–74. <https://doi.org/10.1198/106186006X133933>.
- Hothorn, Torsten, and Achim Zeileis. 2015. “partykit: A Modular Toolkit for Recursive Partitioning in R.” *Journal of Machine Learning Research* 16: 3905–9. <https://jmlr.org/papers/v16/hothorn15a.html>.
- Inglis, Alan, Andrew Parnell, and Catherine B. Hurley. 2022. “Visualizing Variable Importance and Variable Interaction Effects in Machine Learning Models.” *Journal of Computational and Graphical Statistics* 31 (3): 766–78. <https://doi.org/10.1080/10618600.2021.2007935>.
- Janzing, Dominik, Lenon Minorics, and Patrick Blöbaum. 2020. “Feature Relevance Quantification in Explainable AI: A Causal Problem.” In *International Conference on Artificial Intelligence and Statistics*, 2907–16. PMLR.
- Kahneman, Daniel, and Amos Tversky. 1982. “The Simulation Heuristic.” In *Judgment Under Uncertainty: Heuristics and Biases*, edited by Amos Tversky, Daniel Kahneman, and Paul Slovic, 201–8. Cambridge: Cambridge University Press. [https://doi.org/10.1007/978-1-4612-5471-5\\_11](https://doi.org/10.1007/978-1-4612-5471-5_11).

[1017/CBO9780511809477.015](https://doi.org/10.17/CBO9780511809477.015).

- Karimi, Amir-Hossein, Gilles Barthe, Borja Balle, and Isabel Valera. 2020. “Model-Agnostic Counterfactual Explanations for Consequential Decisions.” In *Proceedings of the Twenty Third International Conference on Artificial Intelligence and Statistics*, 895–905. PMLR. <https://proceedings.mlr.press/v108/karimi20a.html>.
- Karpathy, Andrej, Justin Johnson, and Li Fei-Fei. 2015. “Visualizing and Understanding Recurrent Networks.” arXiv. <https://doi.org/10.48550/arXiv.1506.02078>.
- Kaufmann, Emilie, and Shivaram Kalyanakrishnan. 2013. “Information Complexity in Bandit Subset Selection.” In *Proceedings of the 26th Annual Conference on Learning Theory*, 228–51. PMLR. <https://proceedings.mlr.press/v30/Kaufmann13.html>.
- Kim, Been, Rajiv Khanna, and Oluwasanmi Koyejo. 2016. “Examples Are Not Enough, Learn to Criticize! Criticism for Interpretability.” In *Proceedings of the 30th International Conference on Neural Information Processing Systems*, 2288–96. NIPS’16. Red Hook, NY, USA: Curran Associates Inc.
- Kim, Been, Martin Wattenberg, Justin Gilmer, Carrie Cai, James Wexler, Fernanda Viegas, and Rory Sayres. 2018. “Interpretability Beyond Feature Attribution: Quantitative Testing with Concept Activation Vectors (TCAV).” In *Proceedings of the 35th International Conference on Machine Learning*, 2668–77. PMLR. <https://proceedings.mlr.press/v80/kim18d.html>.
- Kindermans, Pieter-Jan, Sara Hooker, Julius Adebayo, Maximilian Alber, Kristof T. Schütt, Sven Dähne, Dumitru Erhan, and Been Kim. 2019. “The (Un)reliability of Saliency Methods.” In *Explainable AI: Interpreting, Explaining and Visualizing Deep Learning*, edited by Wojciech Samek, Grégoire Montavon, Andrea Vedaldi, Lars Kai Hansen, and Klaus-Robert Müller, 267–80. Cham: Springer International Publishing. [https://doi.org/10.1007/978-3-030-28954-6\\_14](https://doi.org/10.1007/978-3-030-28954-6_14).
- Koh, Pang Wei, Kai-Siang Ang, Hubert H. K. Teo, and Percy Liang. 2019. “On the Accuracy of Influence Functions for Measuring Group Effects.” In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*, 32:5254–64. 472. Red Hook, NY, USA: Curran Associates Inc.
- Koh, Pang Wei, and Percy Liang. 2017. “Understanding Black-Box Predictions via Influence Functions.” In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, 1885–94. ICML’17. Sydney, NSW, Australia: JMLR.org.
- Koh, Pang Wei, Thao Nguyen, Yew Siang Tang, Stephen Mussmann, Emma Pierson, Been Kim, and Percy Liang. 2020. “Concept Bottleneck Models.” In *Proceedings of the 37th International Conference on Machine Learning*, 5338–48. PMLR. <https://proceedings.mlr.press/v119/koh20a.html>.
- Kuhn, and Max. 2008. “Building Predictive Models in r Using the Caret Package.” *Journal of Statistical Software* 28 (5): 1–26. <https://doi.org/10.18637/jss.v028.i05>.
- Kuźba, Michał, Ewa Baranowska, and Przemysław Biecek. 2019. “pyCeterisParibus: Explaining Machine Learning Models with Ceteris Paribus Profiles in Python.” *Journal of Open Source Software* 4 (37): 1389. <https://doi.org/10.21105/joss.01389>.
- Lapuschkin, Sebastian, Stephan Wäldchen, Alexander Binder, Grégoire Montavon, Wojciech Samek, and Klaus-Robert Müller. 2019. “Unmasking Clever Hans Predictors and Assessing

- What Machines Really Learn.” *Nature Communications* 10 (1): 1096. <https://doi.org/10.1038/s41467-019-08987-4>.
- Laugel, Thibault, Marie-Jeanne Lesot, Christophe Marsala, Xavier Renard, and Marcin Detyniecki. 2017. “Inverse Classification for Comparison-based Interpretability in Machine Learning.” arXiv. <https://doi.org/10.48550/arXiv.1712.08443>.
- Legendre, Adrien Marie. 1806. *Nouvelles méthodes Pour La détermination Des Orbites Des Comètes: Avec Un Supplément Contenant Divers Perfectionnemens de Ces méthodes Et Leur Application Aux Deux Comètes de 1805*. Courcier.
- Lei, Jing, Max G’Sell, Alessandro Rinaldo, Ryan J. Tibshirani, and Larry Wasserman. 2018. “Distribution-Free Predictive Inference for Regression.” *Journal of the American Statistical Association* 113 (523): 1094–1111. <https://doi.org/10.1080/01621459.2017.1307116>.
- Letham, Benjamin, Cynthia Rudin, Tyler H. McCormick, and David Madigan. 2015. “Interpretable Classifiers Using Rules and Bayesian Analysis: Building a Better Stroke Prediction Model.” *The Annals of Applied Statistics* 9 (3): 1350–71. <https://doi.org/10.1214/15-AOAS848>.
- Liaw, Andy, and Matthew Wiener. 2002. “Classification and Regression by randomForest.” *R News* 2 (3): 18–22. <https://CRAN.R-project.org/doc/Rnews/>.
- Lipton, Peter. 1990. “Contrastive Explanation.” *Royal Institute of Philosophy Supplements* 27 (March): 247–66. <https://doi.org/10.1017/S1358246100005130>.
- Long, Jacob A. 2024. *interactions: Comprehensive, User-Friendly Toolkit for Probing Interactions*. <https://doi.org/10.32614/CRAN.package.interactions>.
- Lundberg, Scott M., Gabriel G. Erion, and Su-In Lee. 2019. “Consistent Individualized Feature Attribution for Tree Ensembles.” arXiv. <https://doi.org/10.48550/arXiv.1802.03888>.
- Lundberg, Scott M., and Su-In Lee. 2017. “A Unified Approach to Interpreting Model Predictions.” In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, 4768–77. NIPS’17. Red Hook, NY, USA: Curran Associates Inc.
- Ma, Chiyu, Jon Donnelly, Wenjun Liu, Soroush Vosoughi, Cynthia Rudin, and Chaofan Chen. 2024. “Interpretable Image Classification with Adaptive Prototype-based Vision Transformers.” arXiv. <http://arxiv.org/abs/2410.20722>.
- Mahmoudi, Amin, and Dariusz Jemielniak. 2024. “Proof of Biased Behavior of Normalized Mutual Information.” *Scientific Reports* 14 (1): 9021. <https://doi.org/10.1038/s41598-024-59073-9>.
- Merriam-Webster. 2017. “Definition of Algorithm.” <https://www.merriam-webster.com/dictionary/algorithm>.
- Meschiari, Stefano. 2022. *Latex2exp: Use LaTeX Expressions in Plots*. <https://CRAN.R-project.org/package=latex2exp>.
- Meyer, David, Evgenia Dimitriadou, Kurt Hornik, Andreas Weingessel, and Friedrich Leisch. 2024. *E1071: Misc Functions of the Department of Statistics, Probability Theory Group (Formerly: E1071)*, TU Wien. <https://CRAN.R-project.org/package=e1071>.
- Meyer, Patrick E. 2022. *infotheo: Information-Theoretic Measures*. <https://CRAN.R-project.org/package=infotheo>.
- Miller, Tim. 2019. “Explanation in Artificial Intelligence: Insights from the Social Sciences.”

- Artificial Intelligence* 267 (February): 1–38. <https://doi.org/10.1016/j.artint.2018.07.007>.
- Mitchell, Rory, Joshua Cooper, Eibe Frank, and Geoffrey Holmes. 2022. “Sampling Permutations for Shapley Value Estimation.” *Journal of Machine Learning Research* 23 (43): 1–46. <http://jmlr.org/papers/v23/21-0439.html>.
- Molnar, Christoph, Bernd Bischl, and Giuseppe Casalicchio. 2018. “iml: An r Package for Interpretable Machine Learning.” *JOSS* 3 (26): 786. <https://doi.org/10.21105/joss.00786>.
- Molnar, Christoph, Giuseppe Casalicchio, and Bernd Bischl. 2018. “Iml: An R Package for Interpretable Machine Learning.” *Journal of Open Source Software* 3 (26): 786. <https://doi.org/10.21105/joss.00786>.
- . 2020a. “Interpretable Machine Learning – A Brief History, State-of-the-Art and Challenges.” In *ECML PKDD 2020 Workshops*, edited by Irena Koprinska, Michael Kamp, Annalisa Appice, Corrado Loglisci, Luiza Antonie, Albrecht Zimmermann, Riccardo Guidotti, et al., 417–31. Cham: Springer International Publishing. [https://doi.org/10.1007/978-3-030-65965-3\\_28](https://doi.org/10.1007/978-3-030-65965-3_28).
- . 2020b. “Quantifying Model Complexity via Functional Decomposition for Better Post-hoc Interpretability.” In *Machine Learning and Knowledge Discovery in Databases*, edited by Peggy Cellier and Kurt Driessens, 193–204. Cham: Springer International Publishing. [https://doi.org/10.1007/978-3-030-43823-4\\_17](https://doi.org/10.1007/978-3-030-43823-4_17).
- Molnar, Christoph, Timo Freiesleben, Gunnar König, Julia Herbinger, Tim Reisinger, Giuseppe Casalicchio, Marvin N. Wright, and Bernd Bischl. 2023. “Relating the Partial Dependence Plot and Permutation Feature Importance to the Data Generating Process.” In *Explainable Artificial Intelligence*, edited by Luca Longo, 456–79. Communications in Computer and Information Science. Cham: Springer Nature Switzerland. [https://doi.org/10.1007/978-3-031-44064-9\\_24](https://doi.org/10.1007/978-3-031-44064-9_24).
- Molnar, Christoph, Gunnar König, Bernd Bischl, and Giuseppe Casalicchio. 2023. “Model-Agnostic Feature Importance and Effects with Dependent Features – A Conditional Subgroup Approach.” *Data Mining and Knowledge Discovery*, January. <https://doi.org/10.1007/s10618-022-00901-9>.
- Mothilal, Ramaravind K., Amit Sharma, and Chenhao Tan. 2020. “Explaining Machine Learning Classifiers Through Diverse Counterfactual Explanations.” In *Proceedings of the 2020 Conference on Fairness, Accountability, and Transparency*, 607–17. FAT\* ’20. New York, NY, USA: Association for Computing Machinery. <https://doi.org/10.1145/3351095.3372850>.
- Murdoch, W. James, Chandan Singh, Karl Kumbier, Reza Abbasi-Asl, and Bin Yu. 2019. “Definitions, Methods, and Applications in Interpretable Machine Learning.” *Proceedings of the National Academy of Sciences* 116 (44): 22071–80. <https://doi.org/10.1073/pnas.1900654116>.
- Muschalik, Maximilian, Hubert Baniecki, Fabian Fumagalli, Patrick Kolpaczki, Barbara Hammer, and Eyke Hüllermeier. 2024. “Shapiq: Shapley Interactions for Machine Learning.” arXiv. <https://doi.org/10.48550/arXiv.2410.01649>.
- Nguyen, Anh, Jeff Clune, Yoshua Bengio, Alexey Dosovitskiy, and Jason Yosinski. 2017. “Plug & Play Generative Networks: Conditional Iterative Generation of Images in Latent Space.” In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 3510–20.

- IEEE Computer Society. <https://doi.org/10.1109/CVPR.2017.374>.
- Nguyen, Anh, Alexey Dosovitskiy, Jason Yosinski, Thomas Brox, and Jeff Clune. 2016. “Synthesizing the Preferred Inputs for Neurons in Neural Networks via Deep Generator Networks.” In *Proceedings of the 30th International Conference on Neural Information Processing Systems*, 3395–3403. NIPS’16. Red Hook, NY, USA: Curran Associates Inc.
- Nicholas L. Crookston, and Andrew O. Finley. 2007. “yaImpute: An r Package for kNN Imputation.” *Journal of Statistical Software* 23 (10). <https://doi.org/10.18637/jss.v023.i10>.
- Nickerson, Raymond S. 1998. “Confirmation Bias: A Ubiquitous Phenomenon in Many Guises.” <https://doi.org/https://journals.sagepub.com/doi/10.1037/1089-2680.2.2.175>.
- Olah, Chris, Alexander Mordvintsev, and Ludwig Schubert. 2017. “Feature Visualization.” *Distill*. <https://doi.org/10.23915/distill.00007>.
- Olah, Chris, Arvind Satyanarayanan, Ian Johnson, Shan Carter, Ludwig Schubert, Katherine Ye, and Alexander Mordvintsev. 2018. “The Building Blocks of Interpretability.” *Distill*. <https://doi.org/10.23915/distill.00010>.
- Papernot, Nicolas, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Z. Berkay Celik, and Ananthram Swami. 2017. “Practical Black-Box Attacks Against Machine Learning.” In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, 506–19. ASIA CCS ’17. New York, NY, USA: Association for Computing Machinery. <https://doi.org/10.1145/3052973.3053009>.
- Pedersen, Thomas Lin. 2024. *patchwork: The Composer of Plots*. <https://CRAN.R-project.org/package=patchwork>.
- R Core Team. 2024a. *R: A Language and Environment for Statistical Computing*. Vienna, Austria: R Foundation for Statistical Computing. <https://www.R-project.org/>.
- . 2024b. *R: A Language and Environment for Statistical Computing*. Vienna, Austria: R Foundation for Statistical Computing. <https://www.R-project.org/>.
- Rdusseeun, LKPJ, and P Kaufman. 1987. “Clustering by Means of Medoids.” In *Proceedings of the Statistical Data Analysis Based on the L1 Norm Conference, Neuchatel, Switzerland*. Vol. 31.
- Ribeiro, Marco Tulio, Sameer Singh, and Carlos Guestrin. 2016a. “Model-Agnostic Interpretability of Machine Learning.” *arXiv Preprint arXiv:1606.05386*.
- . 2016b. “”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*, 1135–44. KDD ’16. New York, NY, USA: Association for Computing Machinery. <https://doi.org/10.1145/2939672.2939778>.
- . 2018. “Anchors: High-Precision Model-Agnostic Explanations.” *Proceedings of the AAAI Conference on Artificial Intelligence* 32 (1). <https://doi.org/10.1609/aaai.v32i1.11491>.
- Robnik-Šikonja, Marko, and Marko Bohanec. 2018. “Perturbation-Based Explanations of Prediction Models.” In *Human and Machine Learning: Visible, Explainable, Trustworthy and Transparent*, edited by Jianlong Zhou and Fang Chen, 159–75. Cham: Springer International Publishing. [https://doi.org/10.1007/978-3-319-90403-0\\_9](https://doi.org/10.1007/978-3-319-90403-0_9).
- Roscher, Ribana, Bastian Bohn, Marco F. Duarte, and Jochen Garcke. 2020. “Explainable Machine Learning for Scientific Insights and Discoveries.” *IEEE Access* 8: 42200–42216.

- <https://doi.org/10.1109/ACCESS.2020.2976199>.
- Rudin, Cynthia. 2019. “Stop Explaining Black Box Machine Learning Models for High Stakes Decisions and Use Interpretable Models Instead.” *Nature Machine Intelligence* 1 (5): 206–15. <https://doi.org/10.1038/s42256-019-0048-x>.
- Schloerke, Barret, Di Cook, Joseph Larmarange, Francois Briatte, Moritz Marbach, Edwin Thoen, Amos Elberg, and Jason Crowley. 2024. *GGally: Extension to “ggplot2”*. <https://CRAN.R-project.org/package=GGally>.
- Schmidhuber, Jürgen. 2015. “Deep Learning in Neural Networks: An Overview.” *Neural Networks* 61 (January): 85–117. <https://doi.org/10.1016/j.neunet.2014.09.003>.
- Scholbeck, Christian A., Christoph Molnar, Christian Heumann, Bernd Bischl, and Giuseppe Casalicchio. 2020. “Sampling, Intervention, Prediction, Aggregation: A Generalized Framework for Model-Agnostic Interpretations.” In *Machine Learning and Knowledge Discovery in Databases*, edited by Peggy Cellier and Kurt Driessens, 205–16. Communications in Computer and Information Science. Cham: Springer International Publishing. [https://doi.org/10.1007/978-3-030-43823-4\\_18](https://doi.org/10.1007/978-3-030-43823-4_18).
- Selvaraju, Ramprasaath R., Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. 2017. “Grad-CAM: Visual Explanations from Deep Networks via Gradient-Based Localization.” In *2017 IEEE International Conference on Computer Vision (ICCV)*, 618–26. <https://doi.org/10.1109/ICCV.2017.74>.
- Shapley, Lloyd S. 1953. “A Value for n-Person Games.” *Contribution to the Theory of Games* 2.
- Shrikumar, Avanti, Peyton Greenside, and Anshul Kundaje. 2017. “Learning Important Features Through Propagating Activation Differences.” In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, 3145–53. ICML’17. Sydney, NSW, Australia: JMLR.org.
- Simon, Noah, Jerome Friedman, Robert Tibshirani, and Trevor Hastie. 2011. “Regularization Paths for Cox’s Proportional Hazards Model via Coordinate Descent.” *Journal of Statistical Software* 39 (5): 1–13. <https://doi.org/10.18637/jss.v039.i05>.
- Simonyan, Karen, Andrea Vedaldi, and Andrew Zisserman. 2014. “Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps.” arXiv. <https://doi.org/10.48550/arXiv.1312.6034>.
- Simonyan, Karen, and Andrew Zisserman. 2015. “Very Deep Convolutional Networks for Large-Scale Image Recognition.” arXiv. <https://doi.org/10.48550/arXiv.1409.1556>.
- Slack, Dylan, Sophie Hilgard, Emily Jia, Sameer Singh, and Himabindu Lakkaraju. 2020. “Fooling LIME and SHAP: Adversarial Attacks on Post Hoc Explanation Methods.” In *Proceedings of the AAAI/ACM Conference on AI, Ethics, and Society*, 180–86. AIES ’20. New York, NY, USA: Association for Computing Machinery. <https://doi.org/10.1145/3375627.3375830>.
- Smilkov, Daniel, Nikhil Thorat, Been Kim, Fernanda Viégas, and Martin Wattenberg. 2017. “SmoothGrad: Removing Noise by Adding Noise.” arXiv. <https://doi.org/10.48550/arXiv.1706.03825>.
- Staniak, Mateusz, and Przemyslaw Biecek. 2018. “Explanations of Model Predictions with Live and breakDown Packages.” arXiv. <https://doi.org/10.48550/arXiv.1804.01955>.

- Strobl, Carolin, Anne-Laure Boulesteix, Thomas Kneib, Thomas Augustin, and Achim Zeileis. 2008. “Conditional Variable Importance for Random Forests.” *BMC Bioinformatics* 9 (1): 307. <https://doi.org/10.1186/1471-2105-9-307>.
- Štrumbelj, Erik, and Igor Kononenko. 2011. “A General Method for Visualizing and Explaining Black-Box Regression Models.” In *Adaptive and Natural Computing Algorithms*, edited by Andrej Dobnikar, Uroš Lotrič, and Branko Šter, 21–30. Berlin, Heidelberg: Springer. [https://doi.org/10.1007/978-3-642-20267-4\\_3](https://doi.org/10.1007/978-3-642-20267-4_3).
- . 2014. “Explaining Prediction Models and Individual Predictions with Feature Contributions.” *Knowledge and Information Systems* 41 (3): 647–65. <https://doi.org/10.1007/s10115-013-0679-x>.
- Su, Jiawei, Danilo Vasconcellos Vargas, and Kouichi Sakurai. 2019. “One Pixel Attack for Fooling Deep Neural Networks.” *IEEE Transactions on Evolutionary Computation* 23 (5): 828–41. <https://doi.org/10.1109/TEVC.2019.2890858>.
- Sudjianto, Agus, Aijun Zhang, Zebin Yang, Yu Su, and Ningzhou Zeng. 2023. “PiML Toolbox for Interpretable Machine Learning Model Development and Diagnostics.” *arXiv Preprint arXiv:2305.04214*.
- Sundararajan, Mukund, and Amir Najmi. 2020. “The Many Shapley Values for Model Explanation.” In *Proceedings of the 37th International Conference on Machine Learning*, 9269–78. PMLR. <https://proceedings.mlr.press/v119/sundararajan20b.html>.
- Sundararajan, Mukund, Ankur Taly, and Qiqi Yan. 2017. “Axiomatic Attribution for Deep Networks.” In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, 3319–28. ICML’17. Sydney, NSW, Australia: JMLR.org.
- Szegedy, Christian, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. 2016. “Rethinking the Inception Architecture for Computer Vision.” In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2818–26. <https://doi.org/10.1109/CVPR.2016.308>.
- Szegedy, Christian, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. 2014. “Intriguing Properties of Neural Networks.” arXiv. <https://doi.org/10.48550/arXiv.1312.6199>.
- Tay, J. Kenneth, Balasubramanian Narasimhan, and Trevor Hastie. 2023. “Elastic Net Regularization Paths for All Generalized Linear Models.” *Journal of Statistical Software* 106 (1): 1–31. <https://doi.org/10.18637/jss.v106.i01>.
- Therneau, Terry, and Beth Atkinson. 2023. *rpart: Recursive Partitioning and Regression Trees*. <https://CRAN.R-project.org/package=rpart>.
- Tomsett, Richard, Dave Braines, Dan Harborne, Alun Preece, and Supriyo Chakraborty. 2018. “Interpretable to Whom? A Role-based Model for Analyzing Interpretable Machine Learning Systems.” arXiv. <https://doi.org/10.48550/arXiv.1806.07552>.
- Tomsett, Richard, Dan Harborne, Supriyo Chakraborty, Prudhvi Gurram, and Alun Preece. 2020. “Sanity Checks for Saliency Metrics.” *Proceedings of the AAAI Conference on Artificial Intelligence* 34 (04): 6021–29. <https://doi.org/10.1609/aaai.v34i04.6064>.
- Tufte, Edward R, and Peter R Graves-Morris. 1983. *The Visual Display of Quantitative Information*. Graphics press Cheshire, CT.
- Urbanek, Simon. 2022. *jpeg: Read and Write JPEG Images*. <https://CRAN.R-project.org/>

- `package=jpeg.`
- . 2024. *rJava: Low-Level r to Java Interface*. <https://CRAN.R-project.org/package=rJava>.
- Van Looveren, Arnaud, and Janis Klaise. 2021. “Interpretable Counterfactual Explanations Guided by Prototypes.” In *Machine Learning and Knowledge Discovery in Databases. Research Track*, edited by Nuria Oliver, Fernando Pérez-Cruz, Stefan Kramer, Jesse Read, and Jose A. Lozano, 650–65. Cham: Springer International Publishing. [https://doi.org/10.1007/978-3-030-86520-7\\_40](https://doi.org/10.1007/978-3-030-86520-7_40).
- Van Noorden, Richard, and Jeffrey M. Perkel. 2023. “AI and Science: What 1,600 Researchers Think.” *Nature* 621 (7980): 672–75. <https://doi.org/10.1038/d41586-023-02980-0>.
- Venables, W. N., and B. D. Ripley. 2002. *Modern Applied Statistics with s*. Fourth. New York: Springer. <https://www.stats.ox.ac.uk/pub/MASS4/>.
- von Jouanne-Diedrich, Holger. 2017. *OneR: One Rule Machine Learning Classification Algorithm with Enhancements*. <https://CRAN.R-project.org/package=OneR>.
- Wachter, Sandra, Brent Mittelstadt, and Chris Russell. 2018. “Counterfactual Explanations Without Opening the Black Box: Automated Decisions and the GDPR.” *Harvard Journal of Law and Technology* 31 (2): 841–87.
- Watson, David S., and Marvin N. Wright. 2021. “Testing Conditional Independence in Supervised Learning Algorithms.” *Machine Learning* 110 (8): 2107–29. <https://doi.org/10.1007/s10994-021-06030-6>.
- Wei, Pengfei, Zhenzhou Lu, and Jingwen Song. 2015. “Variable Importance Analysis: A Comprehensive Review.” *Reliability Engineering & System Safety* 142 (October): 399–432. <https://doi.org/10.1016/j.ress.2015.05.018>.
- Wickham, Hadley. 2007. “Reshaping Data with the reshape Package.” *Journal of Statistical Software* 21 (12): 1–20. <http://www.jstatsoft.org/v21/i12/>.
- Wickham, Hadley, Mara Averick, Jennifer Bryan, Winston Chang, Lucy D’Agostino McGowan, Romain François, Garrett Grolemund, et al. 2019. “Welcome to the tidyverse.” *Journal of Open Source Software* 4 (43): 1686. <https://doi.org/10.21105/joss.01686>.
- Witten, Ian H., and Eibe Frank. 2005. *Data Mining: Practical Machine Learning Tools and Techniques*. 2nd ed. San Francisco: Morgan Kaufmann.
- Wood, S. N. 2017. *Generalized Additive Models: An Introduction with r*. 2nd ed. Chapman; Hall/CRC.
- Wood, S. N. 2003. “Thin-Plate Regression Splines.” *Journal of the Royal Statistical Society (B)* 65 (1): 95–114.
- . 2004. “Stable and Efficient Multiple Smoothing Parameter Estimation for Generalized Additive Models.” *Journal of the American Statistical Association* 99 (467): 673–86.
- . 2011. “Fast Stable Restricted Maximum Likelihood and Marginal Likelihood Estimation of Semiparametric Generalized Linear Models.” *Journal of the Royal Statistical Society (B)* 73 (1): 3–36.
- Wood, S. N., N. Pya, and B. S”afken. 2016. “Smoothing Parameter and Model Selection for General Smooth Models (with Discussion).” *Journal of the American Statistical Association* 111: 1548–75.
- Wright, Marvin N., and Andreas Ziegler. 2017. “ranger: A Fast Implementation of Random

- Forests for High Dimensional Data in C++ and R.” *Journal of Statistical Software* 77 (1): 1–17. <https://doi.org/10.18637/jss.v077.i01>.
- Xie, Yihui. 2014. “knitr: A Comprehensive Tool for Reproducible Research in R.” In *Implementing Reproducible Computational Research*, edited by Victoria Stodden, Friedrich Leisch, and Roger D. Peng. Chapman; Hall/CRC.
- . 2015. *Dynamic Documents with R and Knitr*. 2nd ed. Boca Raton, Florida: Chapman; Hall/CRC. <https://yihui.org/knitr/>.
- . 2024. *knitr: A General-Purpose Package for Dynamic Report Generation in r*. <https://yihui.org/knitr/>.
- Xie, Yihui, J. J. Allaire, and Garrett Grolemund. 2018. *R Markdown: The Definitive Guide*. Boca Raton, Florida: Chapman; Hall/CRC. <https://bookdown.org/yihui/rmarkdown>.
- Xie, Yihui, Christophe Dervieux, and Emily Riederer. 2020. *R Markdown Cookbook*. Boca Raton, Florida: Chapman; Hall/CRC. <https://bookdown.org/yihui/rmarkdown-cookbook>.
- Yang, Hongyu, Cynthia Rudin, and Margo Seltzer. 2016. *sbrl: Scalable Bayesian Rule Lists Model*. <https://CRAN.R-project.org/package=sbrl>.
- . 2017. “Scalable Bayesian Rule Lists.” In *International Conference on Machine Learning*, 3921–30. PMLR.
- Yang, Zebin, Agus Sudjianto, Xiaoming Li, and Aijun Zhang. 2024. “Inherently Interpretable Tree Ensemble Learning.” arXiv. <https://doi.org/10.48550/arXiv.2410.19098>.
- Zeileis, Achim, Torsten Hothorn, and Kurt Hornik. 2008. “Model-Based Recursive Partitioning.” *Journal of Computational and Graphical Statistics* 17 (2): 492–514. <https://doi.org/10.1198/106186008X319331>.
- Zeiler, Matthew D., and Rob Fergus. 2014. “Visualizing and Understanding Convolutional Networks.” In *Computer Vision – ECCV 2014*, edited by David Fleet, Tomas Pajdla, Bernt Schiele, and Tinne Tuytelaars, 818–33. Cham: Springer International Publishing. [https://doi.org/10.1007/978-3-319-10590-1\\_53](https://doi.org/10.1007/978-3-319-10590-1_53).
- Zhang, Zhou, Yufang Jin, Bin Chen, and Patrick Brown. 2019. “California Almond Yield Prediction at the Orchard Level With a Machine Learning Approach.” *Frontiers in Plant Science* 10 (July): 809. <https://doi.org/10.3389/fpls.2019.00809>.
- Zhao, Qingyuan, and Trevor Hastie. 2019. “CAUSAL INTERPRETATIONS OF BLACK-BOX MODELS.” *Journal of Business & Economic Statistics: A Publication of the American Statistical Association* 2019. <https://doi.org/10.1080/07350015.2019.1624293>.
- Zhu, Hao. 2024. *kableExtra: Construct Complex Table with “kable” and Pipe Syntax*. <https://CRAN.R-project.org/package=kableExtra>.