Interpreting Machine Learning Models With SHAP

A Guide With Python Examples And Theory On Shapley Values

Christoph Molnar

Content

1 Preface

In my first book, "Interpretable Machine Learning," I overlooked the inclusion of SHAP. I conducted a Twitter survey to determine the most frequently used methods for interpreting machine learning models. Options included LIME, permutation feature importance, partial dependence plots, and "Other." SHAP was not an option.

To my surprise, the majority of respondents selected "Other," with many comments highlighting the absence of SHAP. Although I was aware of SHAP at that time, I underestimated its popularity in machine learning explainability.

This popularity was a double-edged sword. My PhD research on interpretable machine learning was centered around partial dependence plots and permutation feature importance. On multiple occasions, when submitting a paper to a conference, we were advised to focus on SHAP or LIME instead. This advice was misguided because we should make progress for all interpretation methods, not just SHAP, but it underscores the popularity of SHAP.

SHAP has been subjected to its fair share of criticism: it's costly to compute, challenging to interpret, and overhyped. I agree with some of these criticisms. In the realm of interpretable machine learning, there's no perfect method; we must learn to work within constraints, which this book also addresses. However, SHAP excels in many areas: it can work with any model, it's modular in building global interpretations, and it has a vast ecosystem of SHAP adaptations.

As you can see, my relationship with SHAP is a mix of admiration and frustration – perhaps a balanced standpoint for writing about SHAP. I don't intend to overhype it, but I believe it's a beneficial tool worth understanding.

2 Introduction

Machine learning models are powerful tools, but their lack of interpretability is a challenge. It's often unclear why a certain prediction was made, what the most important features were, and how the features influenced the predictions in general. Many people argue that as long as a machine learning model performs well, interpretability is unnecessary. However, there are many practical reasons why you need interpretability, ranging from debugging to building trust in your model.

> **Interpretability**
>
> I think of "interpretability" in the context of machine learning as a keyword. Under this keyword, you find a colorful variety of approaches that attempt to extract information about how the model makes predictions.

2.1 Interpreting to debug

Interpretability is valuable for model debugging, as illustrated by a study predicting pneumonia (Caruana et al. 2015). The authors trained a rule-based model, which learned that if a patient has asthma, they have a lower risk of pneumonia. Seriously? I'm no doctor, but that seems off. Asthma patients typically have a higher risk of lung-related diseases. It appears the model got it all wrong. However, it turns out that asthma patients in this dataset were less likely to get pneumonia. The indirect reason was that these patients received "aggressive care," such as early antibiotic treatment. Consequently, they were less likely to develop pneumonia. A typical case of "correlation does not imply causation" as you can see in Figure 2.1.

The problematic dependence on the asthma feature was only discovered due to the model's interpretability. Imagine if this scenario involved a neural network.

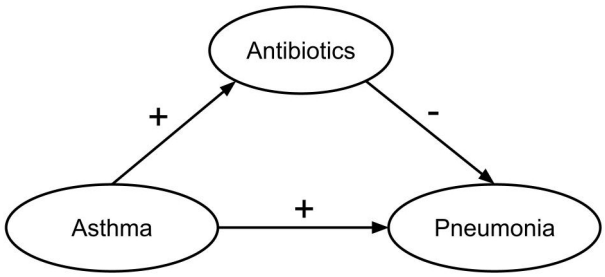


Figure 2.1: Asthma increases the likelihood of pneumonia. However, in the study, asthma also increased the (preemptive) use of antibiotics which generally protects against pneumonia and led to an overall lower pneumonia risk for asthma patients.

No rule would be apparent, stating "asthma ⇒ lower risk." Instead, the network would learn this rule and conceal it, potentially causing harm if deployed in real-life situations.

Although you could theoretically spot the problem by closely examining the data and applying domain knowledge, it's generally easier to identify such issues if you can understand what the model has learned. Machine learning models that aren't interpretable create a distance between the data and the modeler, and interpretability methods help bridge this gap.

2.2 Users may create their own interpretations

Here's a story about how the lack of interpretability led users of a model to develop their own incorrect interpretations. The story relates to sepsis, a life-threatening condition in which the body responds severely to an infection. As one of the most common causes of death in hospitals, sepsis is hard to diagnose, expensive to treat, and detrimental to patients, making early detection systems highly sought after.

Duke University and Duke Health Systems developed Sepsis Watch, an early warning system for sepsis in hospitals. This software system, based on deep neural networks, takes patient data as input and predicts whether the patient

is likely to develop sepsis (Elish and Watkins 2020). If the model detects a potential sepsis case, it triggers an alert that initiates a new hospital protocol for diagnosis and treatment. This protocol involves a rapid response team (RRT) nurse who monitors the alarms and informs the doctors, who then treat the patient. Numerous aspects of the implementation warrant discussion, especially the social implications of the new workflow, such as the hospital hierarchy causing nurses to feel uncomfortable instructing doctors. There was also considerable repair work carried out by RRT nurses to adapt the new system to the hospital environment. Interestingly, the report noted that the deep learning system didn't provide explanations for warnings, leaving it unclear why a patient was predicted to develop sepsis. The software merely displayed the score, resulting in occasional discrepancies between the model score and the doctor's diagnosis. Doctors would consequently ask nurses what they were observing that the doctors were not. The patient didn't seem septic, so why were they viewed as high-risk? However, the nurse only had access to the scores and some patient data, leading to a disconnect. Feeling responsible for explaining the model outputs, RRT nurses collected context from patient charts to provide an explanation. One nurse assumed the model was keying in on specific words in the medical record, which wasn't the case. The model wasn't trained on text. Another nurse also formed incorrect assumptions about the influence of lab values on the sepsis score. While these misunderstandings didn't hinder tool usage, they underscore an intriguing issue with the lack of interpretability: users may devise their own interpretations when none are provided.

2.3 Building trust in your models

Anecdotally, I've heard data scientists express their avoidance of certain models, like neural networks or gradient boosting, due to their lack of interpretability. This decision isn't always left to the developer or data scientist, but could be influenced by their environment: the end user of the model, the middle manager who needs to understand the model's limitations and capabilities, or the senior data scientist who prefers interpretable models and sees no need to change. A lack of interpretability can discourage the use of models deemed uninterpretable. The fear of unexplained outcomes or the inability to use the model in its intended way can be overwhelming. For instance, the coefficients in a linear regression model could be used to inform other decisions, or a dashboard might display

explanations alongside model scores to facilitate others' engagement with the predictions.

2.4 The limitations of inherently interpretable models

Is the solution to exclusively use "inherently" interpretable models? These may include:

- Linear regression and logistic regression
- Generalized additive models
- Decision rules & decision trees

Inherently interpretable typically means the model is constructed in a way that allows for easy understanding of its individual components. The prediction may be a weighted sum (linear model) or based on comprehensible rules. Some have even argued for the use of such models exclusively when the stakes are high (Rudin 2019).

However, there are two problems.

Problem 1: The definition of an interpretable model is ambiguous. One group may understand linear regression models, while another may not due to lack of experience. Even if you accept a linear regression model as interpretable, it can easily be made perplexing. For instance, by log-transforming the target, standardizing the features, adding interaction terms, using harder-to-interpret features, or adding thousands of features, an inherently interpretable model can become uninterpretable.

Problem 2: The models with the highest predictive performance are often not inherently interpretable. In machine learning, a metric is usually optimized. Boosted trees often emerge as the best choice in many scenarios (Grinsztajn et al. 2022). Most people wouldn't deem them interpretable, at least not in their original form. The same can be said for transformers, the standard for text (large language models, anyone?), and convolutional neural networks for image classification. Furthermore, ensembles of models often yield the best results and they are clearly less interpretable as they combine multiple models. Hence, restricting model selection to inherently interpretable models might lead

to inferior performance. This inferior performance could directly result in fewer sales, increased churn, or more false negative sepsis predictions.

So, what's the solution?

2.5 Model-agnostic interpretation is the answer

Model-agnostic methods like explainable artificial intelligence (XAI) or interpretable machine learning (IML) provide solutions for interpreting any machine learning model[1]. Despite the vast differences between machine learning models, from k-nearest neighbors to deep neural networks and support vector machines, model-agnostic methods are always applicable as they don't need knowledge of the model's inner mechanics, such as coefficients.

Consider playing fighting games on a console, where you push inputs (the controller) and observe the outcomes (the character fights). This is similar to how model-agnostic interpretable machine learning operates. The model is treated like a box with inputs and outputs; you manipulate the inputs, observe how the outputs change, and draw conclusions. Most model-agnostic interpretation methods can be summarized by the SIPA framework (Scholbeck et al. 2020):

- **S**ampling data.
- **I**ntervention on the data.
- **P**rediction step.
- **A**ggregating the results.

Various methods operate under the SIPA framework (Molnar 2022), including:

- Partial dependence plots, which illustrate how altering one (or two) features changes the average prediction.
- Individual conditional expectation curves, which perform the same function for a single data point.
- Accumulated Local Effect Plots, an alternative to partial dependence plots.

[1]The terms "Explainable AI" and "interpretable machine learning" are used interchangeably in this book. Some people use XAI more for post-hoc explanations of predictions and interpretable ML for inherently interpretable models. However, when searching for a particular method, it's advisable to use both terms.

- Permutation Feature Importance, quantifying a feature's importance for accurate predictions.
- Local interpretable model-agnostic explanations (LIME), explaining predictions with local linear models (Ribeiro et al. 2016).

SHAP is another model-agnostic interpretation method that operates by sampling data, intervening on it, making predictions, and then aggregating the results.

> **Tip**
>
> Even if you use an interpretable model, this book can be of assistance. Methods like SHAP can be applied to any model, so even if you're using a decision tree, SHAP can provide additional interpretation.

2.6 SHAP: An explainable AI technique

SHAP (Lundberg and Lee 2017a) is a game-theory-inspired method created to explain predictions made by machine learning models. SHAP generates one value per input feature (also known as SHAP values) that indicates how the feature contributes to the prediction of the specified data point. In the example in Figure 2.2, the prediction model estimates the probability of a person earning more than $50k based on that person's socio-economic factors. Some factors positively affect the predicted probability, while others negatively impact it. Understanding this figure isn't crucial at this point; it's simply a goal to keep in mind as we dive into the theory behind SHAP in the following chapters.

SHAP has gained popularity and is applied in various fields to explain predictive models:

- Identifying COVID-19 mortality factors (Smith and Alvarez 2021).
- Predicting heat wave-related mortality (Kim and Kim 2022).
- Wastewater treatment plant management (Wang et al. 2022).
- Genome-wide association studies (Johnsen et al. 2021).
- Accident detection (Parsa et al. 2020).
- NO2 forecasting (García and Aznarte 2020).
- Molecular design (Rodríguez-Pérez and Bajorath 2020).
- Gold price forecasting (Jabeur et al. 2021).

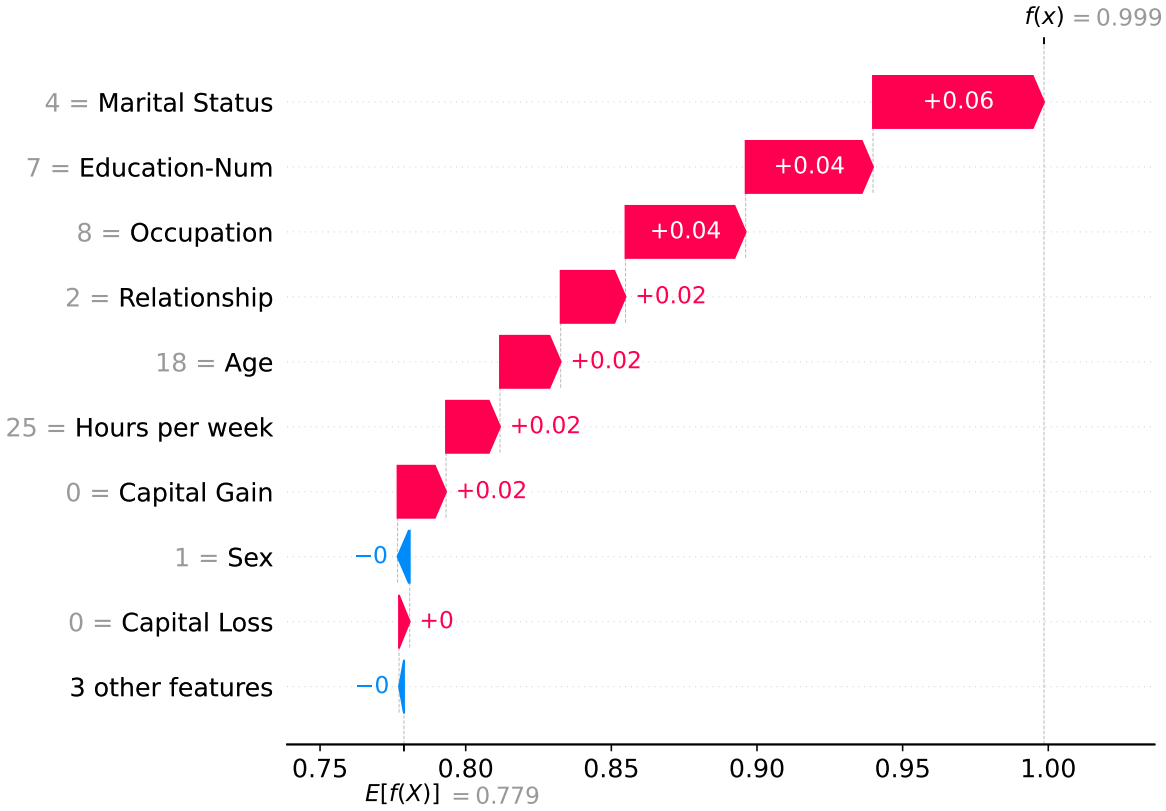


Figure 2.2: SHAP values to explain a prediction.

Given its wide range of applications, you are likely to find a use for SHAP in your work.

Before we talk about the practical application of SHAP, let's begin with its historical background, which provides context for the subsequent theory chapters.

3 A Short History of Shapley Values and SHAP

By the end of this chapter, you will be able to:

- Understand the key historical milestones of SHAP.
- Explain the relationship between SHAP and Shapley values.

This chapter offers an overview of the history of SHAP and Shapley values, focusing on their chronological development. The history is divided into three parts, each highlighted by a milestone:

- 1953: The introduction of Shapley values in game theory.
- 2010: The initial steps toward applying Shapley values in machine learning.
- 2017: The advent of SHAP, a turning point in machine learning.

3.1 Lloyd Shapley's pursuit of fairness

Shapley values have greater importance than might initially be apparent from this book. These values are named after their creator, Lloyd Shapley, who first introduced them in 1953. In the 1950s, game theory saw an active period, during which many core concepts were formulated, including repeated games, the prisoner's dilemma, fictitious play, and, of course, Shapley values. Lloyd Shapley, a mathematician, was renowned in game theory, with fellow theorist Robert Aumann calling him the "greatest game theorist of all time"[1]. After World War II, Shapley completed his PhD at Princeton University with a thesis titled "Additive and Non-Additive Set Functions." In 1953, his paper "A Value for n-Person

[1] https://www.wsj.com/articles/lloyd-shapley-won-the-nobel-prize-for-economics-1923-2016-1458342678

Games" (Shapley et al. 1953) introduced Shapley values. In 2012, Lloyd Shapley and Alvin Roth were awarded the Nobel Prize in Economics[2] for their work in "market design" and "matching theory."

Shapley values serve as a solution in cooperative game theory, which deals with games where players cooperate to achieve a payout. They address the issue of a group of players participating in a collaborative game, where they work together to reach a certain payout. The payout of the game needs to be distributed among the players, who may have contributed differently. Shapley values provide a mathematical method of fairly dividing the payout among the players.

Shapley values have since become a cornerstone of coalitional game theory, with applications in various fields such as political science, economics, and computer science. They are frequently used to determine fair and efficient strategies for resource distribution within a group, including dividing profits among shareholders, allocating costs among collaborators, and assigning credit to contributors in a research project. However, Shapley values were not yet employed in machine learning, which was still in its early stages at the time.

3.2 Early days in machine learning

Fast forward to 2010. Shapley hadn't yet received his Nobel Prize in Economics, but the theory of Shapley values had been established for nearly 60 years. In contrast, machine learning had made tremendous strides during this period. In 2012, the ImageNet competition (Deng et al. 2009), led by Fei-Fei Li, was won for the first time by a team using a deep neural network (AlexNet) with a significant lead over the runner-up. Machine learning continued to advance and attract more research in many other areas.

While Shapley values had previously been defined for linear models, 2010 marks the beginning of model-agnostic estimation of Shapley values. In 2010, Erik Štrumbelj and Igor Kononenko published a paper titled "An efficient explanation of individual classifications using game theory" (Štrumbelj and Kononenko 2010), proposing the use of Shapley values to explain machine learning model predictions.

[2]It's not the real Nobel Prize, but the "Nobel Memorial Prize in Economic Sciences." Officially, it's termed the "Sveriges Riksbank Prize in Economic Sciences in Memory of Alfred Nobel." This prize is a kind of surrogate Nobel award created by economists since they were not included in the original five Nobel Prizes.

In 2014, they further developed their methodology for computing Shapley values (Štrumbelj and Kononenko 2014).

However, this approach did not immediately gain popularity. Some possible reasons why Shapley values were not widely adopted at the time include:

- Explainable AI/Interpretable machine learning was not as widely recognized.
- The papers by Štrumbelj and Kononenko did not include code.
- The estimation method was still relatively slow and not suitable for image or text classification.

Next, we will look at the events that led to the rise of Shapley values in machine learning.

3.3 The SHAP Cambrian explosion

In 2016, Ribeiro et al. (2016) published a paper introducing Local Interpretable Model-Agnostic Explanations (LIME), a method that uses local linear regression models to explain predictions. This paper served as a catalyst for the field of explainable AI and interpretable machine learning. A more cautious claim might be that the paper's publication coincided with a growing interest in interpreting machine learning models. The prevailing sentiment at the time was a concern over the complexity of advanced machine learning algorithms, such as deep neural networks, and the lack of understanding of how these models generate their predictions.

Shortly after the LIME paper, in 2017, Scott Lundberg and Su-In Lee published a paper titled "A Unified Approach to Interpreting Model Predictions" (Lundberg and Lee 2017b). This paper introduced SHapley Additive exPlanations (SHAP), another method to explain machine learning predictions. The paper was published at NIPS, now known as NeurIPS[3]. NeurIPS is a major machine learning conference, and if your research is published there, it's more likely to draw attention. But what exactly did the SHAP paper introduce, given that Shapley values for machine learning were already defined in 2010/2014?

[3]The name NIPS faced criticism due to its association with "nipples" and its derogatory usage against Japanese individuals, leading to its change to NeurIPS.

Lundberg and Lee presented a new way to estimate SHAP values using a weighted linear regression with a kernel function to weight the data points[4]. The paper also demonstrated how their proposed estimation method could integrate other explanation techniques, such as DeepLIFT (Shrikumar et al. 2017), LIME (Ribeiro et al. 2016), and Layer-Wise Relevance Propagation (Bach et al. 2015).

Here's why I believe SHAP gained popularity:

- It was published in a reputable venue (NIPS/NeurIPS).
- It was a pioneering work in a rapidly growing field.
- Ongoing research by the original authors and others contributed to its development.
- The open-source `shap` Python package with a wide range of features and plotting capabilities

The availability of open-source code played a significant role, as it enabled people to integrate SHAP values into their projects.

Naming conventions

The naming can be slightly confusing for several reasons:

- Both the method and the resulting numbers can be referred to as Shapley values (and SHAP values).
- Lundberg and Lee (2017b) renamed Shapley values for machine learning as SHAP, an acronym for SHapley Additive exPlanations.

This book will adhere to these conventions:

- Shapley values: the original method from game theory.
- SHAP: the application of Shapley values for interpreting machine learning predictions.
- SHAP values: the resulting values from using SHAP for the features.
- `shap`: the library that implements SHAP.

"SHAP" is similar to a brand name used to describe a product category, like Post-it, Jacuzzi, Frisbee, or Band-Aid. I chose to use it since it's well-established in the community and it distinguishes between the general game-

[4]The `shap` package no longer uses Kernel SHAP by default, rendering the paper somewhat historical.

theoretic method of Shapley values and the specific machine learning application of SHAP.

Since its inception, SHAP's popularity has steadily increased. A significant milestone was reached in 2020 when Lundberg et al. (2020) proposed an efficient computation method specifically for SHAP, targeting tree-based models. This advancement was crucial because tree-boosting excels in many applications, enabling rapid estimation of SHAP values for state-of-the-art models. Another remarkable achievement by Lundberg involved extending SHAP beyond individual predictions. He stacked SHAP values, similar to assembling Legos, to create global model interpretations. This method was made possible by the fast computation designed for tree-based models. Thanks to numerous contributors, Lundberg continued to enhance the `shap` package, transforming it into a comprehensive library with a wide range of estimators and functionalities. Besides Lundberg's work, other researchers have also contributed to SHAP, proposing extensions. Moreover, SHAP has been implemented in other contexts, indicating that the `shap` package is not the only source of this method.

Given this historical context, we will begin with the theory of Shapley values and gradually progress to SHAP.

4 Theory of Shapley Values

By the end of this chapter, you will be able to:

- Understand the theory of Shapley values.
- Calculate Shapley values for simple games.
- Understand the axioms of Shapley values: efficiency, symmetry, dummy, and additivity.

To learn about SHAP, we first discuss the theory behind Shapley values from game theory. We will progressively define a fair payout[^fair] in a coalition of players and ultimately arrive at Shapley values (spoiler alert). [^fair]: There is no perfect definition of fairness everyone would agree upon. Shapley values define a very specific version of fairness, which can be seen as egalitarian.

4.1 Who's going to pay for that taxi?

Consider a concrete example that can be seen as a coalitional game: splitting the cost of a taxi ride. Alice, Bob, and Charlie have dinner together and share a taxi ride home. The total cost is $51. The question is, how should they divide the costs fairly?

View the taxi ride as a coalitional game: Alice, Bob, and Charlie form a coalition and receive a specific payout. In this case, the payout is negative (costs), but this doesn't change the fact that we can consider this as a coalitional game. To determine a fair distribution of the costs, we first pose simpler questions: How much would the ride cost for a random coalition of passengers? For instance, how much would Alice pay for a taxi ride if she were alone? How much would Alice and Bob pay if they shared a taxi? Let's suppose it would be $15 for Alice alone. Alice and Bob live together, but adding Bob to the ride increases the cost

to $25, as he insists on a more spacious, luxurious taxi, adding a flat $10 to the ride costs. Adding Charlie to Alice and Bob's ride increases the cost to $51 since Charlie lives somewhat further away. We define the taxi ride costs for all possible combinations and compile the following table:

Passengers	Cost	Note
∅	$0	No taxi ride, no costs
{Alice}	$15	Standard fare to Alice's & Bob's place
{Bob}	$25	Bob always insists on luxury taxis
{Charlie}	$38	Charlie lives slightly further away
{Alice, Bob}	$25	Bob always gets his way
{Alice, Charlie}	$41	Drop off Alice first, then Charlie
{Bob, Charlie}	$51	Drop off luxurious Bob first, then Charlie
{Alice, Bob, Charlie}	$51	The full fare with all three of them

The coalition ∅ is a coalition without any players in it, i.e., an empty taxi. This table seems like a step in the right direction, giving us an initial idea of how much each person contributes to the cost of the ride.

4.2 Calculating marginal contributions for the taxi costs

We can take a step further by calculating the so-called marginal contributions of each passenger to each coalition. For example, how much additional cost does Alice incur when she joins a taxi with Bob already in it?

Marginal contribution

The marginal contribution of a player to a coalition is the value of the coalition *with* the player minus the value of the coalition *without* the player. In the taxi example, the value of a coalition is equal to the cost of the ride as detailed in the above table. Therefore, the marginal contribution of, for instance, Charlie to a taxi already containing Bob is the cost of the taxi with Bob and Charlie, minus the cost of the taxi with Bob alone.

Using the table, we can easily calculate the marginal contributions. Taking an example, if we compare the cost between the {Alice, Bob} coalition and Bob alone, we derive the marginal contribution of Alice, the "player", to the coalition {Bob}. In this scenario, it's \$25 - \$25 = \$0, as the taxi ride cost remains the same. If we calculate the marginal contribution of Bob to the {Alice} coalition, we get \$25 - \$15 = \$10, meaning adding Bob to a taxi ride with Alice increases the cost by \$10. We calculate all possible marginal contributions in this way:

Addition	To Coalition	Cost Before	Cost After	Marginal Contribution
Alice	∅	\$0	\$15	\$15
Alice	{Bob}	\$25	\$25	\$0
Alice	{Charlie}	\$38	\$41	\$3
Alice	{Bob, Charlie}	\$51	\$51	\$0
Bob	∅	\$0	\$25	\$25
Bob	{Alice}	\$15	\$25	\$10
Bob	{Charlie}	\$38	\$51	\$13
Bob	{Alice, Charlie}	\$41	\$51	\$10
Charlie	∅	\$0	\$38	\$38
Charlie	{Alice}	\$15	\$41	\$26
Charlie	{Bob}	\$25	\$51	\$26
Charlie	{Alice, Bob}	\$25	\$51	\$26

We're one step closer to calculating a fair share of ride costs. Could we just average these marginal contributions per passenger? We could, but that would assign equal weight to every marginal contribution. However, one could argue that we learn more about how much Alice should pay when we add her to an empty taxi compared to when we add her to a ride with Bob. But how much more informative?

One way to answer this question is by considering all possible permutations of Alice, Bob, and Charlie. There are $3! = 3 * 2 * 1 = 6$ possible permutations of passengers:

- Alice, Bob, Charlie
- Alice, Charlie, Bob

- Bob, Alice, Charlie
- Charlie, Alice, Bob
- Bob, Charlie, Alice
- Charlie, Bob, Alice

We can use these permutations to form coalitions, for example, for Alice. Each permutation then maps to a coalition: People who come before Alice in the order are in the coalition, people after are not. Since in a coalition the order of passengers doesn't matter, some coalitions will occur more often than others when we iterate through all permutations like this: In 2 out of 6 permutations, Alice is added to an empty taxi; In 1 out of 6, she is added to a taxi with Bob; In 1 out of 6, she is added to a taxi with Charlie; And in 2 out of 6, she is added to a taxi with both Bob and Charlie. We use these counts to weight each marginal contribution to continue our journey towards a fair cost sharing.

We could make different decisions regarding how to "fairly" allocate the costs to the passengers. For instance, we could weight the marginal contributions differently. We could divide the cost by 3. Alternatively, we could use solutions that depend on the order of passengers: Alice alone would pay \$15, when we add Bob it's +\$10, which would be his share, and Charlie would pay the remainder. However, all these different choices would lead us away from Shapley values.

4.3 Averaging marginal contributions

In two of these cases, Alice was added to an empty taxi, and in one case, she was added to a taxi with only Bob. By weighting the marginal contributions accordingly, we calculate the following weighted average marginal contribution for Alice, abbreviating Alice, Bob, and Charlie to A, B, and C:

$$\frac{1}{6}(\underbrace{2 \cdot \$15}_{\text{A to } \emptyset} + \underbrace{1 \cdot \$0}_{\text{A to B}} + \underbrace{1 \cdot \$3}_{\text{A to C}} + \underbrace{2 \cdot \$0}_{\text{A to B,C}}) = \$5.50$$

We multiply by $\frac{1}{6}$ because 6 is the sum of the weights (2 + 1 + 1 + 2). That's how much Alice should pay for the ride: \$5.50.

We can calculate the contribution for Bob the same way:

$$\frac{1}{6}(\underbrace{2 \cdot \$25}_{\text{B to } \emptyset} + \underbrace{1 \cdot \$10}_{\text{B to A}} + \underbrace{1 \cdot \$13}_{\text{B to C}} + \underbrace{2 \cdot \$10}_{\text{B to A,C}}) = \$15.50$$

And for Charlie:

$$\frac{1}{6}(\underbrace{2 \cdot \$38}_{\text{C to } \emptyset} + \underbrace{1 \cdot \$26}_{\text{C to A}} + \underbrace{1 \cdot \$26}_{\text{C to B}} + \underbrace{2 \cdot \$26}_{\text{C to A,B}}) = \$30.00$$

The individual contributions sum to the total cost: \$5.50 + \$15.50 + \$30.00 = \$51.00. Perfect! And that's it, this is how we compute Shapley values (Shapley et al. 1953).

Let's formalize the taxi example in terms of game theory and explore the Shapley value theory, which makes Shapley values a unique solution.

4.4 Calculating Shapley values

The upcoming sections will use several game theoretic terms. Even though we've already used most of them in the previous example, here's an overview for reference.

Term	Math Term	Taxi Example
Player	$1, ..., \|N\|$	Passenger, for example Alice
Coalition of All Players	N	{Alice, Bob, Charlie}
Coalition	S	Any combination of passengers, ranging from $\emptyset$ to {Alice, Bob, Charlie}.
Size of a Coalition	$\|S\|$	For example, $\|\{\text{Alice}\}\| = 1$, $\|\{\text{Alice}, \text{Bob}, \text{Charlie}\}\| = 3$
Value Function	$v()$	Defined by the table showing all possible arrangements of passengers in the taxi
Payout	v(N)	\$51, the cost of the taxi ride with all passengers

Term	Math Term	Taxi Example
Shapley Value	ϕ_j	For example, $\phi_1 = \$5.50$ for Alice, $\phi_2 = \$15.50$ for Bob, and $\phi_3 = \$30$ for Charlie.

The value function v can also be referred to as the characteristic function.

We have explored how to calculate Shapley values through the taxi ride example. Now, let's formalize Shapley values for the general case:

$$\phi_j = \sum_{S \subseteq N \setminus \{j\}} \frac{|S|!(N - |S| - 1)!}{N!} (v(S \cup \{j\}) - v(S)) \tag{4.1}$$

The value function $v : P(N) \mapsto \mathbb{R}$ maps from all possible coalitions of N players to a real number, which represents the payout for that coalition. The formula is quite complex, so let's break it down.

- $v(S \cup \{j\}) - v(S)$: This is the core of the equation. It represents the marginal contribution of player j to coalition S. If j is Alice and $S = \{Bob\}$, then this part expresses how much more expensive the ride becomes when Alice joins Bob.
- $\sum_{S \subseteq N \setminus \{j\}}$: The entire formula is a sum over all possible coalitions without j. If we calculate the Shapley value for Alice, we sum over the coalitions: $\emptyset$, $\{Bob\}$, $\{Charlie\}$, and $\{Bob, Charlie\}$.
- $\frac{|S|!(N-|S|-1)!}{N!}$: This term determines the weight of a marginal contribution. $\emptyset$ and $N \setminus \{j\}$ get the highest weights. The $|N|!$ in the denominator ensures that the sum of the weights equals 1.

The complex formula isn't so intimidating after all!

> **Shapley value formula summary**
>
> The Shapley value is the weighted average of a player's marginal contributions to all possible coalitions.

4.5 The axioms behind Shapley values

We now have the formula, but where did it come from? Lloyd Shapley derived it (Shapley et al. 1953), but it didn't just materialize out of thin air. He proposed axioms defining what a fair distribution could look like, and from these axioms, he derived the formula. Lloyd Shapley also proved that based on these axioms, the Shapley value formula yields a unique solution.

Let's discuss these axioms, namely **Efficiency**, **Symmetry**, **Dummy**, and **Additivity**. An axiom is a statement accepted as self-evidently true. Consider the axioms as defining fairness when it comes to payouts in team play.

4.5.1 Efficiency

The efficiency axiom states that the sum of the contributions must precisely add up to the payout. This makes a lot of sense. Consider Alice, Bob, and Charlie sharing a taxi ride and calculating their individual shares, but the contributions don't equal the total taxi fare. All three, including the taxi driver, would find this method useless. The efficiency axiom can be expressed formally as:

$$\sum_{j \in N} \phi_j = v(N)$$

4.5.2 Symmetry

The symmetry principle states that if two players are identical, they should receive equal contributions. Identical means that all their marginal contributions are the same. For instance, if Bob wouldn't need the luxury version of the taxi, his marginal contributions would be exactly the same as Alice's. The symmetry axiom says that in such situations, both should pay the same amount, which seems fair.

We can also express symmetry mathematically for two players j and k:

If $v(S \cup \{j\}) = v(S \cup \{k\})$ for all $S \subseteq N \backslash \{j, k\}$, then $\phi_j = \phi_k$.

4.5.3 Dummy or Null Player

The Shapley value for a player who doesn't contribute to any coalition is zero, which seems quite fair. Let's introduce Dora, Charlie's dog, and consider her an additional player. Assuming there's no extra cost for including Dora in any ride, all of Dora's marginal contributions would be \$0. The dummy axiom states that when all marginal contributions are zero, the Shapley value should also be zero. This rule seems reasonable, especially as Dora doesn't have any money.

To express this axiom formally:

If $v(S \cup \{j\}) = v(S)$ for all $S \subseteq N \backslash \{j\}$, then $\phi_j = 0$.

4.5.4 Additivity

In a game with two value functions v_1 and v_2, the Shapley values for the sum of the games can be expressed as the sum of the Shapley values:

$$\phi_{j,v_1+v_2} = \phi_{j,v_1} + \phi_{j,v_2}$$

Imagine Alice, Bob, and Charlie not only sharing a taxi but also going out for ice cream. Their goal is to fairly divide not just the taxi costs, but both the taxi and ice cream costs. The additivity axiom suggests that they could first calculate each person's fair share of the ice cream costs, then the taxi costs, and add them up per person.

These four[1] axioms ensure the uniqueness of the Shapley values, indicating there's only one solution presented in the Shapley formula, Equation 4.1. The proof of why this is the case won't be discussed in this book, as it would be too detailed. Instead, it's time to relate this approach to explaining machine learning predictions.

[1] A fifth axiom called *Linearity* or *Marginality* exists, but it can be derived from the other axioms, so it doesn't introduce any new requirements for fair payouts.

5 From Shapley Values to SHAP

By the end of this chapter, you will be able to:

- Describe a prediction as a coalitional game.
- Explain how SHAP values are defined.
- Interpret Shapley value axioms for machine learning predictions.

We have been learning about Shapley values from coalitional game theory. But how do these values connect to machine learning explanations? The connection might not seem apparent – it certainly didn't to me when I first learned about SHAP.

5.1 A machine learning example

Consider the following scenario: You have trained a machine learning model f to predict apartment prices. For a specific apartment $x^{(i)}$, the model predicts $f(x^{(i)}) = 300,000$. Your task is to explain this prediction. The apartment has an area of 50 m^2 (538 square feet), is located on the 2nd floor, has a nearby park, and cats are banned. These features are what the model used to make the prediction.

The average prediction for all apartments in the data is €310,000, which places the predicted price of this specific apartment slightly below average. How much did each feature value contribute to the prediction compared to the average prediction? In the apartment example, the feature values `park-nearby`, `cat-banned`, `area-50`, and `floor-2nd` collectively led to a prediction of €300,000. Our goal is to explain the difference between the actual prediction (€300,000) and the average prediction (€310,000), which is a difference of -€10,000. Here's an example of what an answer might look like: `park-nearby` contributed €30,000; `area-50`

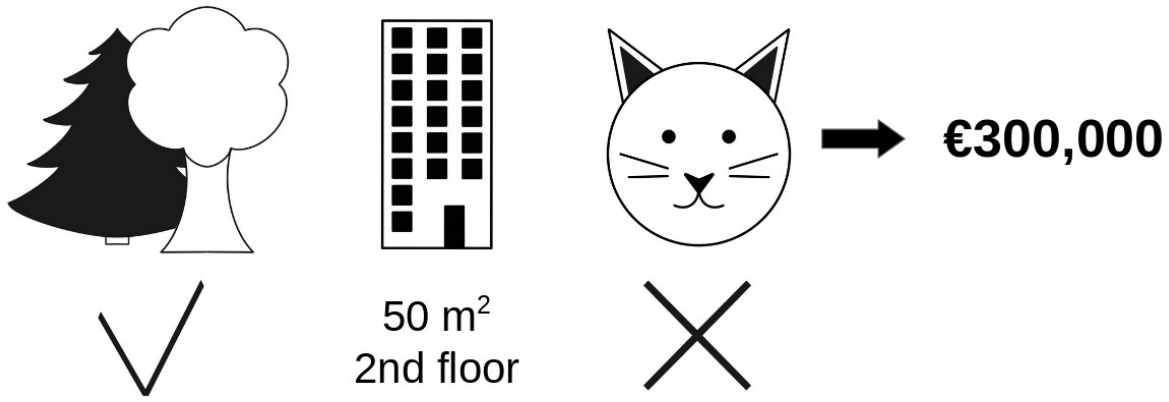


Figure 5.1: The predicted price for a 50 m^2 2nd floor apartment with a nearby park and cat ban is €300,000. Our goal is to explain how each of these feature values contributed to the prediction.

contributed €10,000; `floor-2nd` contributed €0; and `cat-banned` contributed -€50,000. The contributions add up to -€10,000, which is the final prediction minus the average predicted apartment price. From the Shapley theory chapter, we know that Shapley values can provide a fair attribution of a payout. We just need to translate concepts from game theory to machine learning prediction concepts.

5.2 Viewing a prediction as a coalitional game

A prediction can be viewed as a coalitional game by considering each feature value of an instance as a "player" in a game. The "payout" is the predicted value. We refer to this version of Shapley values adapted for machine learning predictions as "SHAP". Let's translate the terms from game theory to machine learning predictions one by one with the following table:

Concept	Machine Learning	Term
Player	Feature index	j
Coalition	Set of features	$S \subseteq \{1, \dots, p\}$
Not in coalition	Features not in coalition S	$C : C = \{1, \dots, p\} \backslash S$

Concept	Machine Learning	Term
Coalition size	Number of features in coalition S	$\|S\|$
Total number of players	Number of features	p
Total payout	Prediction for $x^{(i)}$ minus average prediction	$f(x^{(i)}) - \mathbb{E}(f(X))$
Value function	Prediction for feature values in coalition S minus expected	$v_{f,x^{(i)}}(S)$
SHAP value	Contribution of feature j towards payout	$\phi_j^{(i)}$

You may have questions about these terms, but we will discuss them shortly. The value function is central to SHAP, and we will discuss it in detail. This function is closely related to the simulation of absent features.

5.3 The SHAP value function

The SHAP value function, for a given model f and data instance $x^{(i)}$, is defined as:

$$v_{f,x^{(i)}}(S) = \int f(x_S^{(i)} \cup X_C) d\mathbb{P}_{X_C} - \mathbb{E}(f(X))$$

Note

The value function relies on a specific model f and a particular data point to be explained $x^{(i)}$, and maps a coalition S to its value. Although the correct notation is $v_{f,x^{(i)}}(S)$, I will occasionally use $v(S)$ for brevity. Another misuse of notation: I use the union operator for the feature vector: $x_S^{(i)} \cup X_C$ is a feature vector $\in \mathbb{R}^p$ where values at positions S have values from $x_S^{(i)}$ and the rest are random variables from X_C.

This function provides an answer for the simulation of absent features. The second part $\mathbb{E}(f(X))$ is straightforward: It ensures the value of an empty coalition $v(\emptyset)$ equals 0.

Confirm this for yourself:

$$
\begin{aligned}
v(\emptyset) &= \int f(X_1, \ldots, X_p) d\mathbb{P}_X - E_X(f(X)) && (5.1)\\
&= E_X(f(X)) - E_X(f(X)) && (5.2)\\
&= 0 && (5.3)
\end{aligned}
$$

The first part of the value function, $\int f(x_S^{(i)} \cup X_C) dX_C$, is where the magic occurs. The model prediction function f, which is central to the value function, takes the feature vector $x^{(i)} \in \mathbb{R}^p$ as input and generates the prediction $\in \mathbb{R}$. However, we only know the features in set S, so we need to account for the features not in S, which we index with C.

SHAP's approach is to treat the unknown features as random variables and integrate over their distribution. This concept of integrating over the distribution of a random variable is called marginalization.

> **Marginalization**
>
> Integration of a function typically involves calculating the area under the curve. However, when integrating with respect to a distribution, certain portions under the curve are weighted more heavily based on their likelihood within the integral.

This means that we can input "known" features directly into the model f, while absent features are treated as random variables. In mathematical terms, I distinguish a random variable from an observed value by capitalizing it:

Feature value	Random variable
$x_j^{(i)}$	X_j
$x_S^{(i)}$	X_S
$x_C^{(i)}$	X_C

Feature value	Random variable
$x^{(i)}$	X

Let's revisit the apartment example:

Park	Cat	Area	Floor	Predicted Price
Nearby	Banned	50	2nd	€300,000

Informally, the value function for the coalition of park, floor would be:

$$v(\{\text{park}, \text{floor}\}) = \int f(x_{park}, X_{cat}, X_{area}, x_{floor}) d\mathbb{P}_{X_{cat,area}} - E_X(f(X)),$$

where $x_{park} = \text{nearby}$, $x_{floor} = 2$, and $E_X(f(X)) = 300.000$.

- The features 'park' and 'floor' are "present", so we input their corresponding values into f.
- The features 'cat' and 'area' are "absent", and thus are treated as random variables and integrated over.

5.4 Marginal contribution

We are gradually working our way up to the SHAP value. We've examined the value function, and the next step is to determine the marginal contribution. This is the contribution of feature j to a coalition of features S.

The marginal contribution of j to S is:

$$
\begin{aligned}
v(S \cup j) - v(S) &= \int f(x^{(i)}_{S \cup j} \cup X_{C \setminus j}) d\mathbb{P}_{X_{C \setminus j}} - \mathbb{E}(f(X)) \\
&\quad - \left(\int f(x^{(i)}_{S} \cup X_{C}) d\mathbb{P}_{X_{C}} - \mathbb{E}(f(X)) \right) \\
&= \int f(x^{(i)}_{S \cup j} \cup X_{C \setminus j}) d\mathbb{P}_{X_{C \setminus j}} \\
&\quad - \int f(x^{(i)}_{S} \cup X_{C}) d\mathbb{P}_{X_{C}}
\end{aligned}
$$

For instance, the contribution of 'cat' to a coalition of {park, floor} would be:

$$
v(\{\text{cat}, \text{park}, \text{floor}\}) - v(\{\text{park}, \text{floor}\})
$$

The resulting marginal contribution describes the change in the value of the coalition {park, floor} when the 'cat' feature is included. Another way to interpret the marginal contribution is that present features are known, absent feature values are unknown, so the marginal contribution illustrates how much the value changes from knowing j in addition to already knowing S.

5.5 Putting it all together

Combining all the terms into the Shapley value equation, we get the SHAP equation:

$$
\begin{aligned}
\phi^{(i)}_{j} &= \sum_{S \subseteq \{1, \ldots, p\} \setminus j} \frac{|S|! \, (p - |S| - 1)!}{p!} \\
&\quad \cdot \left(\int f(x^{(i)}_{S \cup j} \cup X_{C \setminus j}) d\mathbb{P}_{X_{C \setminus j}} - \int f(x^{(i)}_{S} \cup X_{C}) d\mathbb{P}_{X_{C}} \right)
\end{aligned}
$$

The SHAP value $\phi^{(i)}_{j}$ of a feature value is the average marginal contribution of a feature value $x^{(i)}_{j}$ to all possible coalitions of features. And that concludes it.

This formula is similar to the one in the Shapley Theory Chapter, but the value function is adapted to explain a machine learning prediction. The formula, once again, is an average of marginal contributions, each contribution being weighted based on the size of the coalition.

5.6 Interpreting SHAP values through axioms

The axioms form the foundation for defining Shapley values. As SHAP values are Shapley values with a specific value function and game definition, they adhere to these axioms. This has been demonstrated by Štrumbelj and Kononenko (2010), Štrumbelj and Kononenko (2014), and Lundberg and Lee (2017b). Given that SHAP follows the principles of Efficiency, Symmetry, Dummy, and Additivity, we can deduce how to interpret SHAP values or at least obtain a preliminary understanding. Let's explore each axiom individually and determine their implications for the interpretation of SHAP values.

5.6.1 Efficiency: SHAP values correspond to the (centered) prediction

SHAP values must total to the difference between the prediction for $x^{(i)}$ and the expected prediction:

$$\sum_{j=1}^{p} \phi_j^{(i)} = f(x^{(i)}) - \mathbb{E}(f(X))$$

Implications: The efficiency axiom is prevalent in explainable AI and is adhered to by methods like LIME. This axiom guarantees that attributions are on the scale of the output, allowing us to interpret the results as contributions to the prediction. Gradients, another method for explaining model predictions, do not sum up to the prediction, hence in my opinion, are more challenging to interpret.

Symmetry: Feature order is irrelevant

If two feature values j and k contribute equally to all possible coalitions, their contributions should be equal.

Given

$$v_{f,x^{(i)}}(S \cup \{j\}) = v_{f,x^{(i)}}(S \cup \{k\})$$

for all

$$S \subseteq \{1, ..., p\} \backslash \{j, k\}$$

then

$$\phi_j^{(i)} = \phi_k^{(i)}$$

Implications: The symmetry axiom implies that the attribution shouldn't depend on any ordering of the features. If two features contribute equally, they will receive the same SHAP value. Other methods, such as the breakdown method (Staniak and Biecek 2018) or counterfactual explanations, violate the symmetry axiom because two features can impact the prediction equally without receiving the same attribution. For example, the breakdown method also computes attributions, but does it by adding one feature at a time, so that the order by which features are added matters for the explanation. Symmetry is essential for accurately interpreting the order of SHAP values, for instance, when ranking features using SHAP importance (sum of absolute SHAP values per feature).

Dummy: Features not influencing the prediction receive a SHAP value of 0

A feature j that does not alter the predicted value, regardless of the coalition of feature values it is added to, should have a SHAP value of 0.

Given

$$v_{f,x^{(i)}}(S \cup \{j\}) = v_{f,x^{(i)}}(S)$$

for all

$$S \subseteq \{1, \dots, p\}$$

then

$$\phi_j^{(i)} = 0$$

Implications: The dummy axiom ensures that unused features by the model receive a zero attribution. This is an obvious implication. For instance, if a sparse linear regression model was trained, we can be sure that a feature with a $\beta_j = 0$ will have a SHAP value of zero for all data points.

5.6.2 Additivity: Additive predictions correspond to additive SHAP values

For a game with combined payouts $v_1 + v_2$, the respective SHAP values are:

$$\phi_j^{(i)}(v_1) + \phi_j^{(i)}(v_2)$$

Implications: Consider a scenario where you've trained a random forest, meaning the prediction is an average of numerous decision trees. The Additivity property ensures that you can compute a feature's SHAP value for each tree separately and average them to obtain the SHAP value for the random forest. For an additive ensemble of models, the final SHAP value equals the sum of the individual SHAP values.

> **Note**
>
> An alternative formulation of the SHAP axioms exists where the Dummy and Additivity axioms are replaced with a Linearity axiom; however, both formulations eventually yield the SHAP values.

This chapter has provided theoretical SHAP values. However, we face a significant problem: In practice, we lack a closed-form expression for f and we are unaware of the distributions of X_C. This means we are unable to calculate the SHAP values, but, fortunately, we can estimate them.

6 Estimating SHAP Values

By the end of this chapter, you will be able to:

- Understand why SHAP values need to be estimated.
- Describe the permutation estimation method.
- Provide an overview of different SHAP estimation methods.

In the previous chapter, we applied the Shapley value concepts from game theory to machine learning. While exact Shapley values can be calculated for simple games, SHAP values must be estimated for two reasons:

- The value function utilized by SHAP requires integration over the feature distribution. However, since we only have data and lack knowledge of the distributions, we must use estimation techniques like Monte Carlo integration.
- Machine learning models often possess many features. As the number of coalitions increases exponentially with the number of features (2^p), it might become too time-consuming to compute the marginal contributions of a feature to all coalitions. Instead, we have to sample coalitions.

Let's assume we have a limited number of features for which we can still iterate through all coalitions. This allows us to focus on estimating the SHAP values from data without sampling coalitions.

6.1 Estimating SHAP values with Monte Carlo integration

Recap: SHAP values are computed as the average marginal contribution of a feature value across all possible coalitions. A coalition, in this context, is any

subset of feature values, including the empty set and the set containing all feature values of the instance. When features are not part of a coalition, the prediction function still requires that we input some value. This problem was theoretically solved by integrating the prediction function over the absent features. Now, let's explore how we can estimate this integral using our apartment example.

The following figure evaluates the marginal contribution of the `cat-banned` feature value when added to a coalition of `park-nearby` and `area-50`. To compute the marginal contribution, we need two coalitions: {park-nearby, cat-banned, area-50} and {park-nearby, area-50}. For the absent features, we would have to integrate the prediction function over the distribution of floor, and floor + cat, respectively.

However, we don't have these distributions, so we resort to using Monte Carlo integration.

> **Monte Carlo integration**
>
> Monte Carlo integration is a method for approximating the integral of a function with respect to a random variable. It does this by drawing samples from the variable and averaging the function output for those samples. This technique allows us to sum over data instead of integrating over distributions.

Using Monte Carlo integration, we can estimate the value functions for our apartment by sampling the absent features from our data and averaging the predictions. In this case, the data are the other apartments. Sometimes, I'll refer to this data as background data.

> **Background data**
>
> The replacement of absent feature values with randomly drawn ones requires a dataset to draw from, known as the background data. This could be the same data that was used to train the model. The background data serves as the context for the interpretation of the resulting SHAP values.

Let's illustrate what sampling from the background data looks like by drawing just one sample for the Monte Carlo integration. Although a single sample results in a very unstable estimate of the integral, it helps us understand the concept.

Let's say the randomly sampled apartment has the following characteristics:

Park	Cat	Area	Floor	Predicted Price
Nearby	Allowed	100	1st	€504,000

Then, we replace the `floor-2nd` value of the original apartment with the randomly drawn `floor-1st` value. We then predict the price of the apartment with this combination (€310,000), which is the value function for the first coalition, v({park-nearby, cat-banned, area-50}).

Park	Cat	Area	Floor	Predicted Price
Nearby	Banned	50	1st	€310,000

Next, we replace `cat-banned` in the coalition with a random value of the cat allowed/banned feature from the same apartment that we sampled. In essence, we are estimating v({park-nearby, area-50}).

Park	Cat	Area	Floor	Predicted Price
Nearby	Allowed	50	1st	€320,000

In this scenario, the replaced value was `cat-allowed`, but it could have been `cat-banned` if we had drawn a different apartment. We predict the apartment price for the coalition of `park-nearby` and `area-50` to be €320,000. Therefore, the marginal contribution of `cat-banned` is €310,000 - €320,000 = -€10,000.

This estimate is based on the values of a single, randomly drawn apartment that served as a "donor" for the cat and floor feature values. This is not an optimal estimate of the marginal contribution as it relies on only one Monte Carlo sample. To obtain better estimates, we can repeat this sampling process and average the marginal contributions.

Now, let's get into the formalities. The value of a coalition of features S is estimated as:

$$\hat{v}(S) = \frac{1}{n} \sum_{k=1}^{n} \left(f(x_S^{(i)} \cup x_C^{(k)}) - f(x^{(k)}) \right)$$

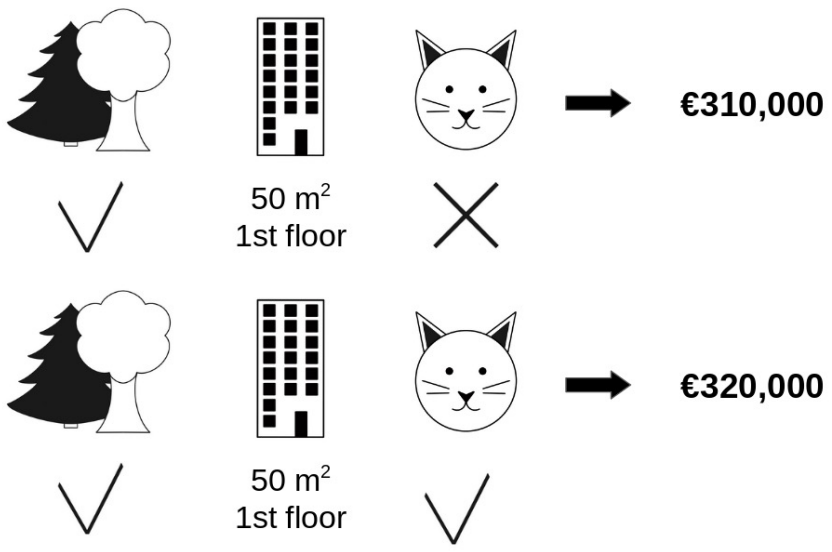


Figure 6.1: One Monte Carlo sample to estimate the marginal contribution of `cat-banned` to the prediction when added to the coalition of `park-nearby` and `area-50`.

Here, n is the number of data samples drawn from the data. The hat on the $\hat{v}$ signifies that this is an estimate of the value function v.

The marginal contribution of a feature j added to a coalition S is given by:

$$\begin{aligned}\hat{\Delta}_{S,j} &= \hat{v}(S \cup \{j\}) - \hat{v}(S) \\ &= \frac{1}{n}\sum_{k=1}^{n}\left(f(x^{(i)}_{S\cup\{j\}} \cup x^{(k)}_{C\setminus\{j\}}) - f(x^{(i)}_{S} \cup x^{(k)}_{C})\right)\end{aligned}$$

Monte Carlo integration allows us to replace the integral $\int$ with a sum $\sum$ and the distribution $\mathbb{P}$ with data samples. I personally appreciate Monte Carlo because it makes integrations over distributions more comprehensible. It not only enables us to compute the integral for unknown distributions, but I also find the operation of summing more intuitive than integration.

6.2 Computing all coalitions, if possible

In the previous section, we discussed how to estimate the marginal contribution using Monte Carlo integration. To calculate the actual SHAP value of a feature, we need to estimate the marginal contributions for all possible coalitions.

Figure 6.2 shows all coalitions of feature values required to determine the exact SHAP value for `cat-banned`.

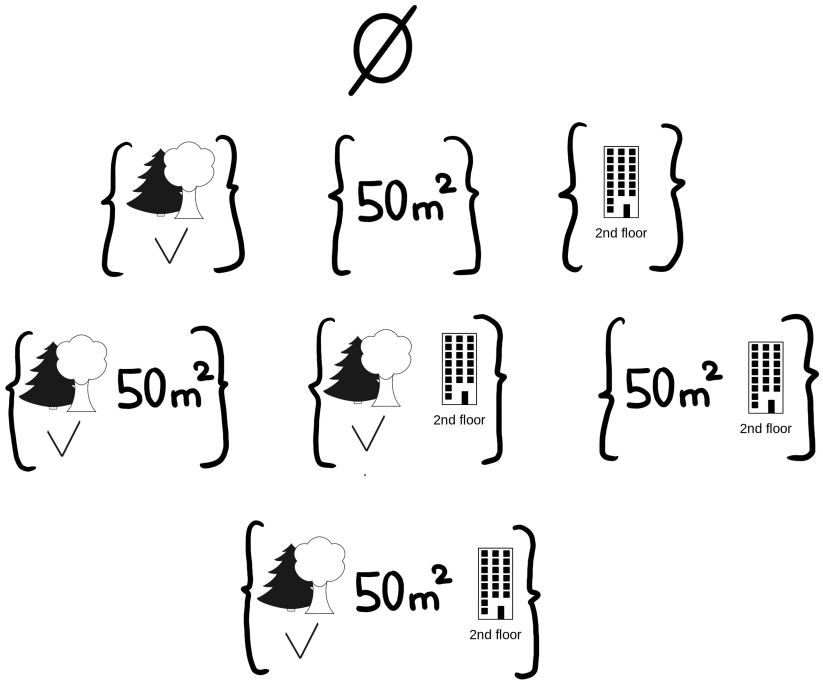


Figure 6.2: All 8 coalitions needed for computing the exact SHAP value of the `cat-banned` feature value.

In total, the following coalitions are possible:

- No feature values
- `park-nearby`
- `area-50`

- `floor-2nd`
- `park-nearby` and `area-50`
- `park-nearby` and `floor-2nd`
- `area-50` and `floor-2nd`
- `park-nearby`, `area-50`, and `floor-2nd`.

For each coalition, we calculate the predicted apartment price with and without the `cat-banned` feature value and derive the marginal contribution from the difference. The exact SHAP value is the (weighted) average of these marginal contributions. To generate a prediction from the machine learning model, we replace the feature values of features not in a coalition with random feature values from the apartment dataset. The SHAP value formula is:

$$\hat{\phi}_j^{(i)} = \sum_{S \subseteq \{1,\ldots,p\} \setminus \{j\}} \frac{|S|!\,(p - |S| - 1)!}{p!} \hat{\Delta}_{S,j}$$

6.3 Handling large numbers of coalitions

The computation time increases exponentially with the number of features due to the potential 2^p coalitions, where p is the number of features. When p is large, we must rely on estimation techniques that don't require going through all coalitions.

Two solutions exist:

- In some cases, we can utilize the structure of the model. For purely additive models, like linear regression models without interaction terms, it's sufficient to compute one marginal contribution per feature. Even for other models such as neural networks, there are model-specific estimation methods that avoid iterating through all coalitions.
- Instead of iterating through all coalitions, it's possible to sample coalitions. There are many different ways to sample coalitions, which will be discussed later.

The estimation methods vary in:

- Speed

- Accuracy, typically as a trade-off with speed
- Applicability: some estimators are model-specific

The permutation estimator is a rather flexible and fast method that we will further examine.

6.4 Estimation through permutation

Estimation through permutation works by creating a random permutation of an instance's feature values and then performing a forward and backward generation of coalitions. The best way to understand permutation estimation is through an example.

Let's consider four feature values: $x_{\text{park}}, x_{\text{cat}}, x_{\text{area}}$, and x_{floor}. For simplicity, we will denote $x^{(i)}_{\text{park}}$ as x_{park}. First, we need a random permutation. For instance:

$$(x_{\text{cat}}, x_{\text{area}}, x_{\text{park}}, x_{\text{floor}})$$

We start from the left and compute the marginal contributions:

- Adding x_{cat} to $\emptyset$
- Adding x_{area} to $\{x_{\text{cat}}\}$
- Adding x_{park} to $\{x_{\text{cat}}, x_{\text{area}}\}$
- Adding x_{floor} to $\{x_{\text{cat}}, x_{\text{area}}, x_{\text{park}}\}$

This is the forward generation. Next, we iterate backwards:

- Adding x_{floor} to $\emptyset$
- Adding x_{park} to $\{x_{\text{floor}}\}$
- Adding x_{area} to $\{x_{\text{park}}, x_{\text{floor}}\}$
- Adding x_{cat} to $\{x_{\text{area}}, x_{\text{park}}, x_{\text{floor}}\}$

This approach only alters one feature at a time, reducing the number of model calls as the first term of a marginal contribution transitions into the second term of the subsequent one. For instance, the coalition $\{x_{\text{cat}}, x_{\text{area}}\}$ is used to calculate the marginal contribution of x_{park} to $\{x_{\text{cat}}, x_{\text{area}}\}$ and of x_{area} to $\{x_{\text{cat}}\}$. We estimate the marginal contribution using Monte Carlo integration. With each

forward and backward generation for a permutation, we get marginal contributions for multiple features, not just a single one. In fact, we get two marginal contributions per feature for one permutation. By repeating the permutation sampling, we get even more marginal contributions and therefore achieve more accurate estimates. The more permutations we sample and iterate over, the more marginal contributions are estimated, bringing the final SHAP estimates closer to their theoretical value.

So, how do we transition from here, from marginal contributions based on permutations, to SHAP values? Actually, the formula is simpler than the original SHAP formula. The SHAP formula contains a complex fraction that we multiply by the sum of the marginal contributions. However, with permutation estimation, we don't sum over coalitions but over permutations. If you recall from the Theory Chapter, we justified the coalition weights by their frequency when listing all possible coalitions. But SHAP values can also be defined via permutations. Let m denote permutations of the features, with $o(k)$ being the k-th permutation, then SHAP can be estimated as follows:

$$\hat{\phi}_j^{(i)} = \frac{1}{m}\sum_{k=1}^{m}\hat{\Delta}_{o(k),j}$$

Now, let's explain $\hat{\Delta}_{o(k),j}$: We have permutation $o(k)$. In this k-th permutation, feature j occupies a particular position. Assuming $o(k)$ is $(x_{\text{cat}}, x_{\text{area}}, x_{\text{park}}, x_{\text{floor}})$ and j is park, then $\hat{\Delta}_{o(k),j} = \hat{v}(\{\text{cat, area, park}\}) - \hat{v}(\{\text{cat, area}\})$. But what is m? If we want to sum over all coalitions, then $m = p!$. However, the motivation for permutation estimation was to avoid computing all possible coalitions or permutations. The good news is that m can be a number smaller than all possible permutations, and you can use a sample of permutations with the above formula. But since we perform forward and backward iterations, the formula looks like this:

$$\hat{\phi}_j^{(i)} = \frac{1}{2m}\sum_{k=1}^{m}(\hat{\Delta}_{o(k),j} + \hat{\Delta}_{-o(k),j})$$

The 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). A simpler version would involve sampling random permutations without the forward and backward steps. One advantage of antithetic sampling is the reuse of resulting coalitions to save computation.

The permutation procedure has an additional benefit: it ensures that the efficiency axiom is always satisfied, meaning when you add up the SHAP values, they will exactly equal the prediction minus the average prediction. Estimation methods relying on sampling coalitions only satisfy the efficiency axiom in expectation. However, individual SHAP values remain estimates, and the more permutations you draw, the better these estimates will be. For a rough idea of how many permutations you might need: the `shap` package defaults to 10.

6.5 Overview of SHAP estimators

Estimation via permutation is an effective choice, particularly for tabular data. There are numerous other ways to estimate Shapley values, which are detailed in the Appendix. Here's a table to illustrate just how many methods there are. Some methods are model-specific, while others are model-agnostic. Model-specific methods are often inspired by other techniques used to explain model predictions.

Method	Estimation (with inspiration)	Model-specific?
Exact	Iterates through all background data and coalitions	Agnostic
Sampling	Samples coalitions	Agnostic
Permutation	Samples permutations	Agnostic
Linear	Exact estimation with linear model weights	Linear
Additive	Simplifies estimation based on additive nature of the model (inspired by GAMs)	GAMs
Kernel	Locally weighted regression for sampled coalitions (inspired by LIME)	Agnostic
Tree, interventional	Recursively iterates tree paths	Tree-based

Method	Estimation (with inspiration)	Model-specific?
Tree, path-dependent	Recursively iterates hybrid paths	Tree-based
Gradient	Computes the output's gradient with respect to inputs (inspired by Input Gradient)	Gradient-based
Deep	Backpropagates SHAP value through network layers (inspired by DeepLIFT)	Neural Networks
Partition	Recursive estimation based on feature hierarchy (inspired by Owen values)	Agnostic

There are more estimation methods than those listed here. The selection shown is based on the estimation methods available in the Python `shap` package.

6.6 From estimators to explainers

The estimation methods mentioned earlier are implemented in the Python package `shap`, which we will use for the code examples. This section concludes the theory portion, except for the appendix. We will now discuss the implementation choices of estimators in `shap`, which will give us a good understanding of the utility of different estimation strategies. In `shap`, the various estimation methods are implemented as objects known as `Explainer`, such as the `Linear` explainer for linear models. When using `shap`, you rarely need to concern yourself with explainers, as the default setting is 'auto', meaning the package will automatically select the best option.

To illustrate how the 'auto' option selects estimation methods:

- If possible, the 'auto' option selects a model-specific version, like the tree explainer for a random forest model, but only in the cases of linear, additive, and tree-based models.
- If there are 10 or fewer features, the exact explainer is used.
- For models with more features, the permutation explainer is employed.
- If the model inputs are images or text data, the partition explainer is typically used.

Tip

The original SHAP paper (Lundberg and Lee 2017b) introduced the Kernel method, which involves sampling coalitions and using a weighted linear regression model to estimate SHAP values. The Kernel method "united" SHAP with LIME and other prediction explanation techniques in machine learning. However, the Kernel method is slow and has been superseded by the permutation method.

7 SHAP for Linear Models

By the end of this chapter, you will be able to:

- Calculate SHAP for linear models.
- Install and use `shap`.
- Interpret and visualize SHAP values.
- Analyze the model with summary and dependence plots.

Let's start with a straightforward example. This example is "simple" because we are using a linear regression model, which is naturally interpretable, or so they say. This allows us to compare SHAP values with the coefficients of the model.

7.1 The wine data

We will utilize the wine dataset from the UCI repository. The objective is to predict wine quality based on its physicochemical properties. The target variable "quality" is an average rating from three blind testers, ranging from 0 to 10.

Let's first examine the features in the data by downloading it.

```
import pandas as pd
# Set the file URL and filename
url = 'https://archive.ics.uci.edu/ml/' \
      'machine-learning-databases/' \
      'wine-quality/winequality-white.csv'
file_name = 'wine.csv'

# Check if the file exists in the current directory
try:
```

```
    wine = pd.read_csv(file_name)
except FileNotFoundError:
    print(f'Downloading {file_name} from {url}...')
    wine = pd.read_csv(url,  sep=";")
    wine.to_csv(file_name, index=False)
    print('Download complete!')
```

Let's analyze the distributions of the features.

```
from tabulate import tabulate
summary = wine.describe().transpose().round(2)
summary = summary.drop("count", axis=1)
# Create a markdown table
markdown_table = tabulate(
  summary, headers='keys', tablefmt='pipe'
)
print(markdown_table)
```

	mean	std	min	25%	50%	75%	max
fixed acidity	6.85	0.84	3.8	6.3	6.8	7.3	14.2
volatile acidity	0.28	0.1	0.08	0.21	0.26	0.32	1.1
citric acid	0.33	0.12	0	0.27	0.32	0.39	1.66
residual sugar	6.39	5.07	0.6	1.7	5.2	9.9	65.8
chlorides	0.05	0.02	0.01	0.04	0.04	0.05	0.35
free sulfur dioxide	35.31	17.01	2	23	34	46	289
total sulfur dioxide	138.36	42.5	9	108	134	167	440
density	0.99	0	0.99	0.99	0.99	1	1.04
pH	3.19	0.15	2.72	3.09	3.18	3.28	3.82
sulphates	0.49	0.11	0.22	0.41	0.47	0.55	1.08
alcohol	10.51	1.23	8	9.5	10.4	11.4	14.2
quality	5.88	0.89	3	5	6	6	9

As observed, the highest quality is 9 (out of 10), and the lowest is 3. The other features have varying scales, but this is not an issue for SHAP values, as they explain the prediction on the outcome's scale.

7.2 Fitting a linear regression model

With the wine dataset in our hands, we aim to predict the quality of a wine based on its physicochemical features. A linear model for one data instance is represented as:

$$f(x^{(i)}) = \beta_0 + \beta_1 x_1^{(i)} + ... + \beta_p x_p^{(i)}$$

where $x^{(i)}$ is the instance for which we want to compute the contributions. Each $x_j^{(i)}$ is a feature value, with $j = 1, ..., p$. The β_j is the weight in the linear regression model corresponding to feature j.

Before fitting the linear model, let's divide the data into training and test sets.

```
from sklearn.model_selection import train_test_split

# Extract the target variable (wine quality) from the data
y = wine['quality']
X = wine.drop('quality', axis=1)

X_train, X_test, y_train, y_test = train_test_split(
  X, y, test_size=0.2, random_state=42
)
```

We'll now train the linear regression model using the scikit-learn package.

> **Tip**
>
> `shap` can be used with all sklearn models.

```
from sklearn.linear_model import LinearRegression

model = LinearRegression()
model.fit(X_train, y_train)
```

How does the model perform? To evaluate, we calculate the mean absolute error (MAE) on the test data.

```
from sklearn.metrics import mean_absolute_error
y_pred = model.predict(X_test)
mae = mean_absolute_error(y_test, y_pred)
print(f"MAE: {mae:.2f}")
```

```
MAE: 0.59
```

This indicates that, on average, the prediction deviates by 0.59 from the actual value.

Next, we aim to understand how the model generates predictions. How is the predicted quality of a given wine related to its input features?

7.3 Interpreting the coefficients

First, let's look at the coefficients and later compare them with the corresponding SHAP values.

```
import numpy as np
coefs = pd.DataFrame({
  'feature': X.columns.values,
  'coefficient': np.round(model.coef_, 3)
})
print(coefs.to_markdown(index=False))
```

feature	coefficient
fixed acidity	0.046
volatile acidity	-1.915
citric acid	-0.061
residual sugar	0.071
chlorides	-0.026
free sulfur dioxide	0.005
total sulfur dioxide	-0
density	-124.264

feature	coefficient
pH	0.601
sulphates	0.649
alcohol	0.229

Interpretation:

- For instance, increasing the fixed acidity of a wine by 1 unit raises the predicted quality by 0.046.
- Increasing the density by 1 unit reduces the predicted quality by 124.264.
- Volatile acidity, citric acid, chlorides, total sulfur dioxide, and density negatively affect the predicted quality.

7.4 Model coefficients provide a global perspective

The coefficient for a feature is constant across all data points, rendering the coefficient a "global" model interpretation.

For a specific data point (a local explanation), we can also consider the feature value. Consider a coefficient $\beta_j = 0.5$. The feature value could be 0, 0.008, or -4, yielding different results when multiplied with the feature value (0, 0.004, -2). A more meaningful way to present a feature effect would be the product of the coefficient and the feature value: $\beta_j \cdot x_j^{(i)}$.

However, this formula is also "incomplete" as it doesn't provide an understanding of whether the contribution is large (in absolute terms) or not. Suppose the contribution is $\beta_j \cdot x_j^{(i)} = 2$, with $\beta_j = 0.5$ and $x_j^{(i)} = 4$. Is that a large contribution? It depends on the range of x_j. If 4 is the smallest possible value of $x_j^{(i)}$, then 2 is actually the smallest possible contribution, but if $x_j^{(i)}$ follows a Normal distribution centered at 0, then a contribution of 4 is more than expected. An easy solution is to center the effect $\beta_j x_j^{(i)}$ around the expected effect $E(\beta_j X_j)$. As we'll see in the coming section, this is the same as how the SHAP values are defined.

7.5 Theory: SHAP for linear models

For linear regression models without interaction terms, the computation of SHAP values is straightforward since the model only has linear relations between the features and target and no interactions. The SHAP value $\phi_j^{(i)}$ of the j-th feature on the prediction $f(x_j^{(i)})$ for a linear regression model is defined as:

$$\phi_j^{(i)} = \beta_j x_j^{(i)} - \mathbb{E}(\beta_j X_j) = \beta_j (x_j^{(i)} - \mathbb{E}(X_j))$$

Where $\mathbb{E}(\beta_j X_j)$ denotes the expected effect estimate for feature j, the contribution refers to the difference between the feature effect and the average effect. Here's why SHAP has such a simplified form for linear models: When calculating the marginal contribution $v(S \cup \{j\}) - v(S)$, all components of the linear function cancel out, leaving only those associated with feature j. In $v(S \cup \{j\})$, only $\beta_j x_j^{(i)}$ remains, while in $v(S)$, only $\beta_j \mathbb{E}(X_j)$ persists. All marginal contributions remain constant due to the absence of feature interactions, hence, the coalition to which we add feature j is irrelevant.

We simply replace the expectation with the mean for the estimation:

$$\phi_j^{(i)} = \beta_j \left(x_j^{(i)} - \frac{1}{n} \sum_{k=1}^{n} (x_j^{(k)}) \right)$$

Fantastic! Now we are aware of each feature's contribution to the prediction. Let's verify the efficiency axiom:

$$\begin{aligned}
\sum_{j=1}^{p} \phi_j^{(i)} &= \sum_{j=1}^{p} (\beta_j x_j^{(i)} - \mathbb{E}(\beta_j X_j)) \\
&= \beta_0 + \sum_{j=1}^{p} \beta_j x_j^{(i)} - (\beta_0 + \sum_{j=1}^{p} \mathbb{E}(\beta_j X_j)) \\
&= f(x) - \mathbb{E}(f(X))
\end{aligned}$$

This is the predicted value for data point x subtracted from the average predicted value. Feature contributions can be negative. Now, let's apply these to the wine quality prediction.

7.6 Installing `shap`

Although the computation of SHAP for linear models is straightforward enough to implement on our own, we'll take the easier route and install the `shap` library, which also provides extensive plotting functions and other utilities.

> **i** `shap` library
>
> The `shap` library was developed by Scott Lundberg, the author of the SHAP paper (Lundberg and Lee 2017b) and many other SHAP-related papers. The initial commit[a] was made on November 22nd, 2016. At the time of writing, the library has over 2000 commits. `shap` is open-source and hosted on Github, allowing public access and tracking of its progress. The repository has received over 19k stars and almost 3k forks. In terms of features, it's the most comprehensive library available for SHAP values. I believe that the `shap` library is the most widely-used implementation of SHAP values in machine learning.
> You can find the `shap` repository at: https://github.com/slundberg/shap
>
> [a] https://github.com/slundberg/shap/tree/7673c7d0e147c1f9d3942b32ca2c0ba93fd37875

Like most Python packages, you can install `shap` using `pip`.

```
pip install shap
```

All examples in this book utilize `shap` version 0.42.0. To install this exact version, execute the following command:

```
pip install shap==0.42.0
```

If you use virtualenv

If you're using virtualenv or venv, activate the environment first. Assuming the environment is named venv:

```
source venv/bin/activate
pip install shap
```

If you use conda

If you're using conda, use the following commands to install `shap`:

```
conda install -c conda-forge shap
```

For the version used in this book:

```
conda install -c conda-forge shap=0.42.0
```

7.7 Computing SHAP values

To better comprehend the effects of features, we can calculate SHAP values for a single data instance. For this, we construct a `LinearExplainer` object.

```
import shap
explainer = shap.LinearExplainer(model, X_train)
```

> **Note**
>
> While the model here is a `LinearRegression` model from the `sklearn` library, `shap` works with any model from `sklearn` as well as with other libraries such as `xgboost` and `lightgbm`. `shap` also works with custom prediction functions, so it's quite flexible!

Alternatively, we could utilize the `Explainer` as follows:

```
shap.Explainer(model, X_train)
```

```
<shap.explainers._linear.Linear at 0x28b6a2ce0>
```

This also creates a `Linear` explainer object. The advantage of the `algorithm='auto'` option, which is the default setting when creating an Explainer, is that `shap` identifies the model as a linear regression model and selects the efficient linear explainer.

Another method involves directly choosing the appropriate `algorithm` in the explainer:

```
shap.Explainer(model, X_train, algorithm='linear')
```

```
<shap.explainers._linear.Linear at 0x28b6a2b00>
```

To ultimately calculate SHAP values, we call the explainer with the data to be explained.

```
shap_values = explainer(X_test)
```

> **Note**
>
> When constructing a prediction model, you divide the data into training and testing sets to prevent overfitting and to achieve a fair evaluation. Although the risk of overfitting doesn't apply in the same way to SHAP, it's considered best practice to use the training data for the Explainer (i.e., for the background data) and compute explanations for new data. This separation prevents a data point's feature values from being "replaced" by its own values. It also means we calculate explanations for fresh data that the model hasn't previously encountered. However, I must confess that I haven't seen much research on dataset choices, so take this information with a pinch of salt.

Next, let's examine the SHAP values.

While initially looking a bit messy, it's insightful to inspect the `Explanation` object. The Explanation object includes `.values`, `.base_values`, and `.data` fields. The `.values` represent the SHAP values as an $n \times p$ array, `.base_values` is the average prediction (consistent for each data point), and `.data` contains the feature values. Each element in these arrays corresponds to one data point in `X_test`, which we used to calculate the SHAP values.

```
print(shap_values.values)
```

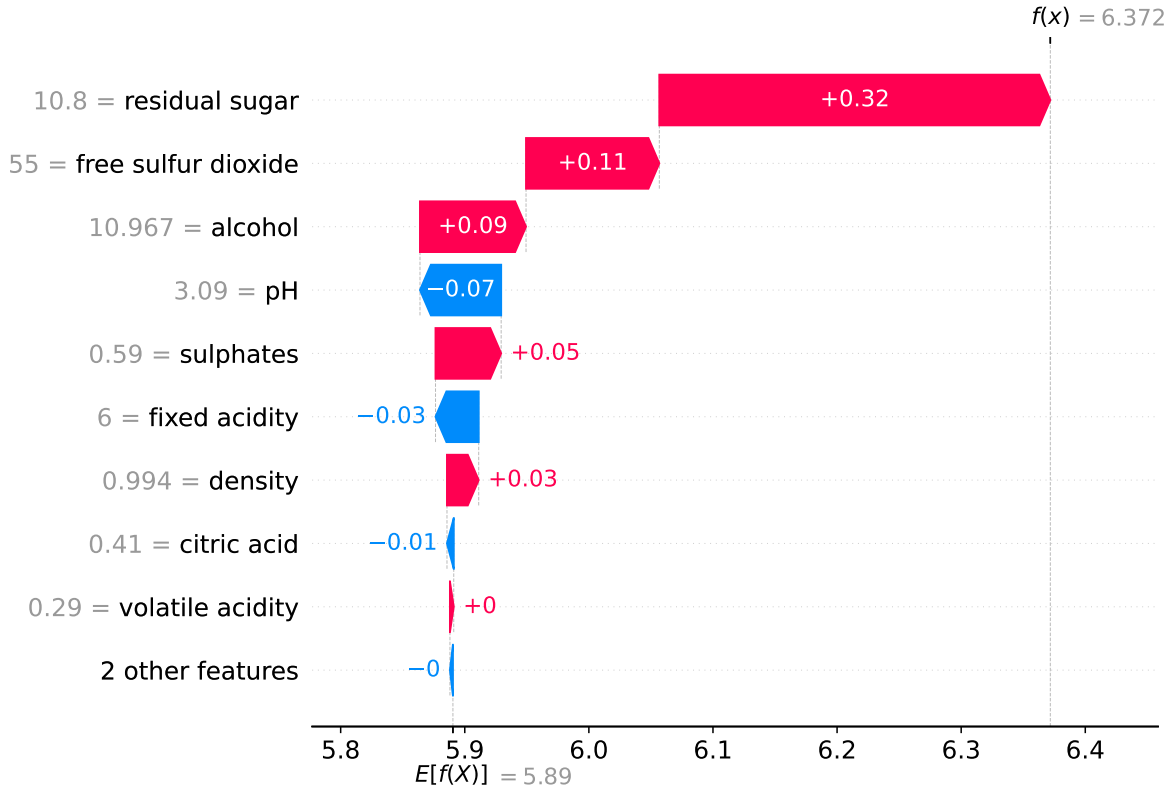
```
[[-0.03479769  0.00306381 -0.00545601 ... -0.06541621  0.05289943
   0.08545841]
 [-0.06234203 -0.45650842  0.00986986 ...  0.00066077  0.01395506
   0.59691113]
 [ 0.01570028  0.07965919 -0.00422994 ...  0.04871676 -0.05095221
   0.36790245]
 ...
 [-0.03938841  0.06051034  0.00680469 ...  0.04871676 -0.05095221
  -0.250421  ]
 [ 0.03406317  0.00306381  0.00067434 ... -0.07142321  0.02044579
  -0.29622273]
 [-0.00266262  0.13710572 -0.00422994 ...  0.18087073 -0.0249893
  -0.13591665]]
```

However, having only the raw SHAP values isn't particularly useful. The true power of the `shap` library lies in its various visualization capabilities.

7.8 Interpreting SHAP values

We have yet to see the SHAP values. So, let's visualize the SHAP values for the first data instance:

```
shap.plots.waterfall(shap_values[0])
```



The Waterfall Plot

- The plot is dubbed "waterfall" because each step resembles flowing water. Water can flow in either direction, just as SHAP values can be positive or negative. Positive SHAP values point to the right.
- The y-axis exhibits the individual features, along with the values for the selected data instance.
- The feature values are ordered by the magnitudes of their SHAP values.
- The x-axis is on the scale of SHAP values.
- Each bar signifies the SHAP value for that specific feature value.
- The x-axis also shows the estimated expected prediction $\mathbb{E}(f(X))$ and the actual prediction of the instance $f(x^{(i)})$.
- The bars start at the bottom from the expected prediction and add up to the actual prediction.

Interpretation: The predicted value of 6.37 for instance 0 differs from the average prediction of 5.89 by 0.48.

- residual sugar=10.8 contributed 0.32
- free sulfur dioxide=55.0 contributed 0.11
- alcohol=10.97 contributed 0.09
- …

The sum of all SHAP values equals the difference between the prediction (6.37) and the expected value (5.89).

> Interpretation Template *(replace [] with your data)*
>
> Prediction [$f(x)$] for instance [i] differs from the average prediction [$\mathbb{E}(f(X))$] by [$f(xi) - \mathbb{E}(f(X))j$] to which [feature name = feature value] contributed [$\phi_j^{(i)}$].

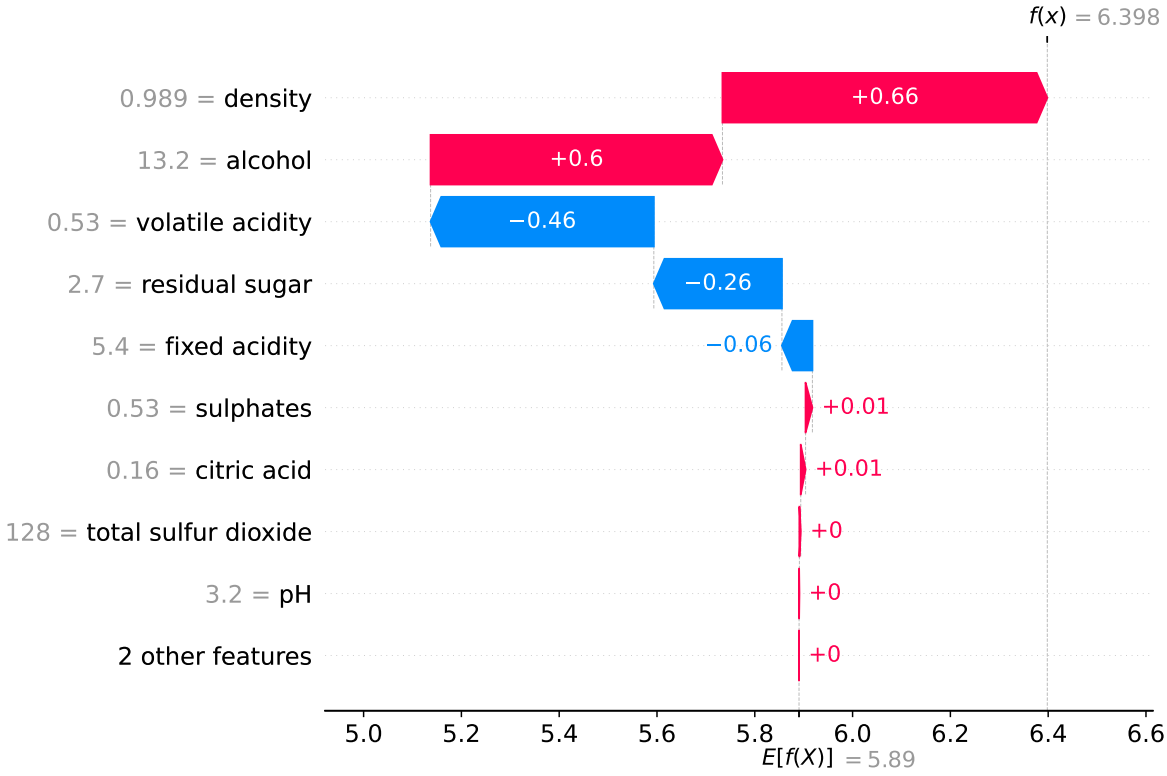
Here's an interesting analogy to view SHAP values: Imagine the prediction as an object floating in a 1-dimensional universe. Its starting point is the average prediction of the background data – this is our "center of gravity". Each feature of the data point can be seen as a force acting on this object and can either push it up or down. The SHAP value describes how strong each force is and in which direction it pushes. Eventually, these forces reach equilibrium, which represents the predicted value.

Back from this little universe to our example, several observations can be made:

- The most influential feature was 'residual sugar' (=10.8), with a SHAP value of 0.32, indicating it had an increasing impact on the quality on average.
- Overall, the prediction surpassed the average, suggesting a high-quality wine.
- Most of this wine's feature values were assigned a positive SHAP value.
- The feature 'pH' with a value of 3.09 had the largest negative SHAP value.

Let's examine another data point:

```
shap.waterfall_plot(shap_values[1])
```



This wine has a similar predicted rating to the previous one, but the contributions to this prediction differ. It has two substantial positive contributions from the 'density' and 'alcohol' values, but also two strong negative factors: 'volatile acidity' and 'residual sugar'.

The waterfall plot lacks context for interpretation. For instance, while we know 'residual sugar' increased the prediction for the first wine, we cannot deduce from the waterfall plot alone whether low or high levels of 'residual sugar' are associated with small or large SHAP values.

7.9 Global model understanding

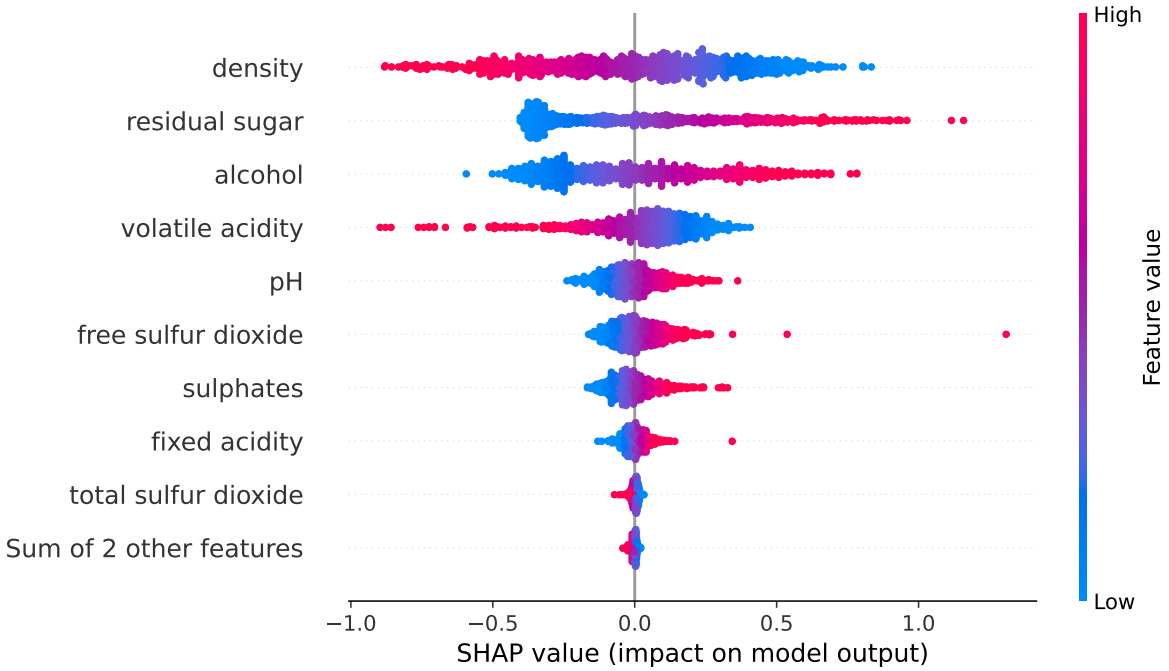
We computed SHAP values to explain individual predictions. However, we can also compute SHAP values for more data points, ideally for the entire (test)

dataset (and training data as background data). By visualizing the SHAP values across all features and multiple data points, we can uncover patterns of how the model makes predictions. This gives us a global model interpretation.

> **Global versus local interpretation**
>
> SHAP values allow for a local interpretation – how features contribute to a prediction. Global interpretations focus on *average model behavior*, which includes how features affect the prediction, how important each feature is for the prediction, and how features interact.

We previously computed the SHAP values for the test data, which are now stored in the `shap_values` variable. We can create a summary plot from this variable for further insights into the model.

```
shap.plots.beeswarm(shap_values)
```



Summary plot

- Also known as beeswarm plot.
- The x-axis represents the SHAP values, while the y-axis shows the features, and the color indicates the feature's value.
- Each row corresponds to a feature.
- The feature order is determined by importance, defined as the average of absolute SHAP values: $I_j = \frac{1}{n} \sum_{i=1}^{n} \phi_j^{(i)}$
- Each dot represents the SHAP value of a feature for a data point, resulting in a total of $p \cdot n$ dots.

The summary plot automatically ranks features based on importance. Density, residual sugar, and alcohol are the most important features for predicting wine quality, according to SHAP values.

The coloring also reveals that the relationships are monotonic for all features: A feature's increase (or decrease) consistently influences the prediction in one direction. Since the model is a linear regression model, the modeled relationship between each input feature and the target must be linear. That means a 1 unit increase in a feature always increases the prediction by β_j, the corresponding coefficient from the linear model. However, that's just true for the linear model. The "true" relationship in the data might be non-linear.

The coloring for each feature in the summary plot reveals the direction of the effect a feature has: For example, higher density wines are associated with lower SHAP values, while wines with more residual sugar have a higher corresponding SHAP value. Again, this information directly corresponds to the coefficients from the linear regression model. In later chapters we will see examples with more insightful summary plots.

How to interpret the summary plot

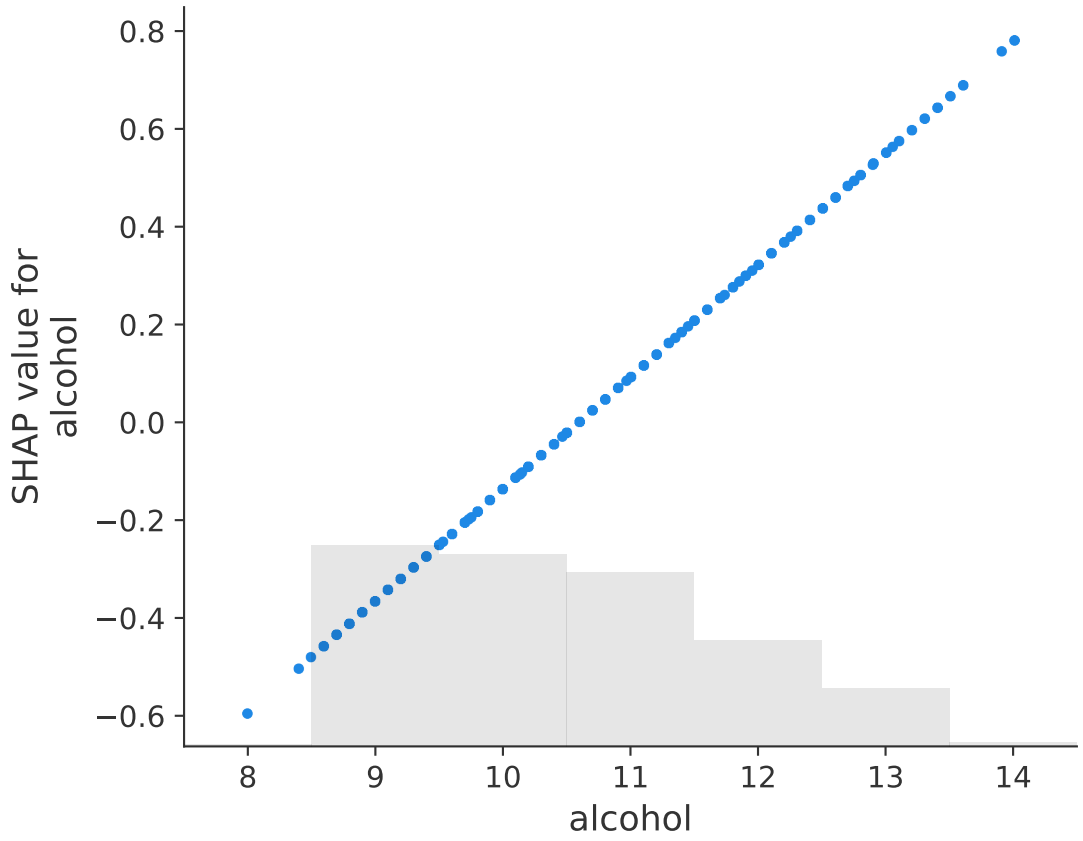
- Observe the ranking of the features. The higher the feature, the greater its SHAP importance.
- For each feature of interest:
  - Examine the distribution of the SHAP values. This provides insight into the various ways the feature values can influence the prediction. For instance, a wide spread indicates a broad range

of influence.
- Understand the color trend for a feature: This offers an initial insight into the direction of a feature effect and whether the relationship is monotonic or exhibits a more complex pattern.
- Look for color clusters that may indicate interesting data clusters. Not relevant for linear models, but for non-linear ones.

7.10 Comparison between coefficients and SHAP values

Now, we'll explore a new type of plot - the SHAP dependence plot. The dependence plot should confirm that the SHAP values also exhibit a linear relationship with the target for features known to have a linear relation.

```
shap.plots.scatter(shap_values[:, 'alcohol'])
```



The dependence plot

- Also referred to as scatter plot.
- Mathematically, the plot contains these points: $\{(x_j^{(i)}, \phi_j^{(i)})\}_{i=1}^n$.
- The x-axis represents the feature value, and the y-axis represents the SHAP value.
- Highlighting feature interactions on the dependence plot can enhance its effectiveness.
- The dependence plot is similar to the summary plot for a single feature, but instead of using color to represent the feature value, these values are distributed across the x-axis.
- The grey histogram indicates the distribution of the feature values. Ranges with little data should be interpreted more cautiously.

This plot demonstrates the global dependence modeled by the linear regression between alcohol and the corresponding SHAP values for alcohol. The dependence plot will be much more insightful for a non-linear model, but it's a great way to confirm that the SHAP values reflect the linear relationship in the case of a linear regression model. As the alcohol content increases, the corresponding SHAP value also increases linearly. This increase corresponds to the slope in the linear regression model:

```
feature = 'alcohol'
ind = X_test.columns.get_loc(feature)
coefs.coefficient[ind]
```

```
0.229
```

A visual inspection of the dependence plot confirms the same slope, as the plot ranges from (8, -0.6) to (14, 0.8), resulting in a slope of $(0.8-(-0.6))/(14-8) \approx 0.23$.

8 Classification with Logistic Regression

By the end of this chapter, you will be able to:

- Interpret SHAP for classification models.
- Decide whether to interpret probabilities or log odds.
- Handle categorical features.
- Use alternatives to the waterfall plot: bar and force plot.
- Describe the cluster plot and the heatmap plot.

This chapter explains how to use and interpret SHAP with logistic regression. Differences from linear regression include:

- Two outcomes instead of one.
- The outcome is either a probability [1] or log odds.
- A non-linear function when the output is a probability.

Logistic regression is suitable for binary classification tasks. It provides probability outputs for two classes by relating the binary output with the features in the following way:

$$P(Y = 1|x^{(i)}) = \frac{1}{1 + \exp(-(\beta_0 + \beta_1 x_1^{(i)} + \beta_2 x_2^{(i)} + ... + \beta_p x_p^{(i)}))}$$

Since the probability of one class defines the other's, you can work with just one probability. Having two classes is a special case of having k classes.

[1] While the output is a number between 0 and 1, classifiers are frequently not well-calibrated, so be cautious when interpreting the output as a probability in real-world scenarios.

> **Tip**
>
> Even though the example here is binary classification, `shap` works the same for multi-class and also when the model is not logistic regression.

From the SHAP perspective, classification is similar to regression, except that the outcome is a score or class probability.

8.1 The Adult dataset

We will use the Adult dataset from the UCI repository for the classification task. This dataset contains demographic and socioeconomic data of individuals from the 1994 U.S. Census Bureau database, aiming to predict whether an individual's income is greater than or equal to $50,000 per year. The dataset includes features such as age, education level, work class, occupation, and marital status. With approximately 32,000 observations, it contains both categorical and numerical features. Conveniently, the `shap` package includes the Adult dataset, which simplifies its use in our example.

8.2 Training the model

First, we load the dataset which, to our convenience, is available in `shap`.

```
import shap
from sklearn.model_selection import train_test_split

X, y = shap.datasets.adult()

X_train, X_test, y_train, y_test = train_test_split(
  X, y, test_size=0.2, random_state=1
)
```

Next, we train the model and compute the SHAP values. Compared to the linear regression example, you will notice something new here:

```
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import OneHotEncoder, StandardScaler
from sklearn.linear_model import LogisticRegression
import numpy as np

# Define the categorical and numerical features
cats = ['Workclass', 'Marital Status', 'Occupation',
        'Relationship', 'Race', 'Sex', 'Country']
nums = ['Age', 'Education-Num', 'Capital Gain',
        'Capital Loss', 'Hours per week']

# Define the column transformer
preprocessor = ColumnTransformer(
    transformers=[
        ('cat', OneHotEncoder(), cats),
        ('num', StandardScaler(), nums)
    ])

# Define the pipeline
model = Pipeline([
    ('preprocessor', preprocessor),
    ('classifier', LogisticRegression(max_iter=10000))
])

# Fit the pipeline to the training data
model.fit(X_train, y_train)

X_sub = shap.sample(X_train, 100)

ex = shap.Explainer(model.predict_proba, X_sub)
shap_values = ex(X_test.iloc[0:100])
```

The novelties include:

- Using a subset of the training data as a background dataset improves computation speed at the cost of less accurate SHAP value estimates. In the

Regression Chapter we'll discuss more elaborate choices of background data in depth.
- Applying SHAP not to the model but to the entire pipeline allows us to compute SHAP values for the original features instead of their processed versions.

The Adult dataset contains both categorical and numerical features, which are transformed before being inputted into the logistic regression model.

- Numerical features are standardized: $x^{(i)}_{j,std} = (x^{(i)}_j - \bar{x}_j)/sd(x_j)$.
- Categorical features are one-hot encoded. For instance, a feature with 1 column and 3 categories transforms into 3 columns, e.g. category "3" might be encoded as (0,0,1).

Following these steps, our dataset expands to approximately 470 columns. The numerical features, like age, are no longer easily interpretable due to standardization, making it necessary to compute the actual age represented by, say, 0.8.

The logistic regression model, however, can process these inputs. This implies that the coefficients are based on this transformed dataset. Applying SHAP values directly on the logistic regression model would yield 470 SHAP values.

Yet, there's a more interpretable method: We can integrate the preprocessing and logistic regression into a pipeline and regard it as our model. This approach is akin to nesting mathematical functions:

- Our model is $y = f(\tilde{x})$, where $\tilde{x}$ is the preprocessed data.
- We have our preprocessing, denoted as g, so the preprocessed data can be expressed as $\tilde{x} = g(x)$.
- A less desirable option would be to apply SHAP values on f.
- A better choice is to define a new function $\tilde{f}(x) = f(g(x))$, where the input is x and not $\tilde{x}$. $\tilde{f}$ represents the pipeline.
- We apply and interpret SHAP values for $\tilde{f}$ instead of f.

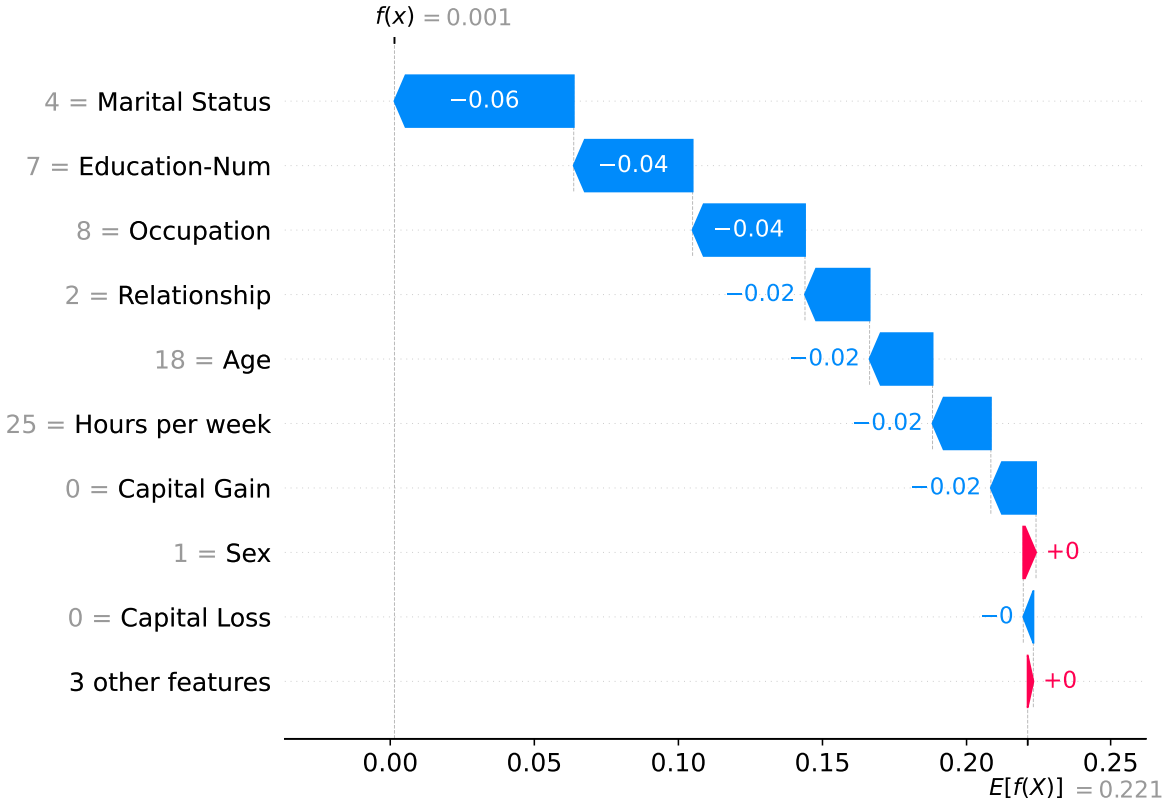
This method facilitates the interpretation of features in their original form.

> **Tip**
>
> When preprocessing your data, think about which steps you want to incorporate into your pipeline when calculating SHAP values. It's sensible to include

steps like feature standardization in the pipeline, while transformations that enhance interpretability should be left out.

Another point of interest is that the model has two outputs: the probability of earning less than \$50k and the probability of earning more than \$50k. This is mirrored in the resulting `shap_values` variable, which gains an extra dimension. So to pick a single SHAP value, we have define 3 things: For which data instance? For which feature? For which model output?

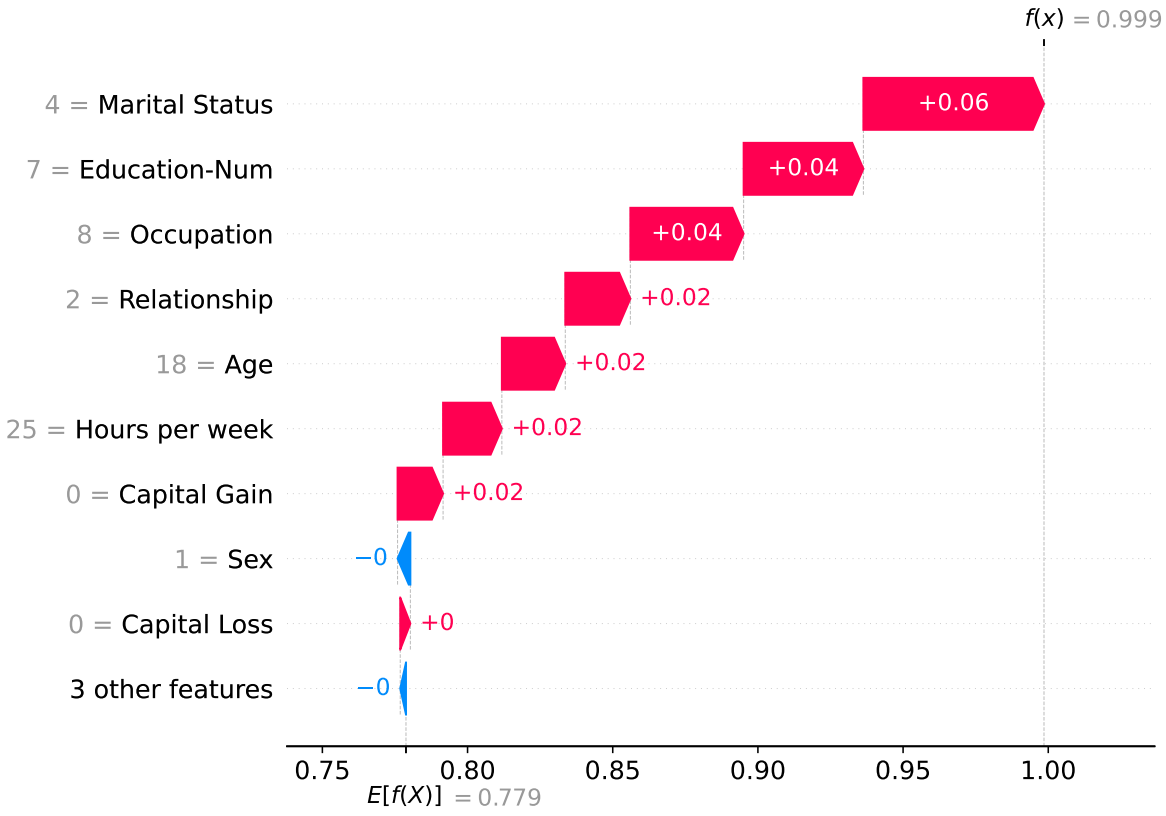
```
class_index = 1
data_index = 1

shap.plots.waterfall(shap_values[data_index,:,class_index])
```



For this individual, the predicted likelihood of earning more than $50k was 0.01%, well below the expected 22%. This plot shows that Marital Status was the most influential feature, contributing -0.05. The interpretation is largely the same as for regression, except that the outcome is on the probability level and we have to choose which class's SHAP values we want to interpret.

Now, let's inspect the SHAP values for the alternative class.

```
class_index = 0
shap.plots.waterfall(shap_values[data_index,:,class_index])
```



This plot shows the probability of this individual earning less than $50k. We can see it's the same figure from before except all SHAP values are multiplied

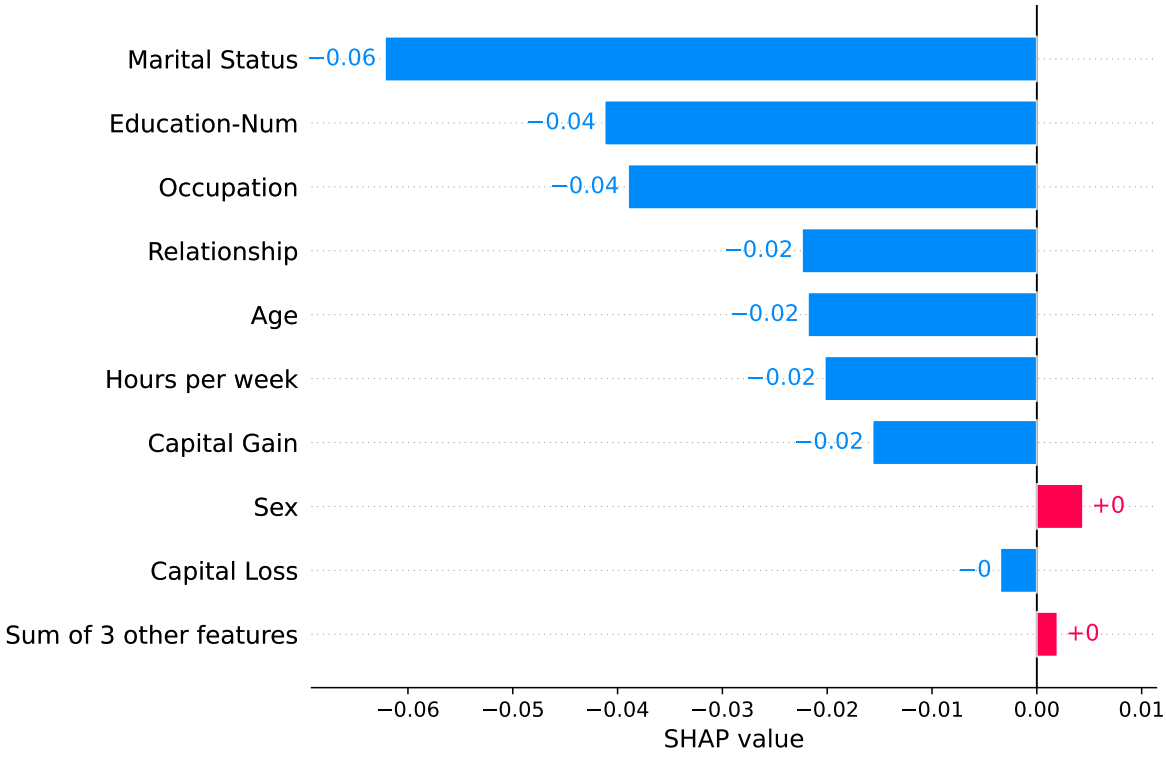
by -1. This is logical since the probabilities for both classes must add up to 1. Thus, a variable that increases the classification by 0.11 for class >50k decreases the classification by 0.11 for class <=50k. We only need to pick one of the two classes. This changes when we have three or more classes, see the Image Chapter for an example involving multiple classes.

8.3 Alternatives to the waterfall plot

The waterfall plot visualizes the SHAP values of a data instance. However, there are other ways to visualize the exact same type of information: the bar plot and the force plot. If I were to arrange the three plots on a spectrum, I'd say that the bar plot is the most conventional, familiar to most people, followed by the waterfall plot and the force plot, which is the most challenging to read.

Let's begin with the bar plot.

```
# First, reset class_index
class_index=1
shap.plots.bar(shap_values[data_index,:,class_index])
```



The interpretation here is the same as that of the waterfall plot, so I will not repeat it. The only difference between the two plots is the arrangement of information, with the bar plot lacking in the presentation of $\mathbb{E}(f(X))$ and $f(x^{(i)})$.

Additionally, the force plot is simply a different representation of the SHAP values:

```
shap.initjs()
shap.plots.force(shap_values[data_index,:,class_index])
```

The force plot is interactive, based on JavaScript, and allows you to hover over it for more insights. Of course, this feature is not available in the static format you're currently viewing, but it can be accessed if you create your own plot and embed it in a Jupyter notebook or a website. The image above is a screenshot of a force plot. The plot is named force plot because the SHAP values are depicted as forces, represented by arrows, which can either increase or decrease

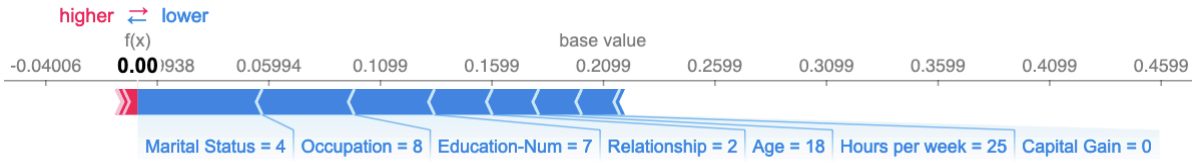


Figure 8.1: Force Plot

the prediction. If you compare it with the waterfall plot, it's like a horizontal arrangement of arrows.

Personally, I find the waterfall plot easier to read than the force plot, and it provides more information than the bar plot.

8.4 Interpreting log odds

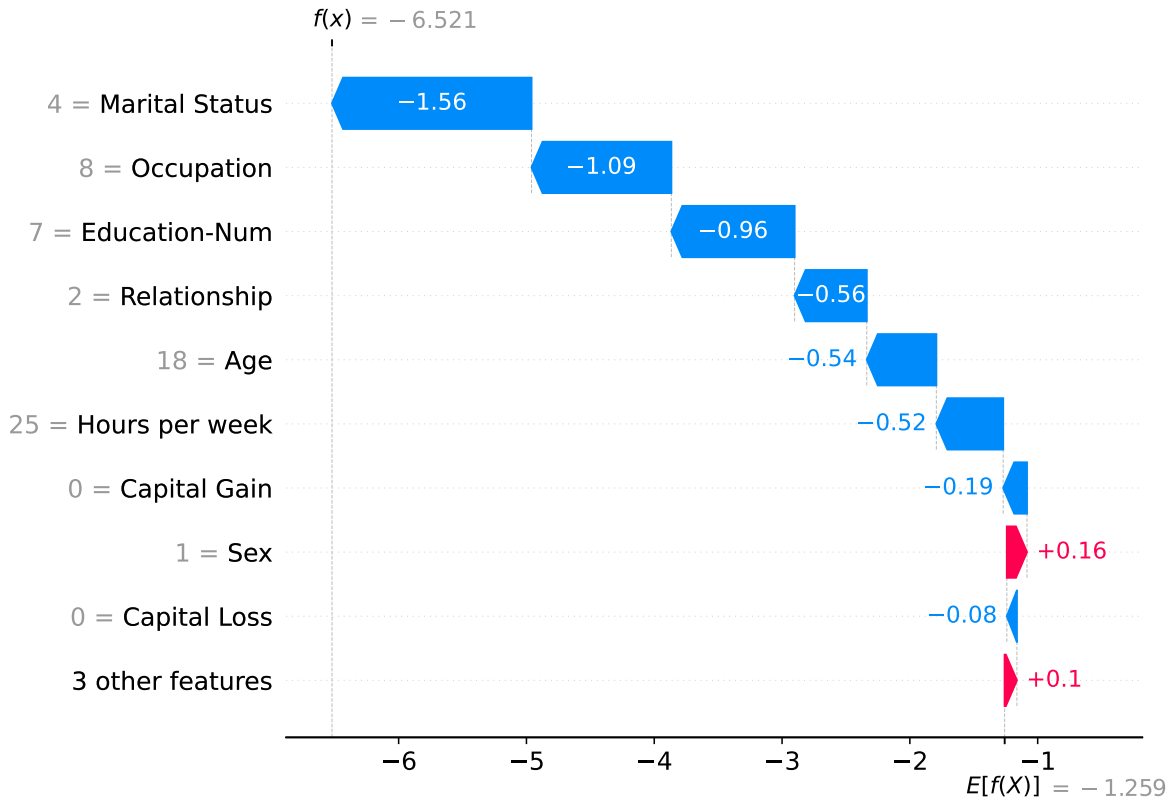
In logistic regression, it's common to interpret the model in terms of log odds rather than probability.

The explainer has a `link` argument for this, which defaults to the identity link $l(x) = x$. For classification, the logit link: $l(x) = log\left(\frac{x}{1-x}\right)$, is a good choice. This turns the probabilities into log odds. In mathematical terms, it's a rearrangement of the terms in the logistic regression model:

$$\log\left(\frac{P(Y=1|x^{(i)})}{P(Y=0|x^{(i)})}\right) = \beta_0 + \beta_1 x_1^{(i)} + \beta_2 x_2^{(i)} + ... + \beta_p x_p^{(i)}$$

We can use this logit link to transform the output of the logistic regression model and compute SHAP values on this new scale:

```
ex_logit = shap.Explainer(
  model.predict_proba, X_sub, link=shap.links.logit
)
sv_logit = ex_logit(X_test.iloc[0:100])

shap.plots.waterfall(sv_logit[data_index,:,class_index])
```



When the outcome of a logistic regression model is defined in terms of log odds, the features impact the outcome linearly. In other words, logistic regression is a linear model on the level of the log odds.

Here's what it means for interpretation: A marital status of 4 contributes -1.56 to the log odds of making >$50k versus <=$50k compared to the average prediction. However, SHAP values shine in their applicability at the probability level, and log odds can be challenging to interpret. So, when should you use log odds and when should you use probabilities?

> **Note**
>
> If your focus is on the probability outcome, use the identity link (which is the default behavior). The logit space is more suitable if you're interested in "evidence" in an information-theoretic sense, even if the effect in probability

space isn't substantial.

Let's discuss when the distinction between log odds and probabilities matters: A shift from 80% to 90% is large in probability space, while a change from 98% to 99.9% is relatively minor.

In probability space, the differences are 0.10 and 0.019. In logit space, we have:

- $log(0.9/0.1) - log(0.8/0.2) \approx 0.8$ and
- $log(0.999/0.001) - log(0.98/0.02) \approx 3$.

In logit space, the second jump is larger. This happens because the logit compresses near 0 and 1, making changes in the extremes of probability space appear larger.

So, which one should you select? If you're primarily concerned with probabilities, and a jump from 80% to 81% is as significant to you as from 99% to 100%, stick with the default and use the identity link. However, if changes in extreme probabilities near 0 and 1 are more critical for your application, choose logits. Whenever rare events, anomalies, and extreme probabilities matter, go with logits. You can also visualize the difference in step sizes in the following Figure 8.2.

8.5 Understanding the data globally

To finish up the model interpretation, let's have a look at the global importances and effects with the summary plot. Here we interpret the model in the probability space again.

```
shap.plots.beeswarm(shap_values[:,:,class_index])
```

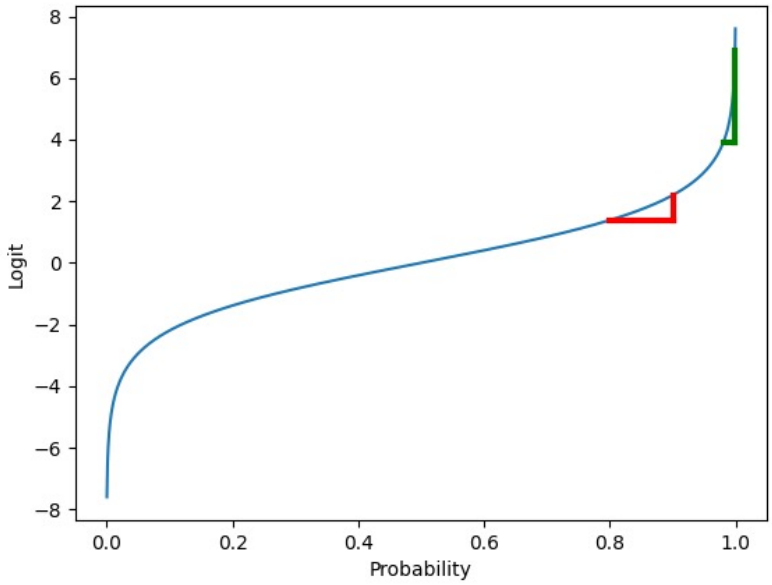
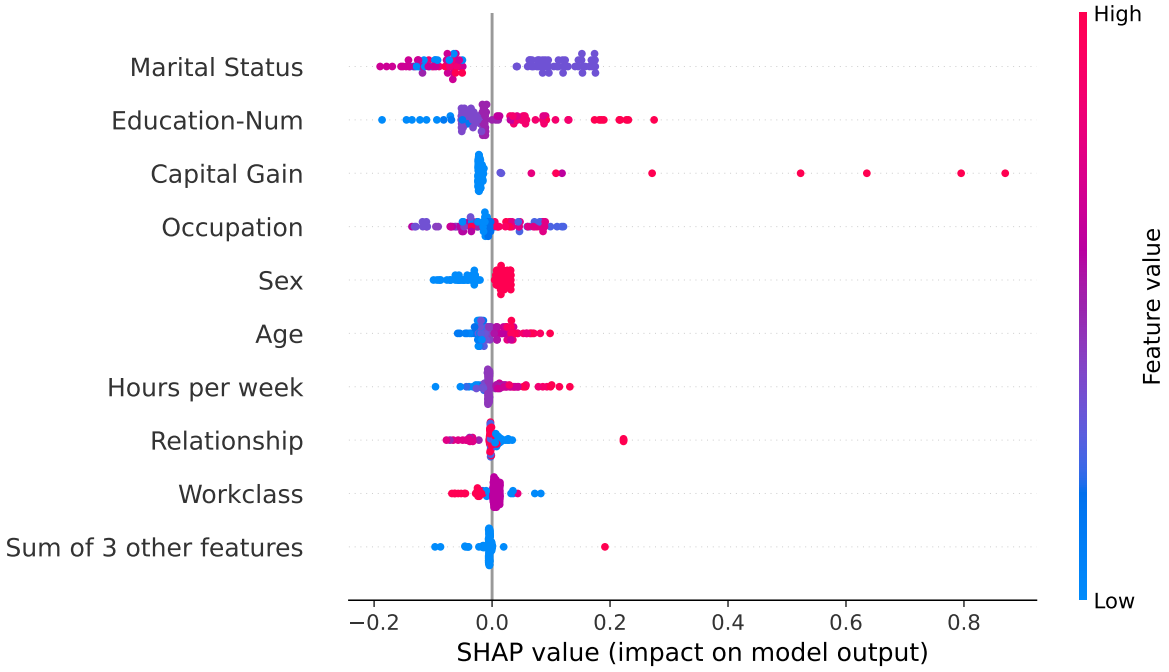


Figure 8.2: Probabilities versus Logits



From our observations, Marital Status and Education emerge as the two most important features. For some individuals, Capital Gain has substantial effects, suggesting that large capital gains result in large SHAP values.

8.6 Clustering SHAP values

Clustering can be applied to your data using SHAP values. The objective of clustering is to identify groups of similar instances. Typically, clustering relies on features, which are often of different scales. For instance, height may be measured in meters, color intensity from 0 to 100, and some sensor output between -1 and 1. The tricky part is calculating distances between instances with such diverse, non-comparable features.

SHAP clustering operates by clustering the SHAP values of each instance, meaning that instances are clustered by their explanation similarity. All SHAP values share the same unit — the unit of the prediction space. This concept is also called "supervised clustering" since we use information from a supervised model – here via SHAP – to cluster the data. Any clustering method can be employed. The following example utilizes hierarchical agglomerative clustering to sort the instances.

The plot comprises numerous force plots, each explaining the prediction of an instance. We rotate the force plots vertically and arrange them side by side according to their clustering similarity. It's a JavaScript plot. You might want to reduce the number of data points so that the plot doesn't get overwhelming, both visually and for your CPU. Here's how it looks for the first 21 data points:

```
shap.plots.force(sv_logit[0:20:,:,0])
```

i Cluster plot

- The cluster plot consists of vertical force plots.
- Data instances are distributed across the x-axis, while SHAP values are spread across the y-axis.
- Color signifies the direction of SHAP values.
- The larger the area for a feature, the larger the SHAP values across

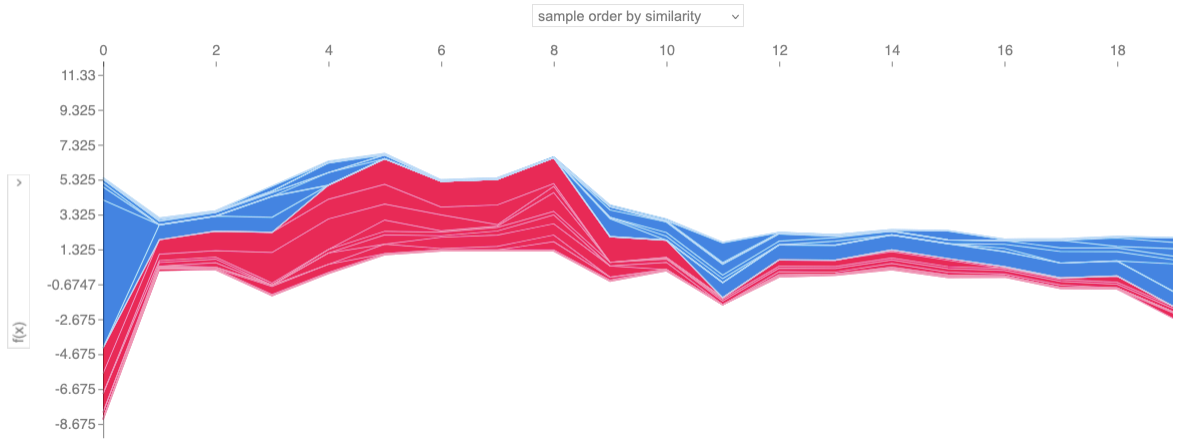


Figure 8.3: Clustering Plot

the data, indicating the importance of this feature.
- By default, the data instances are arranged by their similarity in SHAP values.

You can hover over the plot for more information, change what you see on the x-axis, and experiment with various other orderings. The cluster plot is an exploratory tool.

You can also alter the ordering by clicking on the interactive graph, for example, by the prediction:

8.7 The heatmap plot

The heatmap plot is another tool for global interpretation. Unlike the dependence or importance plot, the heatmap plot displays all SHAP values without aggregation. Some features are automatically summarized – a plotting behavior that can be controlled by setting the `max_display` argument.

```
shap.plots.heatmap(sv_logit[:,:,class_index])
```

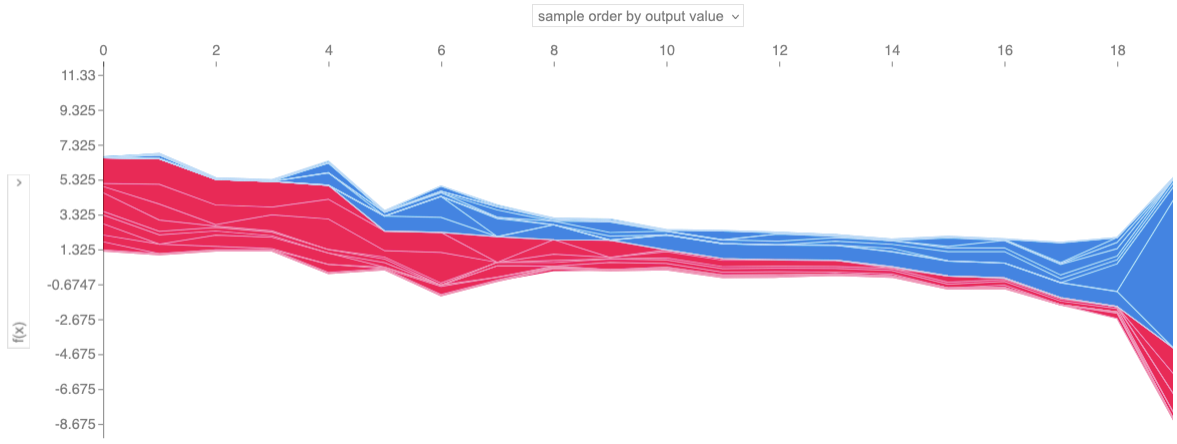
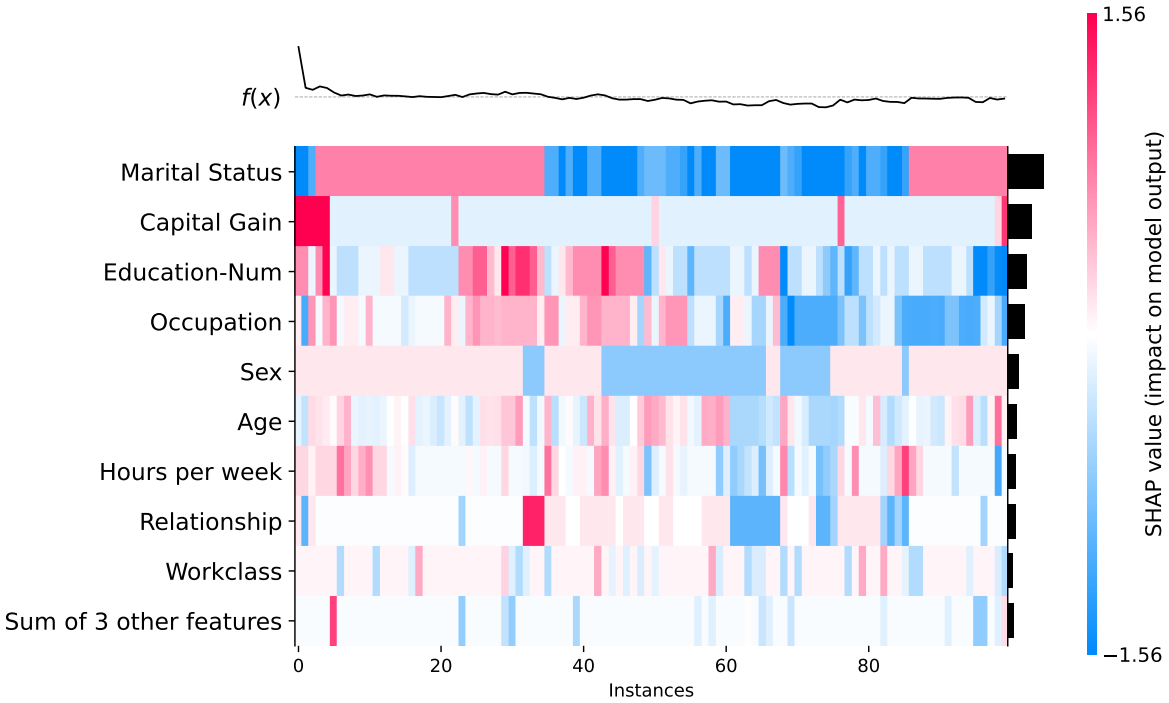


Figure 8.4: Clustering Plot



Heatmap plot

- Each row on the y-axis represents a feature, and instances are distributed across the x-axis.
- The color signifies the SHAP value.
- Instances are arranged based on clustered SHAP values.
- The curve at the top displays the predicted value for the data.

9 SHAP Values for Additive Models

By the end of this chapter, you will be able to:

- Use SHAP for additive regression models.
- Interpret non-linear effects.
- Explain SHAP feature importance.

In this section, we introduce the concept of non-linear relationships between a feature and the target, while excluding interactions between the features.

9.1 Introducing the generalized additive model (GAM)

Generalized additive models (GAMs) are perfect for modeling purely additive effects using non-linear base functions.

A GAM models the target as follows:

$$f(x) = \beta_0 + \beta_1 f_1(x_1) + ... + f_p(x_p)$$

Unlike the simple linear model, we allow the functions f_j to be non-linear. If for all features, $f_j(x_j) = x_j$, we arrive at the linear model. Thus, linear regression models are special cases of GAMs.

With GAMs, we can use arbitrary functions for the features. Popular choices include spline functions, which allow for flexible, smooth functions with a gradient. Tree-based basis functions, which have a fast implementation, are also an option. Additive models expand our understanding of SHAP values, as they allow us to examine non-linear functions without interactions. Although we could add

interaction terms to a GAM, we will not do so in this chapter, as the interpretation becomes more complex.

9.2 Fitting the GAM

We return to the wine example and fit a GAM instead of a linear regression model. We'll use the `interpret` Python library for this. You can install it with:

```
pip install interpret
```

Next, we fit a model. We're using the Explainable Boosting Regressor from the interpret package. The Explainable Boosting Machine (EBM) is a tree-based GAM. It offers optional automated interaction detection, which we won't use in this example. In our case, each tree in the ensemble can only use one feature to avoid modeling interactions.

Here we train the model:

```
import pandas as pd
from sklearn.model_selection import train_test_split
from interpret.glassbox import ExplainableBoostingRegressor

wine = pd.read_csv('wine.csv')
y = wine['quality']
X = wine.drop('quality', axis=1)
X_train, X_test, y_train, y_test = train_test_split(
  X, y, test_size=0.2, random_state=42
)

model = ExplainableBoostingRegressor(interactions=0)
model.fit(X_train, y_train)
```

Let's evaluate the model's predictions on the test data:

```
from sklearn.metrics import mean_absolute_error

y_pred = model.predict(X_test)
mae = mean_absolute_error(y_test, y_pred)
print(f"MAE: {mae:.2f}")
```

```
MAE: 0.55
```

The mean absolute error on the test data is less than that of the linear regression model. This is promising as it indicates that by using a GAM and allowing non-linear feature effects, we improved predictions. The increase in performance indicates that some relations between wine quality and features are non-linear.

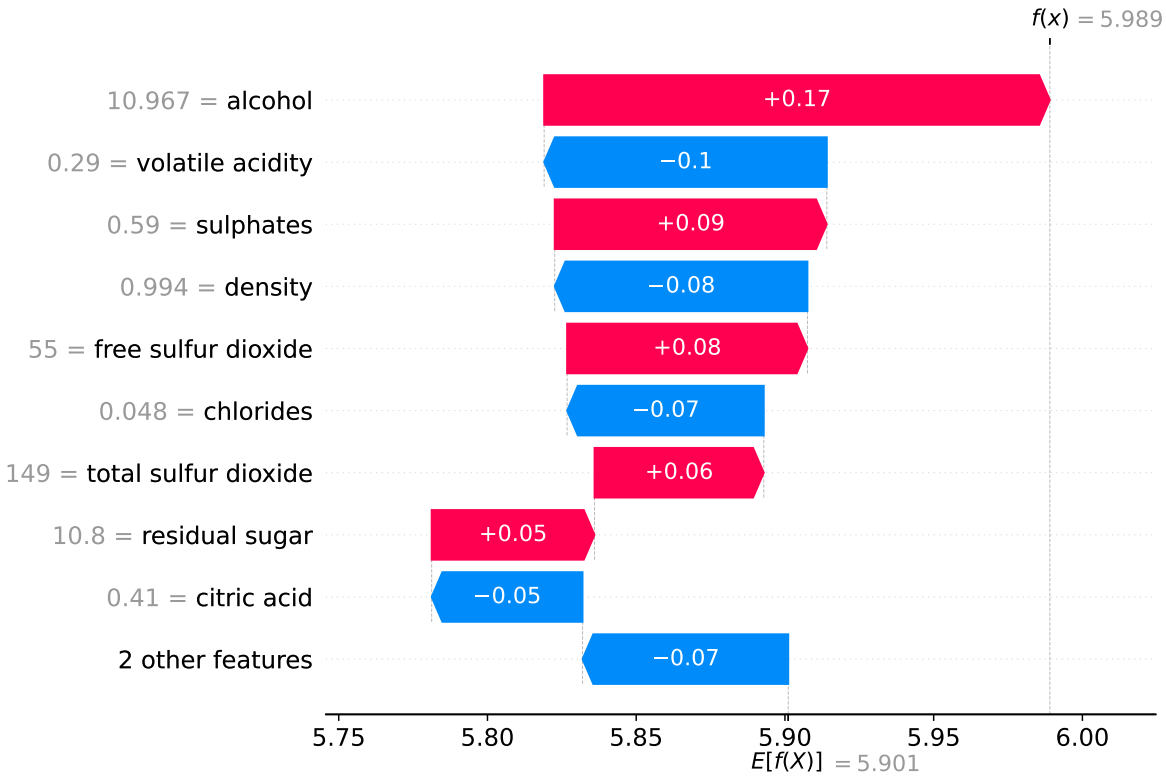
9.3 Interpreting the GAM with SHAP

Now let's see what the SHAP values have to say about how the model works:

```
import shap
explainer = shap.Explainer(model.predict, X_train)
shap_values = explainer(X_test)
```

The question now is: How do we interpret the SHAP values? And do they align with our expectations? First, let's elucidate the initial prediction of a data point:

```
shap.plots.waterfall(shap_values[0], max_display=10)
```



> **Note**
>
> The explainer used here is a Permutation explainer. The Additive explainer method isn't used because the additive estimation in `shap` is only implemented for `interpret.glassbox.ExplainableBoostingClassifier`, and we are using the Regressor.

This waterfall plot provides a different perspective than the purely linear model.

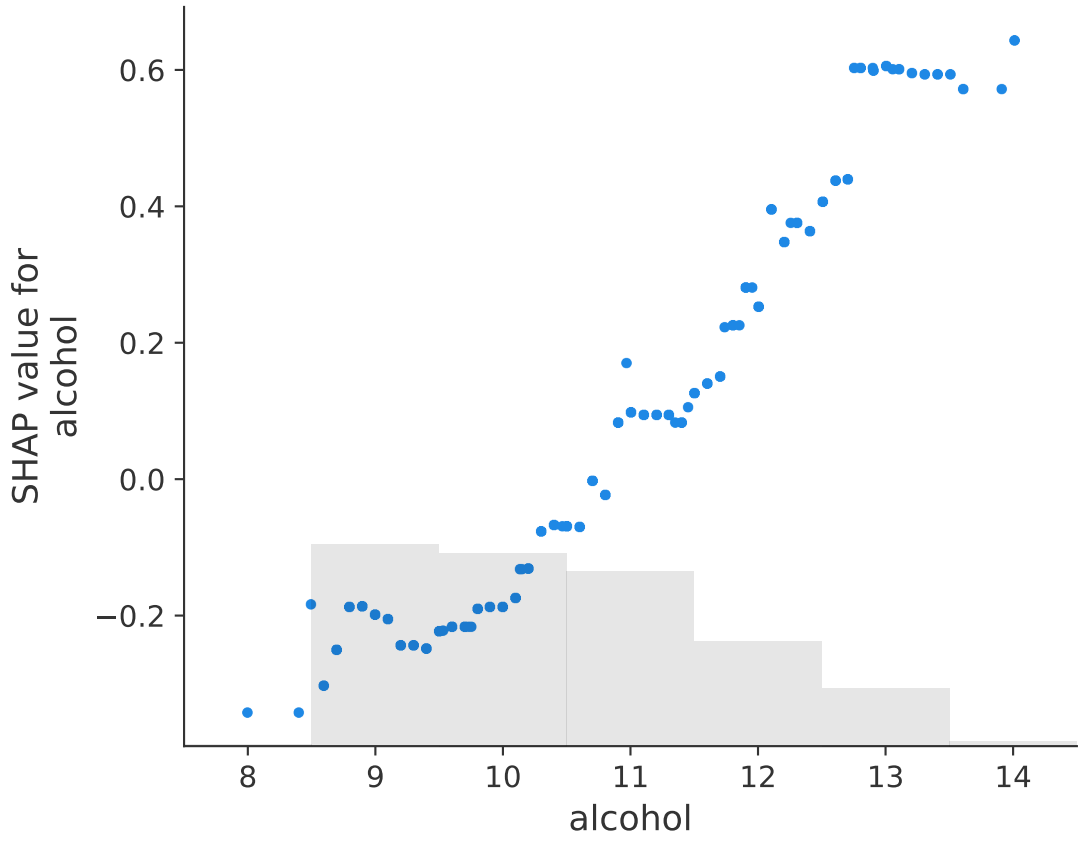
- For this particular wine, the most important features were alcohol and free sulfur dioxide, whereas, in the linear model, they were residual sugar and free sulfur dioxide.
- The quality predicted by the GAM is approximately 6.0, lower than the 6.4 predicted by the linear model.
- This example clearly illustrates how the global average prediction and the local prediction can be similar, but numerous SHAP values cancel each

other out.

9.4 SHAP recovers non-linear functions

A GAM is an additive model, which implies that we can inspect each feature in isolation to understand its effect without considering interaction effects. Let's examine the SHAP dependence plot for alcohol:

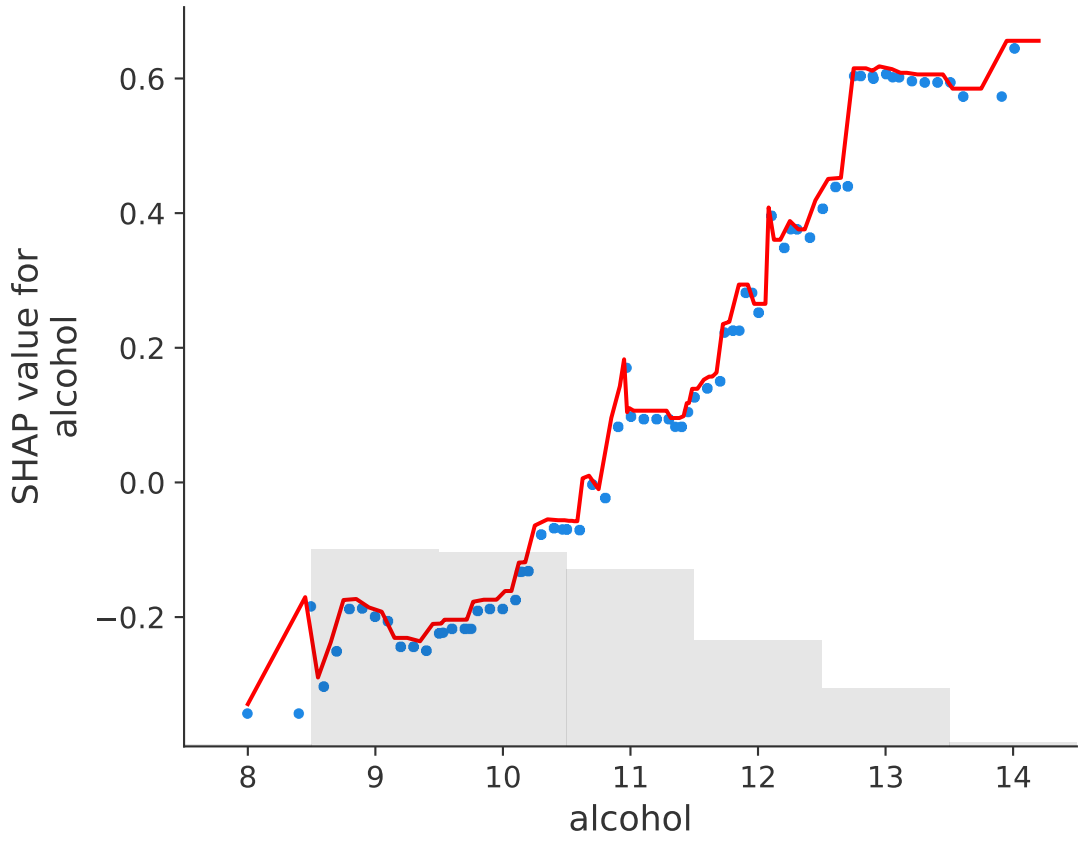
```
shap.plots.scatter(shap_values[:,"alcohol"])
```



In the case of alcohol, there is a positive relationship between alcohol levels and the SHAP values. The SHAP contribution increases with the alcohol content, but it plateaus at extremely high and low levels.

Let's compare these SHAP values with the alcohol effect learned by the GAM. We can plot the SHAP values and overlay the alcohol curve extracted directly from the GAM.

```
import matplotlib.pyplot as plt
import numpy as np

shap.plots.scatter(shap_values[:,"alcohol"], show=False)

# First get the index of the alcohol feature
idx = model.explain_global().data()['names'].index('alcohol')
# extract the relevant data from the tree-based GAM
explain_data = model.explain_global().data(idx)
# the alcohol feature values
x_data = explain_data["names"]
# the part of the prediction function for  alcohol
y_data = explain_data["scores"]
y_data = np.r_[y_data, y_data[np.newaxis, -1]]
plt.plot(x_data, y_data, color='red')
plt.show()
```



As evident, the SHAP values follow the same trajectory as we would see when simply altering one of the features (here, alcohol). This reinforces our confidence in understanding SHAP values. There's a paper (Bordt and Luxburg 2022) that demonstrates that when the model is a GAM, the non-linear components can be recovered by SHAP.

Like the linear case, in the additive case, SHAP values accurately track the feature effect and align with what we would expect.

9.5 Analyzing feature importance

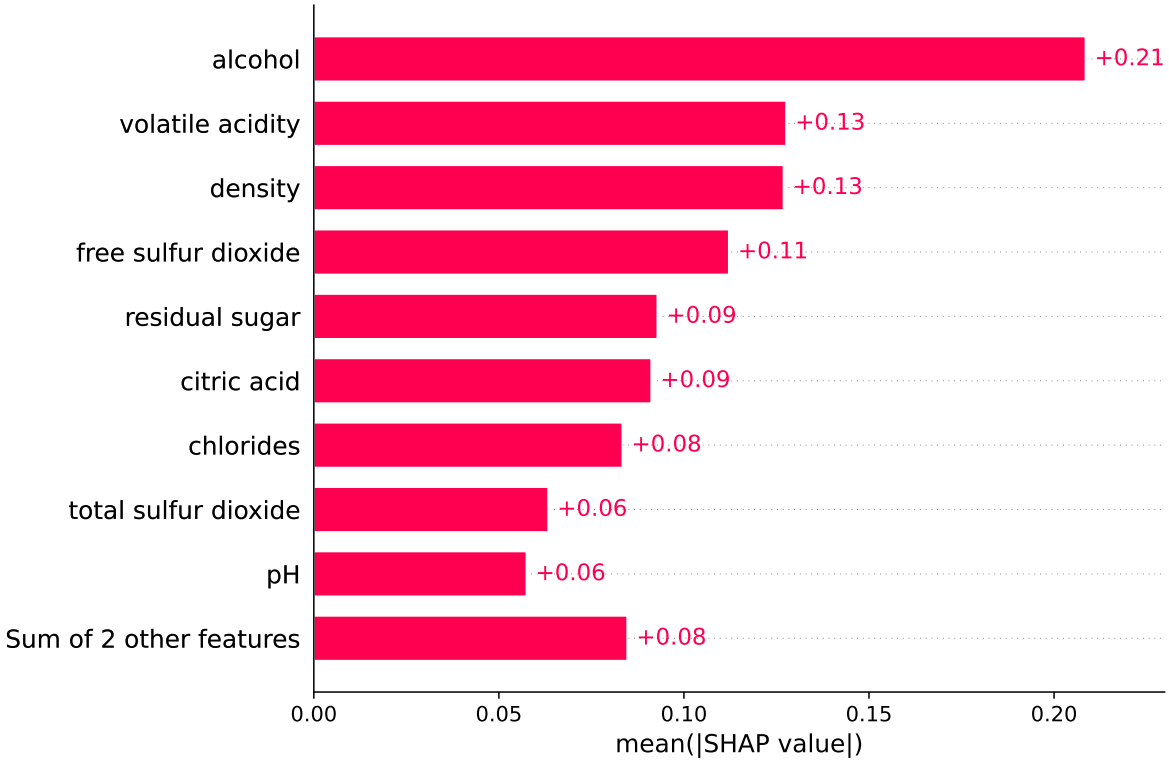
`shap` offers another type of plot: the importance plot. The principle behind SHAP feature importance is simple: Features with large absolute SHAP values

are important. To determine global importance, we average the **absolute** SHAP values per feature across the data:

$$I_j = \frac{1}{n} \sum_{i=1}^{n} |\phi_j^{(i)}|$$

We then sort the features by decreasing importance and plot them. This method of sorting features is also used in the summary plot.

```
shap.plots.bar(shap_values)
```



SHAP feature importance provides an alternative to permutation feature importance[1]. There's a significant difference between these importance measures:

[1] https://christophm.github.io/interpretable-ml-book/feature-importance.html

> **Note**
>
> Permutation Feature Importance (PFI) is derived from the decline in model performance, whereas SHAP relies on the magnitude of feature attributions. This difference becomes particularly pronounced when the model is overfitting. A feature that doesn't actually correlate with the target will have an expected PFI of zero but may exhibit a non-zero SHAP importance.

10 Understanding Feature Interactions with SHAP

By the end of this chapter, you will be able to:

- Describe what a feature interaction is.
- Interpret interactions within the dependence plot.
- Understand SHAP interaction values.

Interpreting models becomes more complex when they contain interactions. This chapter presents a simulated example to explain how feature interactions influence SHAP values.

10.1 A function with interactions

Imagine you're at a concert. Everyone loves being able to actually see the band and not just the backs of other people's heads. How much you can see depends on your height and how close you are to the stage. We'll ignore the factor of who's in front of you for now. We're simulating a score of how much a fan will enjoy the concert based on two features:

- x_1: Height in cm.
- x_2: Distance to the stage (from 0 to 10).
- y: How much the fan enjoys the concert.

We simulate the target as:

$$y = 0.1 \cdot x_1 - 1 \cdot x_2 + 10 \cdot \mathbb{1}_{x_1<160 \text{ and } x_2>7} - 8$$

- The taller the fan, the better.
- The closer to the stage, the better.
- Both of these have linear relationships with concert enjoyment.
- Additional interaction: Small fans who are far from the stage get a "bonus": some kind soul allows them to sit on their shoulders for a better view of the concert.
- The -8 is just so that the output is roughly between 0 (bad concert experience) and 10 (great concert experience).

> **Feature Interaction**
>
> Two features interact when the prediction can't be explained by the sum of both feature effects. Alternatively formulated: interaction means the effect of one feature changes depending on the value of the other feature.
> Consider the price of a hotel room, which depends on room size and sea view. Both factors individually contribute to the price: a larger room costs more, as does a room with a sea view. However, size and view interact: for small rooms, the sea view adds less value than it does for large rooms, since small rooms are less inviting for extended stays.

We'll simulate some data and train a random forest on it.

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestRegressor
import matplotlib.pyplot as plt
np.random.seed(42)

n = 1000
X = pd.DataFrame({
  'x1': np.random.uniform(140, 200, n),
  'x2': np.random.uniform(0, 10, n)
})

y = 0.1 * X.x1 -  1 *  X.x2 + 10 * (X.x1 < 160) * (X.x2 > 7) - 8

X_train, X_test, y_train, y_test = train_test_split(
```

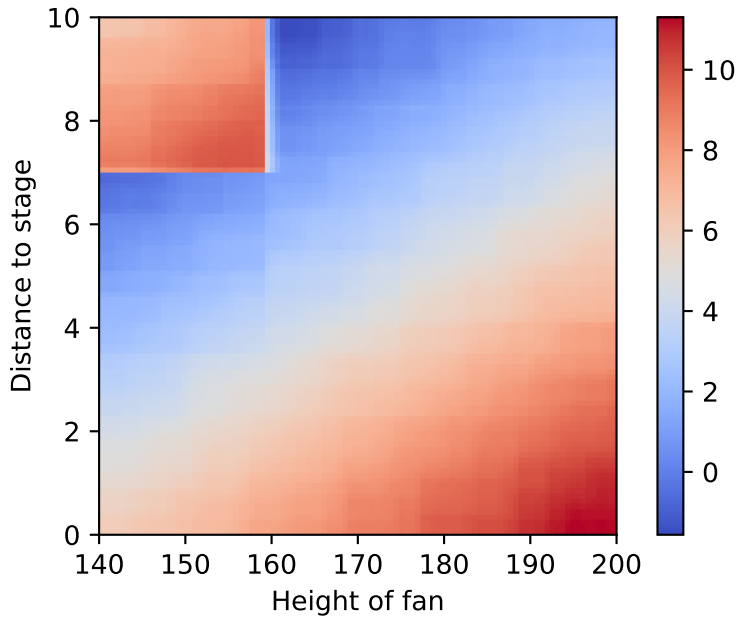
```
    X, y, test_size=0.2, random_state=42
)
```

And finally we train the model:

```
rf_model = RandomForestRegressor(n_estimators=100, random_state=42)
rf_model.fit(X_train, y_train)
```

Here's a visualization of how the features height and distance relate to the prediction:

```
# Generate x1 and x2 grids
x1 = np.linspace(140, 200, 100)
x2 = np.linspace(0, 10, 100)
xx1, xx2 = np.meshgrid(x1, x2)

# Flatten the grids and predict color
Xgrid = np.column_stack((xx1.ravel(), xx2.ravel()))
color = rf_model.predict(Xgrid)

# Reshape the predicted color array
color = color.reshape(xx1.shape)

# Plot the heatmap
plt.imshow(color, extent=[x1.min(), x1.max(), x2.min(), x2.max()],
           origin='lower', cmap='coolwarm', aspect=6)
plt.xlabel('Height of fan')
plt.ylabel('Distance to stage')
plt.colorbar()
plt.show()
```

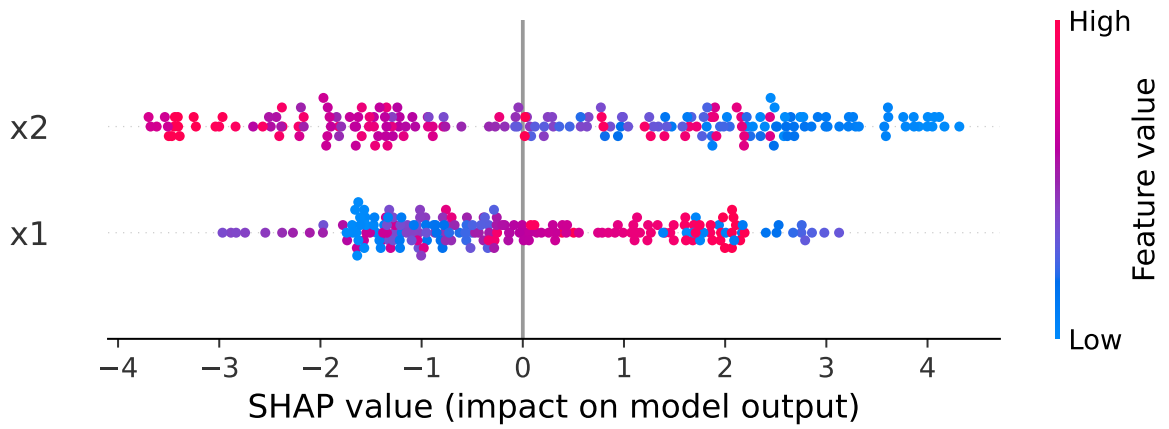


The random forest appears to approximate the function quite accurately. Next, we will generate explanations for the predictions.

10.2 Computing SHAP values

Let's calculate some SHAP values:

```
import shap

explainer = shap.TreeExplainer(rf_model)
shap_values = explainer(X_test)
shap.plots.beeswarm(shap_values)
```



The summary plot reveals the following general relationships:

- The further away from the stage (x_2), the smaller the SHAP values.
- The taller the fan (x_1), the larger the SHAP values.

However, there are some exceptions: Small fans sometimes receive large SHAP values due to the interaction effect of sitting on shoulders. To investigate further, let's examine the dependence plot, Figure 10.1.

```
shap.plots.scatter(shap_values[:,0], color=shap_values)
```

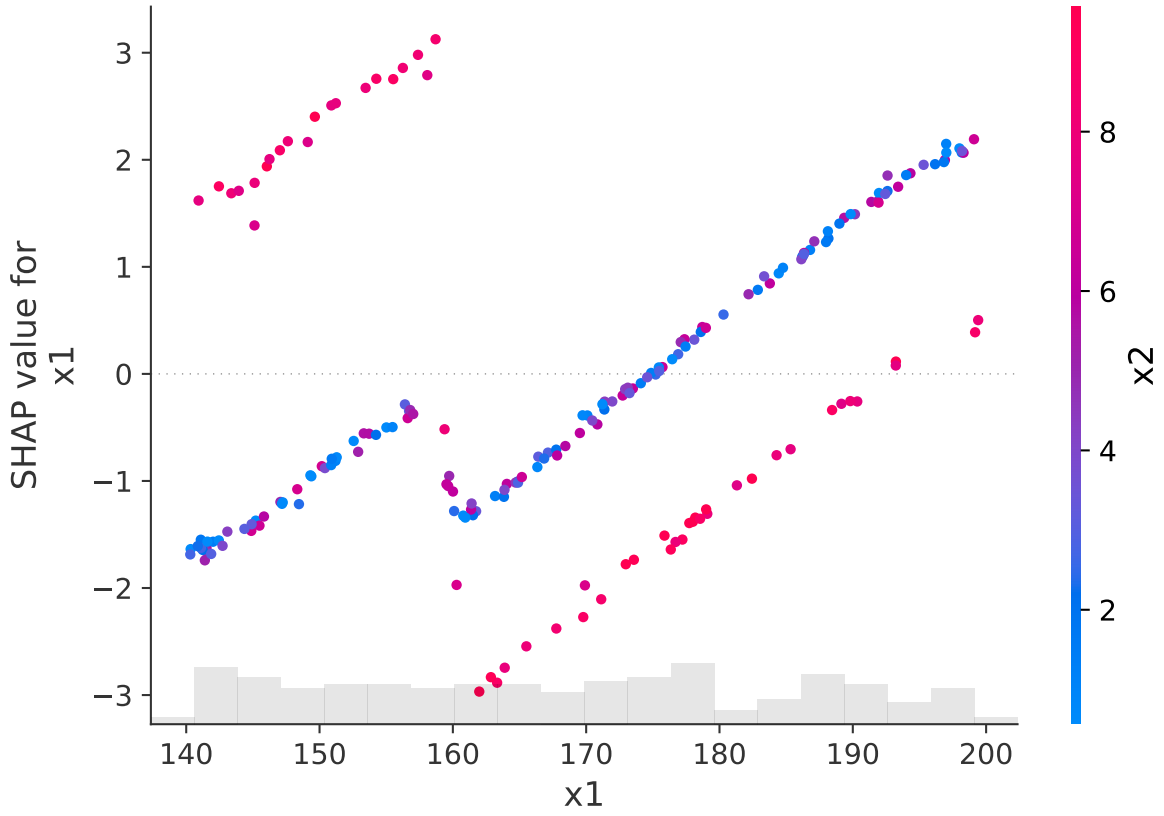


Figure 10.1: Dependence plot for x1, the height feature

The dependence plot colors the points by the values of feature x_2, as we provided the SHAP values for the color option. By default, the points are colored by the feature with the highest approximate interaction. Given our model only contains two features, the selection is naturally feature x_2. We can make three observations:

1. A large jump at $x_1 = 160$ is logical because, according to our simulated data, fans taller than 160cm will not sit on someone's shoulders.
2. Ignoring the jump, there seems to be a linear upward trend, which aligns with the linear dependence of Y on x_1. The slope reflects the coefficient in the simulated function ($\beta_1 = 0.1$).
3. There are two "clusters" of points: one with a small jump and one with a large jump.

The third point becomes clearer when we note the curves are colored by the feature value x_2. There are two "lines":

- One line represents fans who are >7 away from the stage (x_2). Here we see the large jump, which is expected since fans taller than 160cm have no chance of getting on someone's shoulders.
- The other line represents values of x_2 below 7. It has a smaller jump, but why is there a jump at all? Fans in this "cluster" don't get to sit on someone's shoulders when they are smaller than 160cm.

The reason why the interaction also "bleeds" into the cluster where we wouldn't expect it has to do with how SHAP values function.

10.3 SHAP values have a "global" component

Let's consider two fans:

1. Mia, who is 159cm tall and 2 units away from the stage.
2. Tom, who is 161cm tall and standing right next to Mia, also 2 units away from the stage.

Here are the model's predictions for how much they will enjoy the concert:

```
# Creating data for Mia and Tom
Xnew = pd.DataFrame({'x1': [159, 161], 'x2': [2,2]})

print("""
Mia: {mia}
Tom: {tom}
Expected: {exp}
""".format(
  mia=round(rf_model.predict(Xnew)[0], 2),
  tom=round(rf_model.predict(Xnew)[1], 2),
  exp=round(explainer.expected_value[0], 2)
))
```

```
Mia: 5.88
Tom: 6.07
Expected: 4.86
```

They have a rather similar predicted joy for the concert, with Mia having a slightly worse prediction – makes sense given she is slightly smaller and neither of them qualify for shoulders.

Let's examine their SHAP values.

```
shap_values = explainer(Xnew)

print('Mia')
print(shap_values[0].values)

print('Tom')
print(shap_values[1].values)
```

```
Mia
[-0.15944093  1.18150926]
Tom
[-1.37794848  2.5913748 ]
```

The SHAP values for Mia and Tom differ significantly.

- Mia's slightly negative value for height is understandable, given her relative shortness compared to the majority of simulated heights.
- Mia's positive SHAP value for distance makes sense as she is quite near the front.
- Tom's SHAP values follow similar trends, which is expected since they share the same distance and similar heights.
- However, Tom's SHAP values are more pronounced.

But shouldn't Mia have a smaller SHAP value for height than Tom? Neither of them benefits from the shoulder bonus, so Mia being smaller than Tom should mean that her SHAP value for "height" should be smaller than Tom's, right? But

surprisingly, Mia's SHAP value is influenced by the interaction term, despite her not being directly affected by the shoulder bonus!

This outcome is a result of the calculation process of SHAP values: When computing the SHAP value for Mia's height, one of the marginal contributions involves adding her height to the empty coalition ($\emptyset$). For this marginal contribution, we have to sample the stage distance feature. And sometimes we sample distances > 7, which activate the shoulder bonus. But only for Mia, not for Tom. The shoulder bonus strongly increases concert enjoyment and, as a consequence, Mia's SHAP value for height becomes greater than Tom's. So even though Mia is too close to the stage to get the shoulder bonus, her height's SHAP value accounts for this interaction. This example shows that SHAP values have a global component: Interactions influence data points that are far away.

10.4 SHAP values are different from a "what-if" analysis

This characteristic distinguishes SHAP values from a "what-if" analysis. A what-if analysis asks: How would the prediction change if we altered Mia or Tom's height? The likely answer is that the effect would be similar for both if we, for instance, increased their height by 10 cm. However, SHAP values do not operate this way. They reflect interactions with other features, even if we have to change multiple feature values of a data instance.

> ⚠ Warning
>
> SHAP values should not be interpreted as a what-if analysis (e.g., "what if I increased the feature by one unit").

In a what-if analysis, we would only evaluate how the prediction changes when the height changes, which should be similar for both Tom and Mia.

But, what does such a contrived example have to do with real machine learning applications? The extreme phenomenon observed here occurs subtly in real applications, albeit in more complex ways. The interactions might be more nuanced and intricate, and there may be more features involved. However, this global phenomenon is likely to occur. Interactions within a machine learning model can

be highly complex. Thus, bear this limitation in mind when interpreting SHAP values.

11 The Correlation Problem

By the end of this chapter, you will be able to:

- Describe the challenge of correlated features for SHAP.
- Select solutions to reduce the correlation problem.

SHAP values encounter a subtle yet relevant issue when dealing with correlated features. Simulating the absence of features by replacing them with sampled values from the background data can generate unrealistic data points. These implausible data points are then used to produce explanations, resulting in several problems:

- Including unlikely or unrealistic data points in the explanation may not be desirable.
- Physically impossible data points (e.g., a 2-meter tall person weighing 10 kg) may present a conceptual issue.
- The model may not have been trained on data in these areas and might yield unexpected results.

In this chapter, we will first dive into the problem in more detail using a small simulation, followed by a discussion of possible solutions.

11.1 Correlated features cause extrapolation

To illustrate this issue, I simulated two highly correlated features, X_1 and X_2. Assuming a machine learning model is later trained on this data, we won't actually train a model but will instead explore how sampling background data can be problematic.

```
import numpy as np

np.random.seed(42)

p = 0.9
mean = [0, 0] # mean vector
cov = [[1, p], [p, 1]] # covariance matrix
n = 100 # number of samples

x1, x2 = np.random.multivariate_normal(mean, cov, n).T
```

Next, let's create a data point for which we will simulate the sampling from the background data.

```
point = (-1.7, -1.7)
```

I'm interested in the SHAP value for feature X_1. I will demonstrate how SHAP values sample from the marginal distribution and compare that to what sampling from the conditional distribution would look like.

```
import matplotlib.pyplot as plt
# set number of samples
m = 15

# create marginal and conditional distribution
x2_cond = np.random.normal(
  loc=p*point[0], scale=np.sqrt(1-p**2), size=m
)
x2_marg = np.random.choice(x2, size=m)

# create scatter plot with fixed x1 and variable x2
plt.subplot(121)
plt.scatter(x1, x2, color='black', alpha=0.1)
plt.scatter(np.repeat(point[0], m), x2_marg, color='blue')
plt.scatter(point[0], point[1], color='red')
plt.ylabel('x2')
```

```
plt.xlabel('x1')
plt.subplot(122)
plt.scatter(x1, x2, color='black', alpha=0.1)
plt.scatter(np.repeat(point[0], m), x2_cond, color='green')
plt.scatter(point[0], point[1], color='red')
plt.xlabel('x1')
plt.show()
```

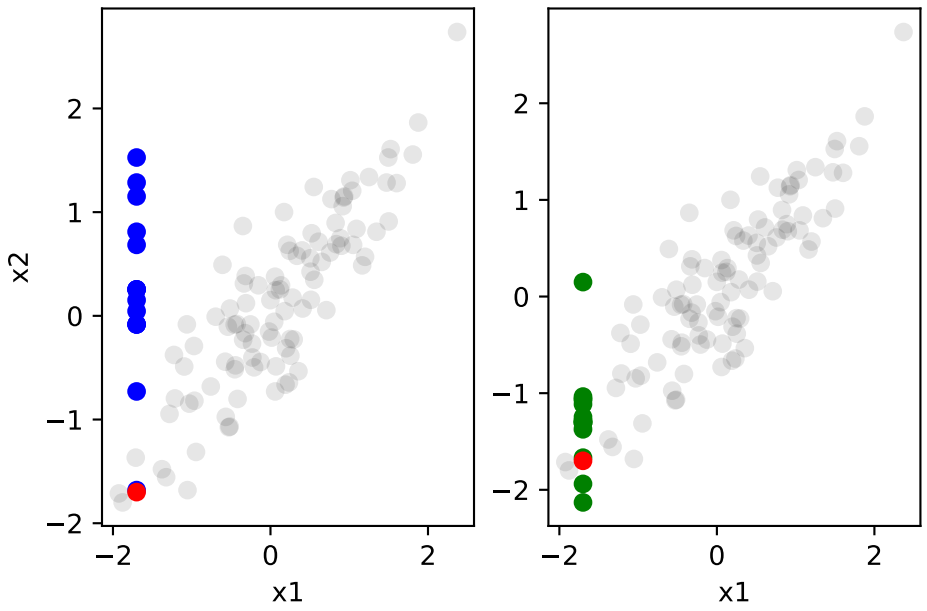


Figure 11.1: Marginal and conditional sampling

The plot on the left shows sampling from the marginal distribution: we disregard the correlation between X_1 and X_2 and sample X_2 independently from X_1. Marginal sampling in this correlated case creates new data points outside of the distribution. This is what occurs with SHAP values as we have used them throughout the book. On the right, we see conditional sampling. Conditional sampling means that we respect the distribution of X_2, given that we already know the value for X_1 and sample from $P(X_2|X_1)$ instead of $P(X_2)$.

Conditional sampling preserves the distribution, whereas marginal sampling may distort it when features are correlated.

11.2 A philosophical problem

Let's dive deeper into the correlation problem before discussing solutions. The problem of correlation and extrapolation is not just a technical challenge but also a philosophical one: do we want the interpretation to reflect the modeled relations or the data? Consider this: what does it mean when features are correlated and what are the consequences for interpretation? From an information theory perspective, correlation implies that two features share information. If information is shared, we cannot simply sample one feature without considering the correlated feature.

For instance, suppose we have two strongly correlated features: rainfall yesterday and cloudiness yesterday. If it rained yesterday, it must have been cloudy - these features are correlated. How can we then isolate the effect of rain while ignoring the cloudiness? Even if we disentangle the two, it would make for a weird interpretation.

11.3 Solution: Reduce correlation in the model

Although it may appear overly simplistic, avoiding the correlation problem can be resolved by eliminating correlated features. This solution requires flexibility in training the model with a different set of features. If you have the liberty to alter the features, we can leverage a range of techniques to reduce the correlation:

- Use feature selection methods, especially those that eliminate correlated features.
- Eliminate features with minimal variance.
- Implement dimensionality reduction techniques. However, this may lead to a loss of interpretability; for example, principal component analysis generates features that are difficult to interpret.

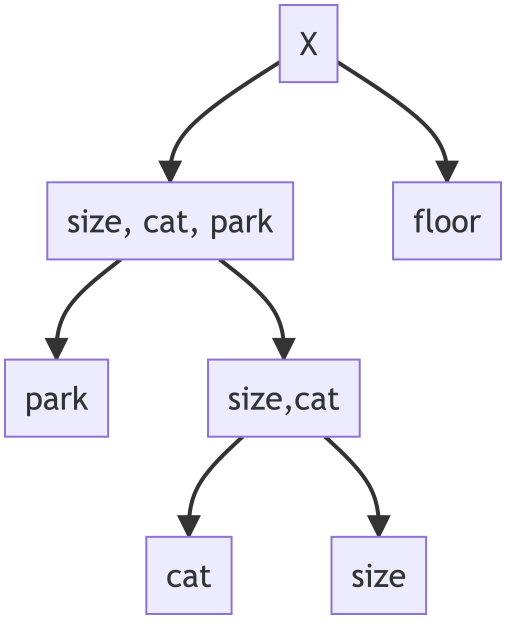
- Apply feature engineering to decrease correlation. For instance, if you have the features "apartment rent" and "number of rooms", they will be correlated. You can decorrelate them by converting the rent into "rent per square meter".
- Combine features. Perhaps having the amount of rain in the morning and afternoon as separate features is unnecessary. Would daily rainfall be sufficient? Test it out.

Reducing correlated features and the overall number of features can significantly enhance model fitting. You can assess how each of these steps impacts predictive performance, aiding in your decision-making for possible trade-offs.

11.4 Solution: Combined explanation of correlated features with Partition explainer

Rather than using the `Independent` explainer, opt for the `Partition` explainer. The Partition explainer generates a feature hierarchy presented as a tree. While any metric could be used to produce the hierarchy, it should be a correlation-based metric that hierarchically clusters the features, like Pearson correlation.

Within this hierarchy, SHAP values are calculated recursively. This procedure is its own game, known as Owen's game, and we get Owen values back. Owen values are closely linked to Shapley values, as we will see in the following example. Let's say for the apartment example we have the following binary tree of correlations:



- Cat and size are most strongly correlated.
- Both are slightly correlated with the park feature.
- The floor feature is not correlated with any of the features.

To compute Owen values, also known as hierarchical Shapley values, we start at the top with two feature groups: {size,cat,park} and {floor}. We treat both groups as players and compute their SHAP values $\phi_{\text{size,cat,park}}$ and ϕ_{floor}. In the next recursion, we split up the $\phi_{\text{size,cat,park}}$ among the groups {park} and {size, cat}, again as if we were computing SHAP, but with groups as players. We then move down to {cat} and {size}. The Partition explainer doesn't fully eliminate the correlation problem. As we move deeper down the correlation tree, we start to also break up strongly correlated features to recursively compute Owen's value. However, it's reduced, since we only try to explain the group's SHAP value and permute fewer features.

Owen's game alters the complexity from $\mathcal{O}(2^p)$ to $\mathcal{O}(p^2)$ (when the clustering tree is balanced), where p is the number of input features for the model.

11.5 Solution: Conditional sampling

As suggested in Figure 11.1, conditional sampling can also be utilised.

Here's a brief example:

- We have four features: X_1 through X_4.
- Calculate the SHAP value for X_1.
- During the sampling process, we add X_1 to the coalition $\{X_2\}$.
- Compute the predictions for the coalitions $\{X_2\}$ and $\{X_1, X_2\}$.
- The missing features are $\{X_1, X_3, X_4\}$ and $\{X_3, X_4\}$, and we sample these from the background data.
- Sampling can be carried out in two ways:
  - From $P(X_1, X_3, X_4)$ and $P(X_3, X_4)$ (marginal sampling), or
  - From $P(X_1, X_3, X_4|X_2)$ and $P(X_3, X_4|X_1, X_2)$ (conditional sampling)

Conditional sampling helps avoid extrapolation problems. Aas et al. (2021) suggested incorporating this method into KernelSHAP. However, implementing conditional sampling is not straightforward, given that supervised machine learning mainly learns $P(Y|X_1, \ldots, X_p)$, and we now need to learn complex distributions for numerous variables. Generally, we make simplifying assumptions for the conditional distributions to facilitate sampling, as suggested by Aas et al. (2021):

- Use multivariate Gaussians.
- Utilize Gaussian copulas.
- Apply kernel estimators (if the dimensions are not too many).

The paper suggests more techniques, but these are the most significant.

> ⚠ Warning
>
> Switching the sampling method from marginal to conditional changes the value function to a conditional value function:
>
> $$v_{f,x^{(i)}}(S) = \int f(x_S^{(i)} \cup X_C) d\mathbb{P}_{X_C|X_S=x_S^{(i)}} - \mathbb{E}(f(X))$$
>
> This shift changes the game. For example, the resulting SHAP values might seem to not adhere to the Dummy axiom, as unused features with, for ex-

ample, $\beta_j = 0$ in a linear model can suddenly have non-zero SHAP values. However, conditional SHAP does not actually violate the Dummy axiom, as they still qualify as (conditional) SHAP values based on the new payout using conditional distributions. Without correlation, marginal and conditional SHAP are identical, since then $P(X_1) = P(X_1|X_2)$.

If you want to employ conditional sampling in `shap`, use the Linear explainer with the `feature_perturbation='correlation_dependent'` option. The Tree explainer provides a similar option called `feature_perturbation='tree_path_dependent'`, which uses the splits in the underlying tree-based model to ensure SHAP is not extrapolating. However, I advise against using the tree-path option, as it does not accurately model the conditional distribution and does not approximate the conditional SHAP values well (Aas et al. 2021).

Whether a conditional SHAP interpretation is useful depends on your interpretation objectives. If you want to audit the model, marginal sampling may be more suitable. If your goal is to better understand the data, conditional sampling might be the better choice. This trade-off is often described as being true to the model (marginal sampling) or true to the data (conditional sampling) (Chen et al. 2020). Sundararajan and Najmi (2020) further discusses this concept.

12 Regression Using a Random Forest

By the end of this chapter, you will be able to:

- Interpret SHAP for a complex model with interactions and non-linear effects.
- Use the Partition Explainer for correlated features.
- Apply SHAP to analyze subsets of the data.

In this chapter, we will examine the wine dataset again and fit a tree-based model, specifically a random forest. This model potentially contains numerous interactions and non-linear functions, making its interpretation more complex than in previous chapters. Nevertheless, we can employ the fast `shap.TreeExplainer`.

12.1 Fitting the Random Forest

Random forests are ensembles of decision trees, and their prediction is an average of the tree predictions. Random forests usually provide good results without any adjustments.

Note

Gradient boosted trees algorithms such as LightGBM and xgboost are other popular tree-based models. The `shap` application demonstrated here works the same way with them.

```
import pandas as pd
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split

wine = pd.read_csv('wine.csv')
y = wine['quality']
X = wine.drop(columns=['quality'])

X_train, X_test, y_train, y_test = train_test_split(
  X, y, test_size=0.2, random_state=42
)

model = RandomForestRegressor(random_state=42)
model.fit(X_train, y_train)
```

Next, we evaluate the performance of our model, hoping for better results than with the GAM:

```
from sklearn.metrics import mean_absolute_error

y_pred = model.predict(X_test)
mae = mean_absolute_error(y_test, y_pred)
print('MAE:', round(mae, 2))
```

```
MAE: 0.42
```

This model performs better than the GAM, suggesting that additional interactions are beneficial. Despite the GAM also being tree-based, it did not model interactions.

12.2 Computing SHAP Values

Now, let's interpret the model:

```
import shap
# Compute the SHAP values for the sample
explainer = shap.Explainer(model, X_train)
shap_values = explainer(X_test)
```

> **Important**
>
> The code above produces an error: "This check failed because for one of the samples the sum of the SHAP values was 5.881700, while the model output was 5.820000." The Tree Explainer is an exact explanation method, and `shap` checks if additivity holds: the model prediction should equal the sum of SHAP values + base_value. In this case, there is a discrepancy in some SHAP values. I'm not entirely sure why this happens - it may be due to rounding issues. You might encounter this too, so here are two options to handle it: either set check_additivity to False or use a different explainer, like the Permutation Explainer. If you disable the check, ensure the difference is acceptable:
>
> ```
> import numpy as np
> shap_values.base_values +  np.sum(shap_values, axis=1) - \
>   model.predict(X_test)
> ```

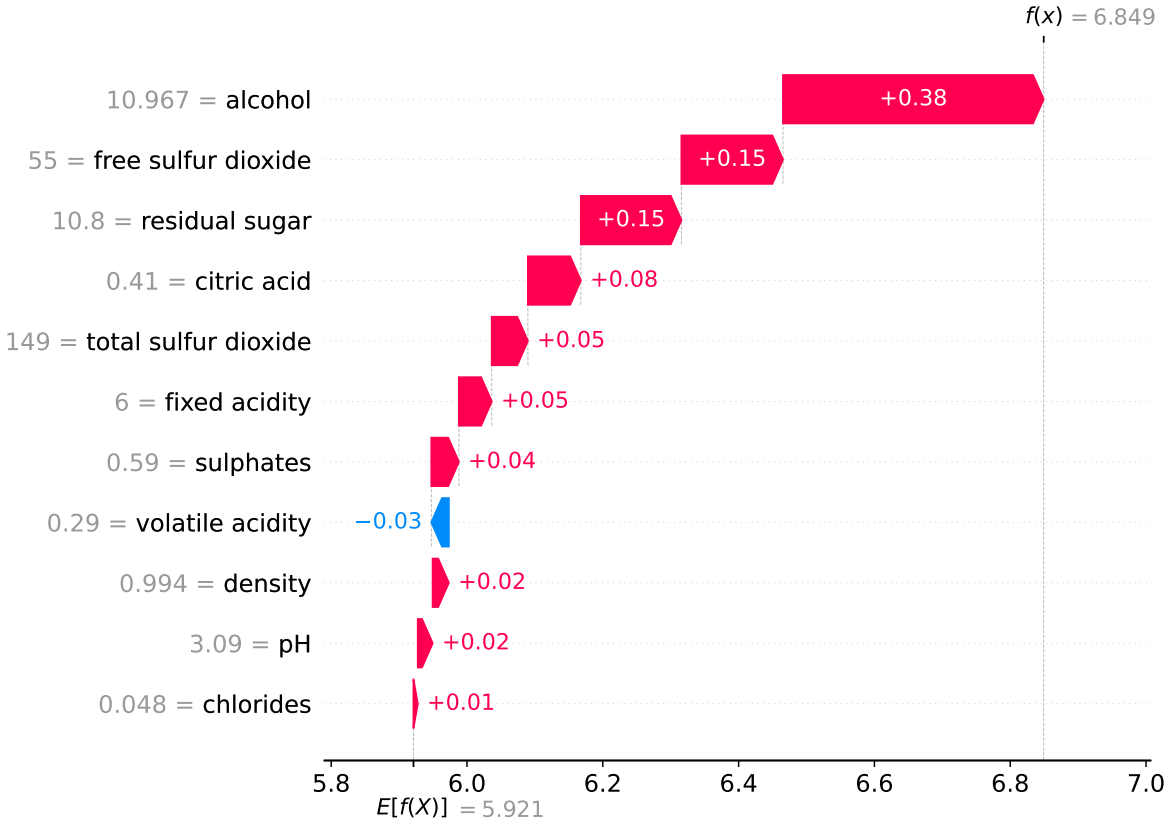
Let's try again without checking for additivity:

```
import shap
explainer = shap.Explainer(model, X_train)
shap_values = explainer(X_test, check_additivity=False)
```

> **Warning**
>
> Please provide a dataset or masker when creating an Explainer for a tree-based model. While other explainers will not function without data, the tree explainer will default to `feature_perturbation='tree_path_dependent'`, which is not recommended due to its ambiguous interpretation.

Let's revisit the SHAP values for the wine from the Linear Chapter and the Additive Chapter.

```
shap.plots.waterfall(shap_values[0], max_display=11)
```

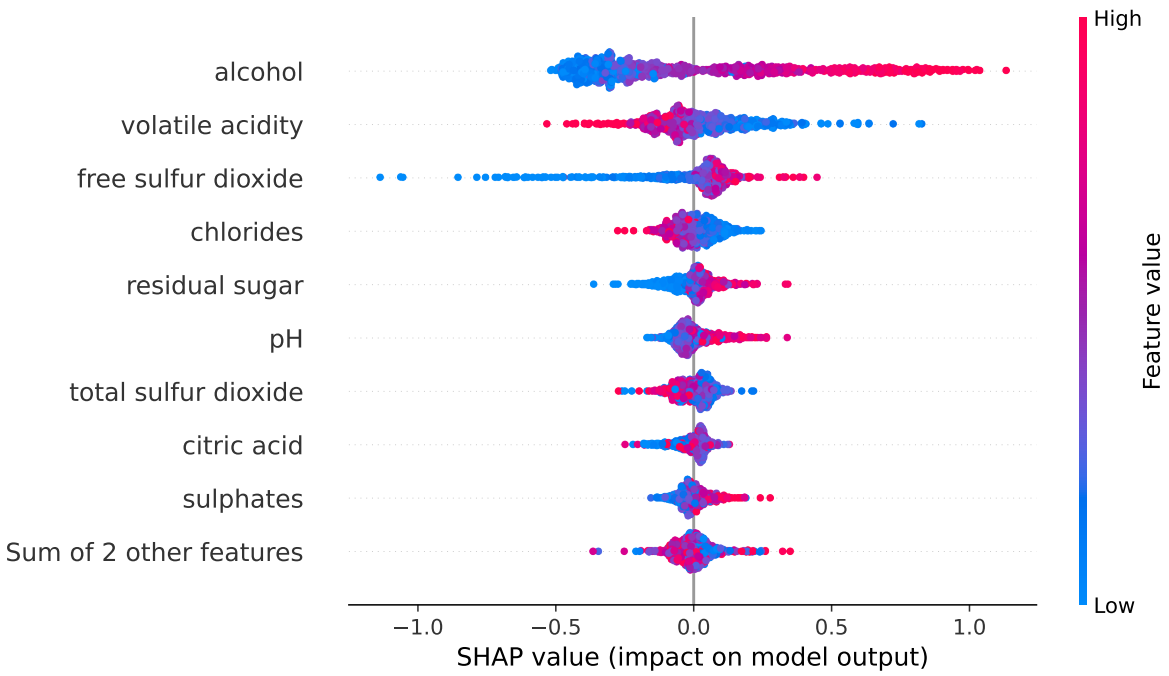


While the results differ from both the linear and the GAM models, the interpretation process remains the same. A key difference is that the random forest model includes interactions between the features. However, since there's only one SHAP value per feature value (and not one for every interaction), we don't immediately see how features interact.

12.3 Global model interpretation

Global SHAP plots provide an overall view of how features influence the model's predictions. Let's examine the summary plot:

```
shap.plots.beeswarm(shap_values)
```



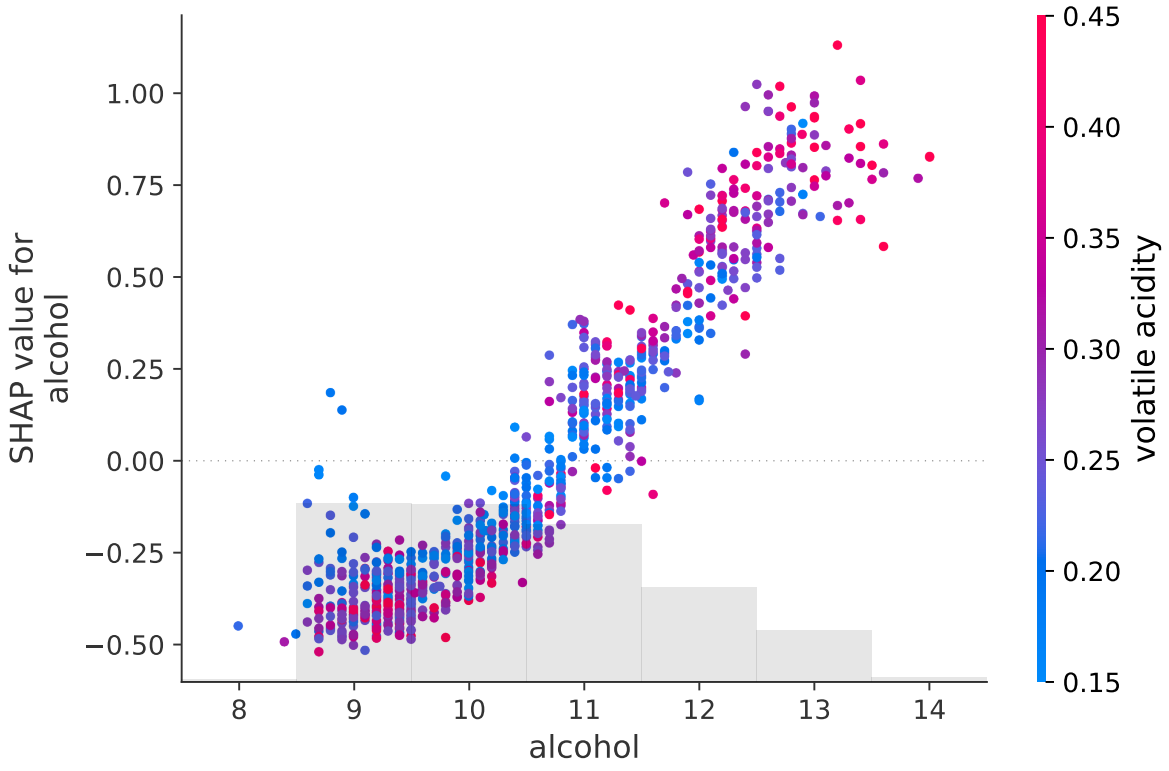
Key observations:

- Alcohol and volatile acidity emerged as the most important features.
- Several features, such as alcohol and volatile acidity, showed a somewhat monotonic relationship with the target.
- The factors with the largest absolute contributions to the predicted quality of some wines included:
  - High alcohol levels leading to higher predicted quality.

– Low levels of free sulfur dioxide resulting in lower quality.

We can examine interactions in global plots like the dependence plots. Here's the dependence plot for the alcohol feature:

```
shap.plots.scatter(shap_values[:,"alcohol"], color=shap_values)
```



The `shap` package automatically detects interactions. In this case, `shap` identified `volatile acidity` as a feature that greatly interacts with `alcohol` and color-coded the SHAP values accordingly. By default, the `shap` dependence plot chooses the feature that has the strongest interaction with the feature of interest. The dependence plot function calls the `approximate_interactions` function, which measures the interaction between features through the correlation of

SHAP values, with a stronger correlation indicating a stronger interaction. Then it ranks features based on their interaction strength with a chosen feature. You can also manually select a feature.

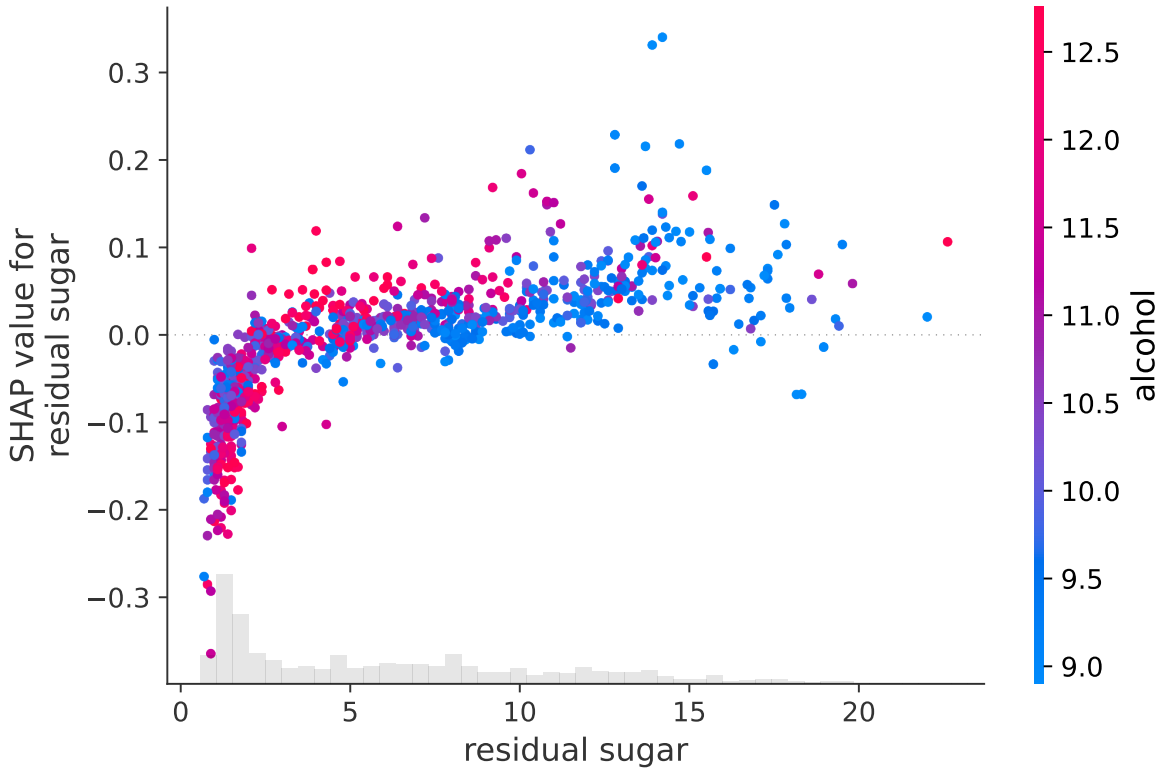
Here are some important observations:

- Generally, a higher alcohol level corresponds to a higher SHAP value.
- Examining cases with low volatile acidity reveals an interesting interaction with wines that have a low alcohol level. For wines with low alcohol (between 8% and 11%), if the wines have low volatile acidity, then the SHAP value for alcohol is higher compared to wines with similar alcohol levels.
- The relationship reverses for wines with higher alcohol levels: a higher volatile acidity level is associated with slightly higher SHAP values for alcohol.
- We could infer that volatile acidity alters the effect of alcohol on the predicted wine quality.
- However, this interaction is subtle, and we should avoid overinterpreting it, particularly considering the insights from the Interaction Chapter about the complexity of interactions.

> **Note**
>
> Here's some advice on interpreting the interaction part of the dependence plot:
>
> - Select one of the two variables.
> - For this variable, choose two ranges or categories.
> - Compare the SHAP values within these ranges.
> - Note whether any differences are related to changes in the other feature.

Next, let's examine the dependence plot for residual sugar as another example. Residual sugar represents the remaining sugar in the wine, with higher amounts indicating a sweeter taste.

```
shap.plots.scatter(
  shap_values[:,"residual sugar"], color=shap_values
)
```



Key observations:

- Higher residual sugar is associated with higher SHAP values.
- The `shap` package identifies alcohol as having the highest interaction with residual sugar.
- Alcohol and residual sugar are negatively correlated with a correlation coefficient of -0.5 (see later in this chapter); this makes sense as sugar is converted into alcohol during the fermentation process.
- Comparing curves for low (below 12) and high alcohol levels (above 12):
  - High variance in SHAP values is observed when alcohol content is low.
  - High alcohol content is associated with low residual sugar and higher SHAP values, compared to low alcohol content.

12.4 Analyzing correlated features

As mentioned in the Correlation Chapter, correlated features require additional consideration. Let's examine which features are correlated and how to use the Partition explainer. We'll start with a correlation plot that displays the Pearson correlation between the features, given by the formula:

$$r_{xy} = \frac{\sum_{i=1}^{n}(x^{(i)} - \bar{x})(z^{(i)} - \bar{z})}{\sqrt{\sum_{i=1}^{n}(x^{(i)} - \bar{x})^2}\sqrt{\sum_{i=1}^{n}(z^{(i)} - \bar{z})^2}}$$

The correlation ranges from -1, representing a perfect negative correlation, to +1, indicating a perfect positive correlation. A value of 0 suggests no correlation. In this case, x and z represent two features, while $\bar{x}$ and $\bar{z}$ are their respective averages. Note, however, that Pearson correlation only measures linear correlation, which isn't the only way features may be correlated. Use other measures of correlation, such as mutual information, if you want to capture different notions of correlation.

```
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np

# Compute the correlation matrix
corr = X_train.corr()

# Generate a mask for the upper triangle
mask = np.triu(np.ones_like(corr, dtype=bool))

# Set up the matplotlib figure
f, ax = plt.subplots(figsize=(11, 9))
# Generate a diverging colormap
cmap = sns.diverging_palette(230, 20, as_cmap=True)

# Draw the heatmap with the mask and correct aspect ratio
sns.heatmap(corr, mask=mask, cmap=cmap, vmax=.3, center=0,
    square=True, linewidths=.5, cbar_kws={"shrink": .5},
```

```
    annot=True, fmt=".1f")
plt.show()
```

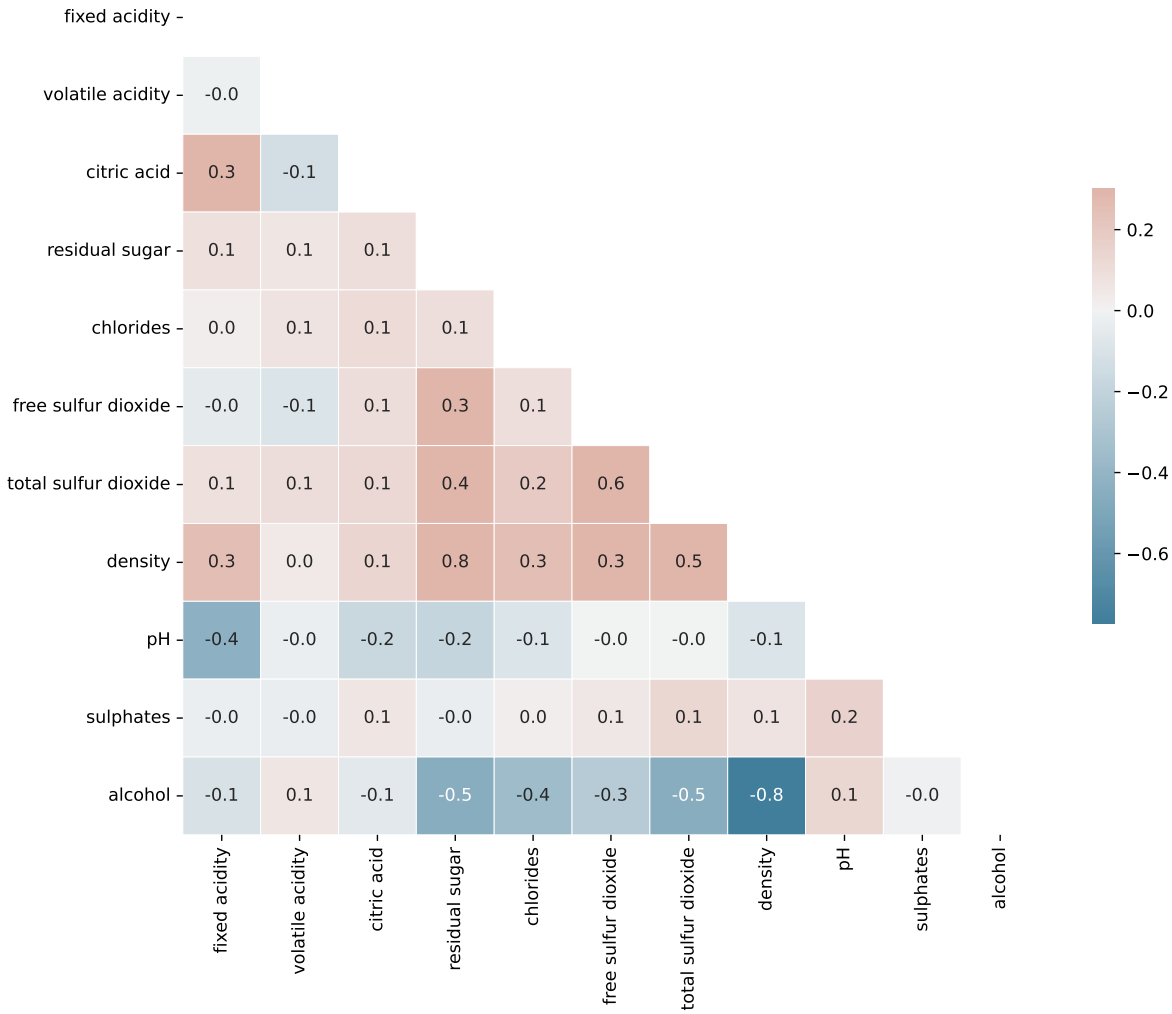


Figure 12.1: Pairwise feature correlations

Figure Figure 12.1 shows that density correlates with residual sugar (0.8) and total sulfur dioxide (0.5). Density also has a strong negative correlation with

alcohol. Volatile acidity does not show a strong correlation with other features.

The Partition explainer is a method that handles correlated features by computing SHAP values based on hierarchical feature clusters.

An obvious strategy is to use correlation to cluster features so that highly correlated features are grouped together. However, one modification is needed: Correlation leads to extrapolation, which we need to manage, but it doesn't matter whether the correlation is positive or negative. Clustering based on correlation would cause features with a strong negative correlation to be far apart in the clustering hierarchy, which is not ideal for our goal of reducing extrapolation. Therefore, in the following example, we perform tree-based hierarchical clustering on the **absolute correlation**. Features that are highly correlated, whether negatively or positively, are grouped together hierarchically until the groups with the least correlation are merged.

```
import matplotlib.pyplot as plt
from scipy.cluster import hierarchy

correlation_matrix = X_train.corr()
correlation_matrix = np.corrcoef(correlation_matrix)
correlation_matrix = np.abs(correlation_matrix)
dist_matrix = 1 - correlation_matrix

import scipy.cluster.hierarchy as sch

clustering = sch.linkage(dist_matrix, method="complete")

#clustering = shap.utils.hclust(X_train, metric='correlation')

plt.figure(figsize=(10, 7))
plt.title("Dendrograms")
dend = hierarchy.dendrogram(clustering, labels=X_train.columns)

# Rotate labels for better readability
plt.xticks(rotation=90)

# Increase label size for better visibility
plt.tick_params(axis='x', which='major', labelsize=12)
```

```
plt.ylabel('Correlation Distance')

plt.show()
```

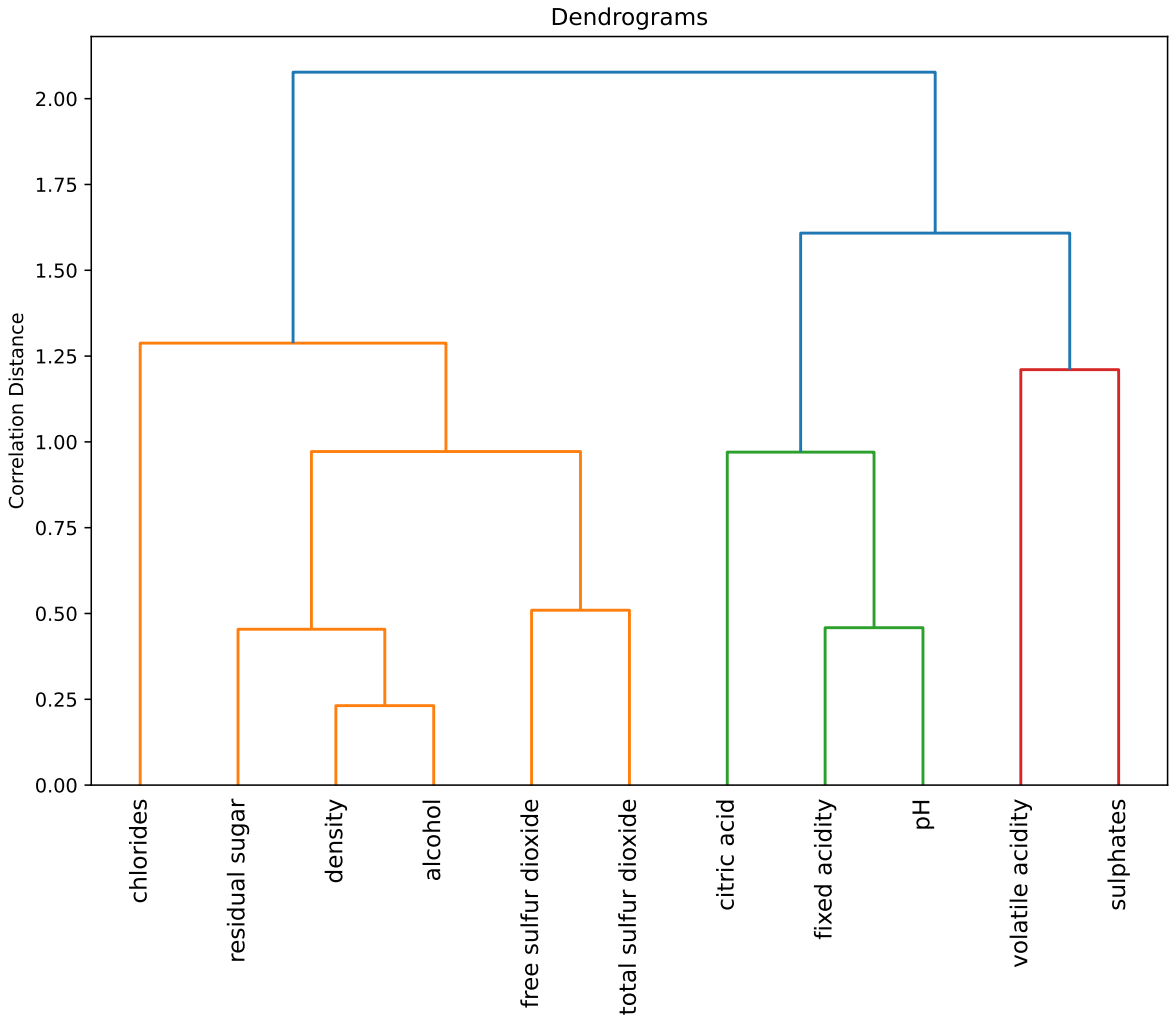


Figure 12.2: Hierarchically clustered features based on correlation

Figure Figure 12.2 shows the clustering results: density and alcohol are combined first, then merged with residual sugar, and finally with the cluster of free and total

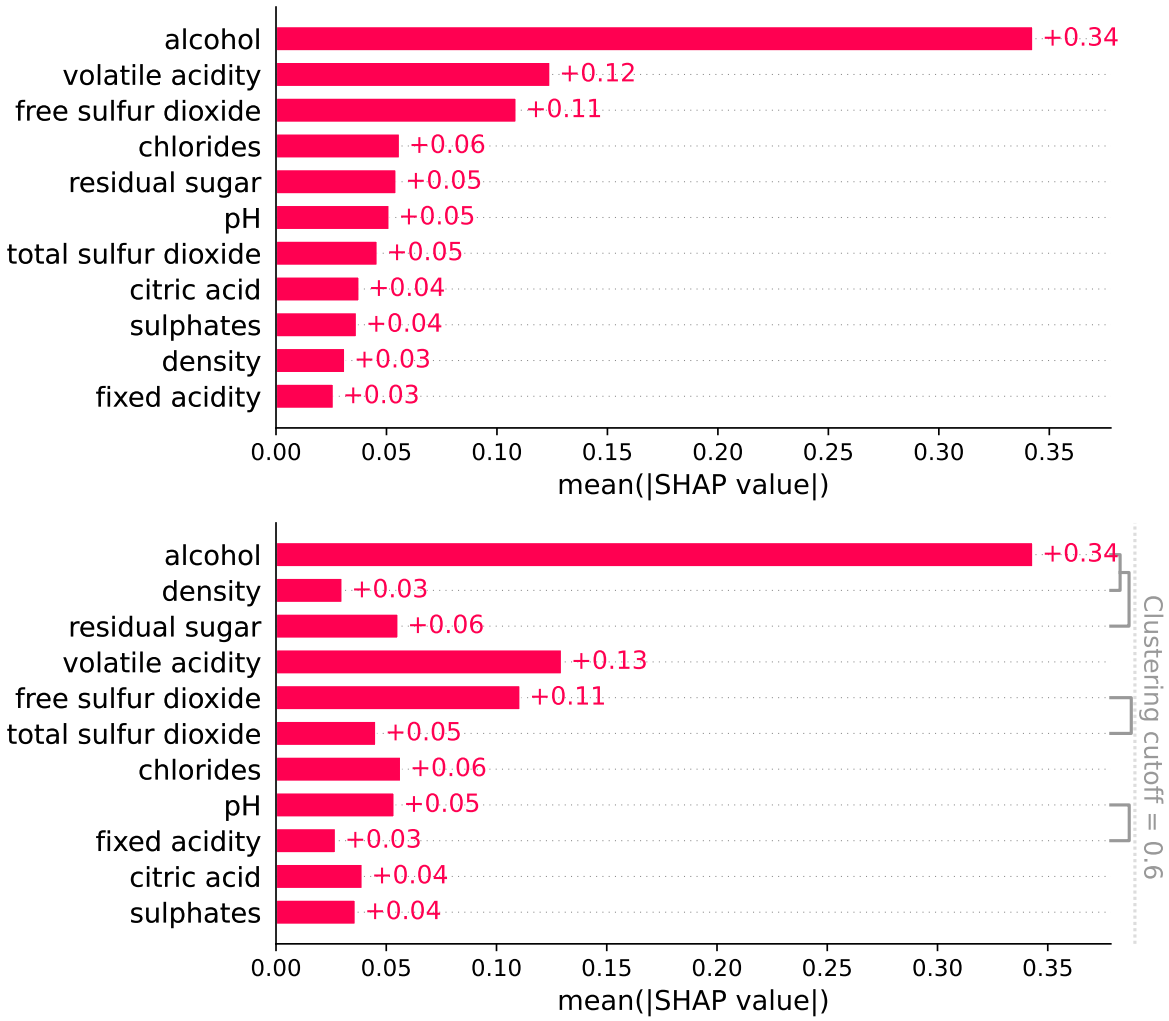
sulfur dioxide. As we ascend, the correlation weakens. This clustering hierarchy is input into the Partition Explainer to produce SHAP values:

```
masker = shap.maskers.Partition(X_train, clustering=clustering)
explainer2 = shap.PartitionExplainer(model.predict, masker)
shap_values2 = explainer2(X_test)
```

> **Note**
>
> By using the Partition Explainer instead of the Tree Explainer, we avoid the issue of checking additivity.

We now have our new SHAP values. The key question is: Do the results differ from when we ignored feature correlation? Let's compare the SHAP importances:

```
fig = plt.figure(figsize=(6,12))
ax0 = fig.add_subplot(211)
shap.plots.bar(shap_values, max_display=11, show=False)
ax1 = fig.add_subplot(212)
shap.plots.bar(
  shap_values2, max_display=11, show=False, clustering_cutoff=0.6
)
plt.tight_layout()
plt.show()
```



While the SHAP importances are not identical, the differences are not substantial. However, the real benefit is the new interpretation we gain from clustering and the Partition Explainer:

- We may add the SHAP values for both alcohol and density and interpret this as the effect for the alcohol and density group. There was no extrapolation between the two features, meaning no unlikely combination of alcohol and density was formed.
- Similarly, we may interpret the combined SHAP importance: we can interpret $0.34 + 0.03 = 0.37$ as the SHAP importance of the alcohol+density

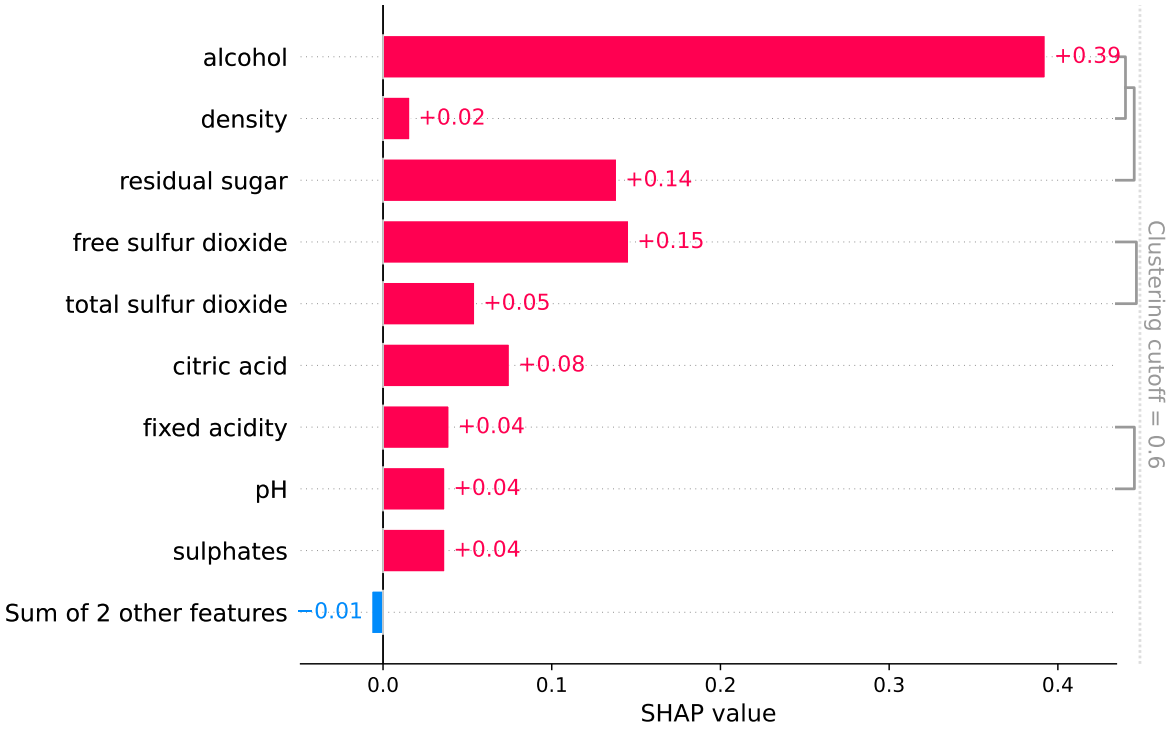
group.

- Free and total sulfur dioxide form a cluster with a combined importance of 0.16.
- Collectively, they are more important than volatile acidity and, due to their high correlation, we have a solid argument for analyzing them together.

As the user, you can decide how high up you go in the hierarchy by increasing the `clustering_cutoff` and then adding up the SHAP values (or SHAP importance values) for clusters. The higher the cutoff, the larger the groups, but also the more the correlation problem is reduced.

Now let's compare the SHAP explanation for the first data instance:

```
shap.plots.bar(shap_values2[0], clustering_cutoff=0.6)
```



Again, there are only slight differences in the SHAP values, and we can combine

the SHAP values of clusters in addition to interpreting the individual SHAP values. For the computation of a combined SHAP value, the features within that group were not subjected to extrapolation through marginal sampling. Revisit the Correlation Chapter for a refresher on this concept. For instance, the feature group "alcohol, density, and residual sugar" contributed a significant +0.55 (0.39 + 0.02 + 0.14) to the predicted quality. We know that for the group SHAP value of 0.55, alcohol, density, and residual sugar were always kept together in coalitions.

However, the individual SHAP values are still partially susceptible to extrapolation. For instance, the SHAP value for alcohol was computed by attributing 0.41 to both density and alcohol. For this attribution, density was also sampled by marginal sampling, which introduces extrapolation, such as combining high alcohol values with high density. So we have a trade-off between extrapolation and group granularity: The higher we ascend in the clustering hierarchy, the less extrapolation but the larger the feature groups become, which also complicates interpretation.

12.5 Understanding models for data subsets

Global interpretation is based on aggregating SHAP values. We can also use this aggregation to analyze data subsets. For example, we can examine wines with an alcohol content above 12, a useful subset of wines. In technical terms, this means that we subset the SHAP values and generate summary, dependence, and importance plots. However, when we wish to investigate a subset of data, such as wines rich in alcohol, should we also subset the background data used to estimate the SHAP values?

The choice of background data depends on the goal of interpretation:

- Are you interested in explaining the prediction difference compared to all wines?
- Or in comparison to the prediction of alcohol-rich wines?

Modifying the background data alters the value function. Wines with higher alcohol content have higher predicted qualities. A wine predicted to be of above-average quality may actually be below average if the average is calculated from alcohol-rich wines. In the first scenario, the SHAP values of the wines would sum

up to a positive value, whereas in the second scenario, they would sum up to a negative value. Let's explore the two methods of comparing subsets.

```
# create the data subsets
ind_test = np.where(X_test['alcohol'].values > 12)
ind_train = np.where(X_train['alcohol'].values > 12)
X_train_sub = X_train.iloc[ind_train]
X_test_sub = X_test.iloc[ind_test]

# SHAP where background data is based on subset
explainer_sub = shap.Explainer(model, X_train_sub)
shap_values_sub = explainer_sub(X_test_sub)

# SHAP where background data includes all wines
shap_values_sub_all = shap_values[ind_test]
```

We start with the SHAP values for a single wine.

```
import matplotlib.pyplot as plt

fig = plt.figure(figsize=(6,12))
ax0 = fig.add_subplot(211)
shap.plots.waterfall(shap_values_sub[1], show=False)
ax1 = fig.add_subplot(212)
shap.plots.waterfall(shap_values_sub_all[1], show=False)
plt.tight_layout()
plt.show()
```

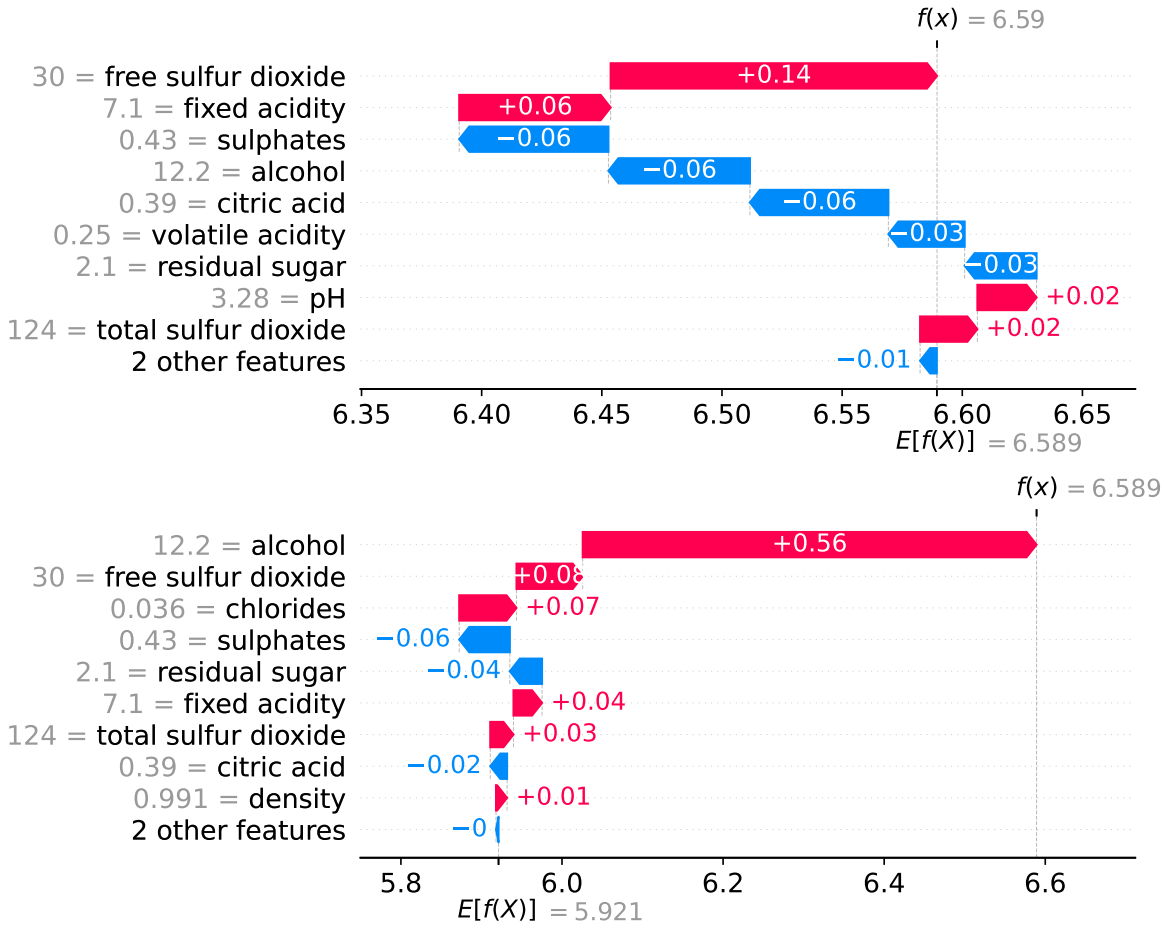


Figure 12.3: Top: Background data are wines with alcohol > 12. Bottom: Background data includes all wines

Figure 12.3 illustrates the distinct explanations produced by SHAP:

- While this wine's quality is above average when compared to all wines, it is below average in predicted quality when compared to alcohol-rich wines.
- Alcohol, typically the most impactful feature, is not relevant for this wine. This is logical because we conditioned the background data on alcohol. The question then becomes: How much did an alcohol level of 12.2 contribute to the prediction, compared to the average prediction for alcohol-rich wines?

- The reference changes: Since wines rich in alcohol are associated with higher predicted quality, $\mathbb{E}(f(X))$ is also higher when we use all wines as background data. This means that the SHAP values only need to explain a smaller difference of almost 0 instead of approximately 0.7.

> Interpretation template for subsets *(replace [] with your data)*
>
> The prediction [$f(x)$] for instance [i] differs from the average prediction of [$\mathbb{E}(f(X))$] for [subset] by [$f(x^{(i)} - \mathbb{E}(f(X))$] to which [feature name = feature value] contributed [$\phi_j^{(i)}$].

Here's an example of interpretation:

The predicted value of 6.59 for instance 1 differs from the expected average prediction of 5.92 for wines with an alcohol content greater than 12 by 0.67.

- free sulfur dioxide=30.0 contributed 0.136
- sulphates=0.53 contributed -0.062
- fixed acidity=7.1 contributed 0.063
- ...

The sum of all SHAP values equals the difference between the prediction (6.59) and the expected value (5.92).

Keeping the background data set for all wines and subsetting the SHAP values produces the same individual SHAP values, but it changes the global interpretations:

```
# sort based on SHAP importance for all data and all wines
ordered = np.argsort(abs(shap_values.values).mean(axis=0))[::-1]
plt.subplot(131)
shap.plots.beeswarm(
  shap_values, show=False, color_bar=False, order=ordered
)
plt.xlabel("")
plt.subplot(132)
shap.plots.beeswarm(
  shap_values_sub_all, show=False, color_bar=False, order=ordered
)
```

```
plt.gca().set_yticklabels([])  # Remove y-axis labels
plt.ylabel("")
plt.subplot(133)
shap.plots.beeswarm(
  shap_values_sub, show=False, color_bar=False, order=ordered
)
plt.gca().set_yticklabels([])  # Remove y-axis labels
plt.ylabel("")
plt.xlabel("")
plt.tight_layout()
plt.show()
```

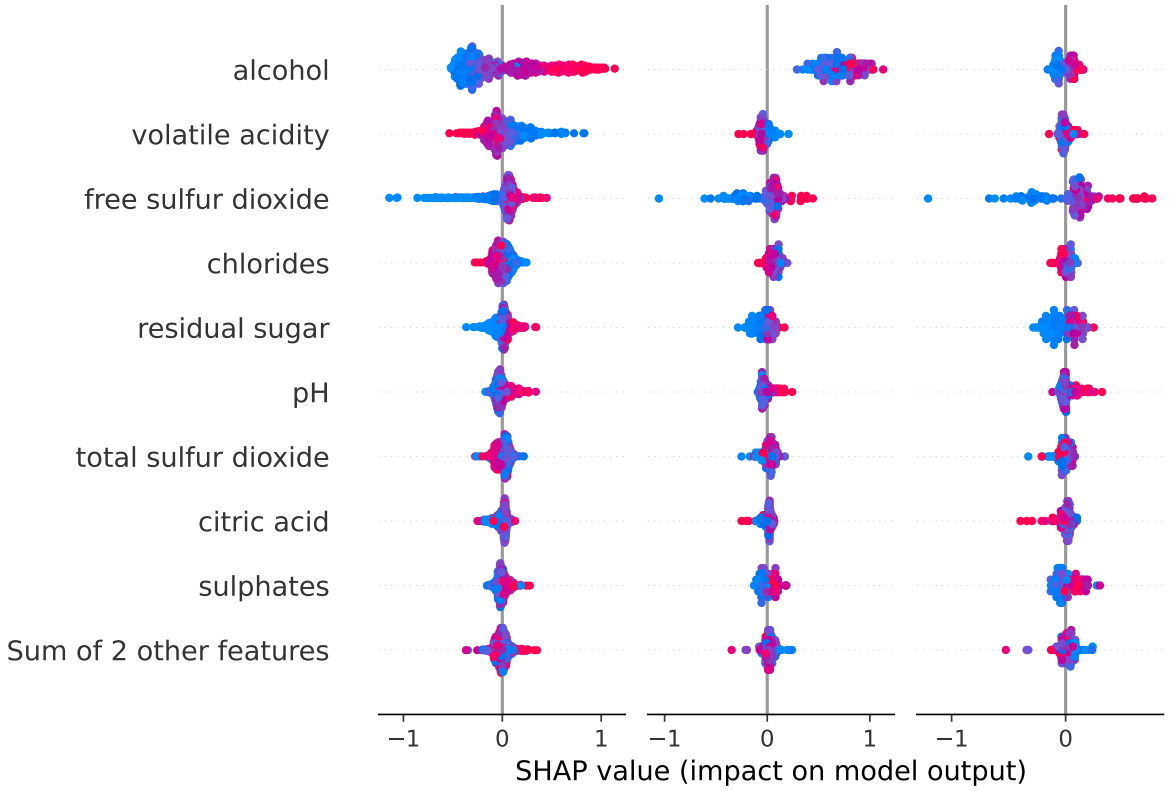


Figure 12.4: The left plot includes all SHAP values with all wines as background data. The middle plot contains SHAP values for wines high in alcohol with all wines as background data. The right plot displays SHAP values for wines high in alcohol with background data also comprising wines high in alcohol. The feature order for all plots is based on the SHAP importance of the left plot.

Figure 12.4 shows how subsetting SHAP values alone or together with the background data influences the explanations. Alcohol, according to SHAP, is the most crucial feature. Its importance remains when we subset SHAP values for wines high in alcohol. Its significance increases because these wines, high in alcohol, have a high predicted quality due to their alcohol content. However, when we also alter the background data, the importance of alcohol significantly decreases, as evidenced by the close clustering of the SHAP values around zero. More insights can be found in Figure 12.4. For instance, consider volatile acidity. A

higher volatile acidity typically correlates to lower SHAP values, but different patterns emerge when considering wines rich in alcohol. Firstly, the SHAP values of volatile acidity exhibit a smaller range. Moreover, some wines with high volatile acidity surprisingly present positive SHAP values, contradicting the usual relationship between volatile acidity and predicted quality.

Tip

Be inventive: Any feature can be employed to form subsets. You can even resort to variables that were not used as model features for subset creation. For example, you may want to examine how explanations change for protected attributes like ethnicity or gender, variables which you would not normally employ as features.

13 Image Classification with Partition Explainer

By the end of this chapter, you will be able to:

- Utilize SHAP for image models.
- Understand various methods to simulate the absence of image parts.

Up until now, we've explored tabular data. Now, let's explore image data.

Image classification is a common task typically solved using deep learning. Given an image, the model identifies a class based on the visible content. Rather than training our own image classifier, we'll utilize a pre-trained ResNet model (He et al. 2016) trained on ImageNet data (Deng et al. 2009). ImageNet is a large-scale image classification challenge where models are required to categorize objects within digital images. It boasts a dataset of over 1 million images from 1000 different categories, aiming to develop a model that accurately classifies each image.

This example's code is derived from a notebook from the `shap` library[1].

13.1 Importing the pretrained network

We'll employ a pre-trained ResNet from TensorFlow.

[1] https://github.com/slundberg/shap/blob/master/notebooks/image_examples/image_classification/Explain%20ResNet50%20using%20the%20Partition%20explainer.ipynb

```
import json
import numpy as np
import tensorflow as tf
from tensorflow.keras.applications.resnet50 import ResNet50
from tensorflow.keras.applications.resnet50 import preprocess_input
import shap

model = ResNet50(weights='imagenet')
X, y = shap.datasets.imagenet50()
```

We'll use the 50 images supplied by SHAP. One of these is a cheeseburger, which we'll encounter again later.

```
import json
import os
import urllib.request

json_file_path = 'imagenet_class_index.json'
# Verify if the JSON file is on disk
if os.path.exists(json_file_path):
    with open(json_file_path) as file:
        class_names = [v[1] for v in json.load(file).values()]
else:
    url = 'https://s3.amazonaws.com/deep-learning-models/' + \
          'image-models/imagenet_class_index.json'
    with urllib.request.urlopen(url) as response:
        json_data = response.read().decode()
    with open(json_file_path, 'w') as file:
        file.write(json_data)
    class_names = [v[1] for v in json.loads(json_data).values()]
```

13.2 Applying SHAP for image classification

Now, let's use SHAP to explain some image classifications. We already have our model, class names, and sample images.

What we need now are:

- A prediction function that wraps the model.
- A masker that defines how to simulate absent features (pixel clusters).

The prediction function is straightforward: it takes an image as a numpy array, preprocesses it for the ResNet, and feeds it to the ResNet. Thus, it takes a numpy array as input and outputs probability scores. We make use of the Partition Explainer that partitions the image into equal rectangles and recursively computes the SHAP values.

```
# Wrapping the model
def predict(x):
    tmp = x.copy()
    preprocess_input(tmp)
    return model(tmp)
```

Next, we define the masker:

```
masker = shap.maskers.Image(
  'blur(128,128)', shape = X[0].shape
)
```

The masker's role is to "remove" the pixels not included in the coalition. From its parameters, we can infer that it blurs the parts of the image that are absent. I will illustrate what this looks like later. First, let's calculate the SHAP values for two images: a cheeseburger and a pocket watch. Since this is a classification task, we need to decide the classes for which we want explanations. It's standard to select the top classes, especially the one with the highest probability, as it often has significance in subsequent stages of using the model.

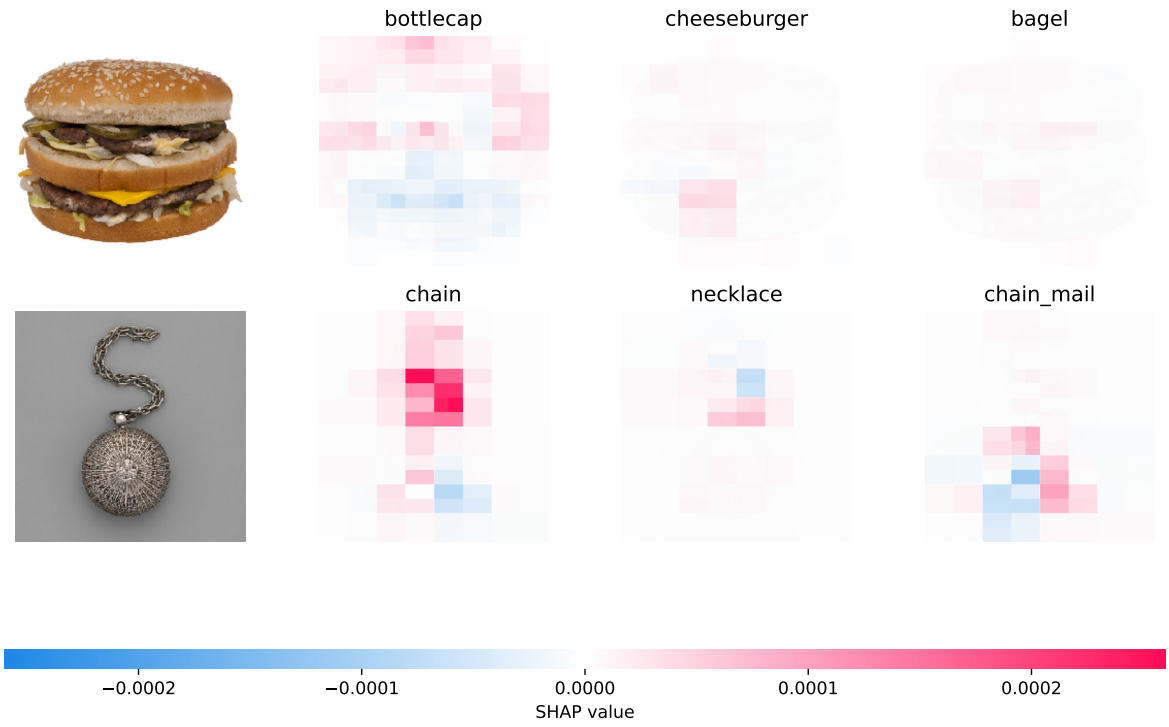
We'll select the top 3 classes (`topk`).

```
topk = 3

# which of the 50 images to display
index = [21, 28]
```

```
explainer = shap.Explainer(
    predict, masker, output_names=class_names
)

shap_values = explainer(
    X[index], max_evals=1000,
    batch_size=50,
    outputs=shap.Explanation.argsort.flip[:topk],
    silent=True
)
```

The above code calculates all SHAP values. However, with Image SHAP, the raw values aren't particularly useful, so let's visualize them. The SHAP values are optimally visualized by overlaying them on the original image. Below are the selected images:

```
shap.image_plot(shap_values, pixel_values=X[index]/255)
```



Before we dive into the interpretation of the SHAP values, notice the level on which we computed the SHAP values: The image is split into smaller rectangles, a bit like a chessboard, and we get a SHAP value for each rectangle. That's automatically done by the Partition Explainer. Now, let's interpret these SHAP values: The burger is misclassified as the top class is "bottlecap". The SHAP values show that the top part of the burger primarily influenced the bottlecap classification. The middle parts of the burger negatively impacted the classification, but the top part resembled a bottlecap too closely for the Resnet model. The second class, however, is cheeseburger, which is accurate. In this case, the middle parts mainly contributed to the correct classification. The second image is of a pocket watch. But, there's no pocket watch class among the 1000 ImageNet classes. Therefore, "chain" might be a reasonable classification. The chain part of the watch is the correct reason for this classification.

One disadvantage of calculating the SHAP values for multiple images is that the color scale for the Shapley values, seen at the bottom, applies to all images. As the watch has larger values, the values for the cheeseburger are scaled closer to white.

Let's learn about the different maskers, as they're more than just a "set-and-forget" parameter.

13.3 The impact of various maskers

Remember the theory of how SHAP values work: They measure how much features, or groups of features, contribute to coalitions of features. This means some feature values are present while others are absent. The main question is always how to simulate the absence of features.

For tabular data, it's about drawing from a background dataset, but there are other options for images. Ideally, you could replace the absent parts of the image with parts from randomly drawn datasets of other images. However, this might create bizarre random images that are hard to justify.

Another option is to blur out the absent pixels. This way, we don't completely remove the information, but "weaken" it. The blur's strength determines how much it's removed. Another method is inpainting, where we have no information

about the missing data but use algorithms to "guess" the missing part based on the rest of the image that isn't missing.

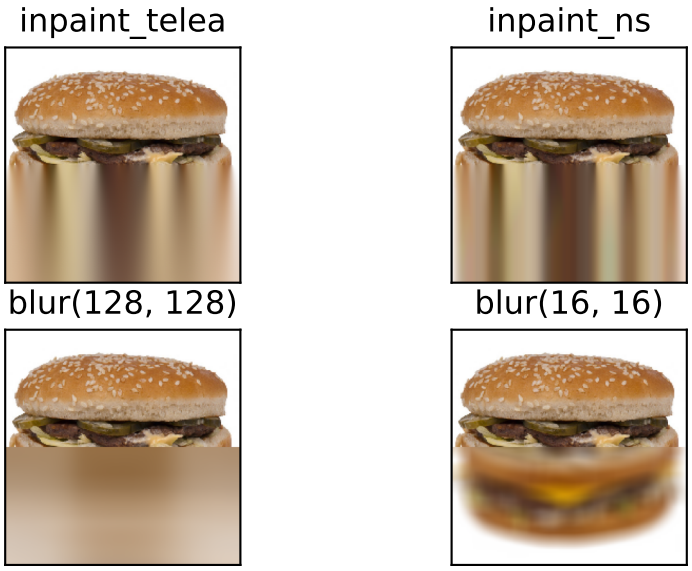
Specifically, here are the options in SHAP:

- `inpaint_telea`: Telea inpainting fills the missing area using a weighted average of neighboring pixels, based on a specified radius.
- `inpaint_ns`: NS (Navier-Stokes) inpainting is based on fluid dynamics and uses partial differential equations.
- `blur(16, 16)`: Blurring depends on kernel size, which can be set by the user. Larger values involve distant pixels in the blurring process, and blurring is faster than inpainting.

The `shap` package uses the `cv2` package for inpainting and blurring operations. Below are examples of how these different options look, primarily featuring melting burgers.

```
import matplotlib.pyplot as plt

# Define a masker to mask certain portions of the input image.
mask_names = ['inpaint_telea', 'inpaint_ns',
              'blur(128, 128)', 'blur(16, 16)']

sh = X[0].shape
masks = [shap.maskers.Image(m, sh) for m in mask_names]

# Create a numpy array of the shape (224, 224, 3)
arr = np.zeros((224 * 224 * 3), dtype=bool)
# Set the upper half of the image to True
arr[:75264] = True
# Set the lower half of the image to False
arr[75264:] = False

fig, axs = plt.subplots(2, 2)
ind = 0
for i in [0, 1]:
    for j in [0, 1]:
        axs[i, j].imshow(masks[ind](x=X[21], mask=arr)[0][0] / 255)
        axs[i, j].set_title(mask_names[ind])
```

```
        axs[i, j].set_xticks([])
        axs[i, j].set_yticks([])
        axs[i, j].tick_params(axis='both', which='both', length=0)
        ind += 1

plt.show()
```

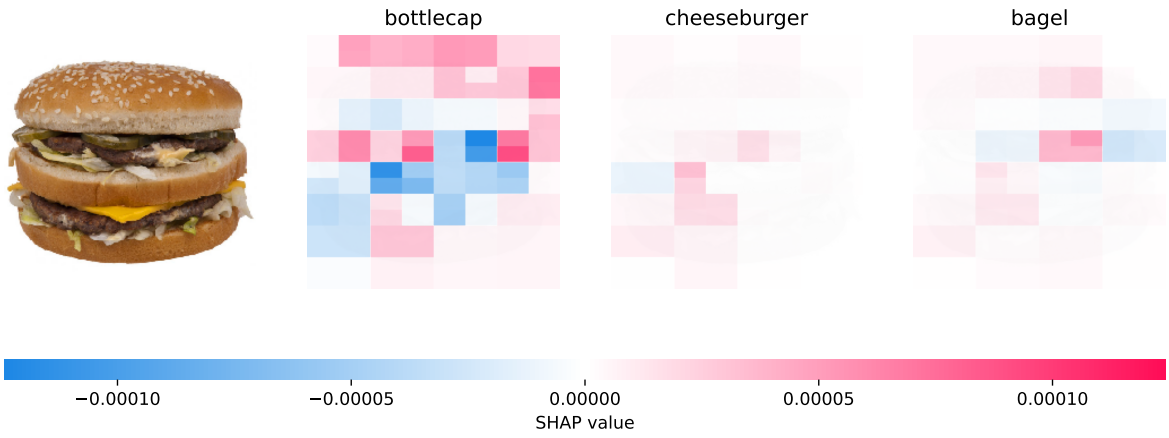


The upper half of the image is "present" and the lower half is "absent". This mirrors the SHAP values concept, where the image is divided into two "players": the upper and lower halves. This is also akin to the Partition Explainer, which further segments along the x and y axes.

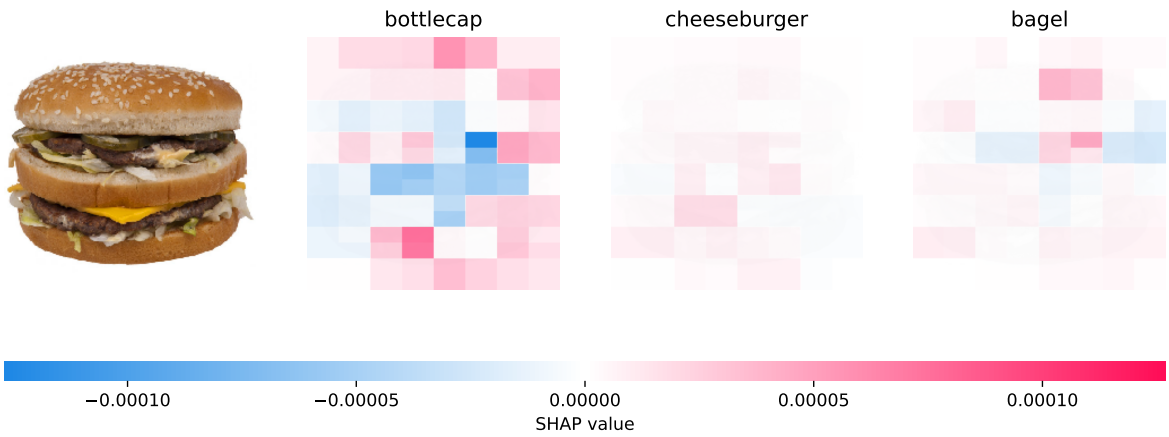
Unlike replacing all missing values with gray pixels, maskers don't entirely alter the image. Blurring even maintains the original data, merely smoothing it and causing some information loss. For instance, the blur(16, 16) kernel doesn't significantly alter the image; the bottom of the burger remains fairly recognizable. Next, let's examine the effect of the masks on the SHAP values.

```
topk = 3

# Iterate through all the masks
for mask_name in mask_names:
    print(mask_name)
    mask = shap.maskers.Image(mask_name, shape=sh)
    explainer = shap.Explainer(
        predict, mask, output_names=class_names, silent=True
    )
    shap_values = explainer(
      X[[21]], max_evals=500, batch_size=50,
      outputs=shap.Explanation.argsort.flip[:topk]
    )
    shap.image_plot(shap_values, pixel_values=X[[21]] / 255)
```

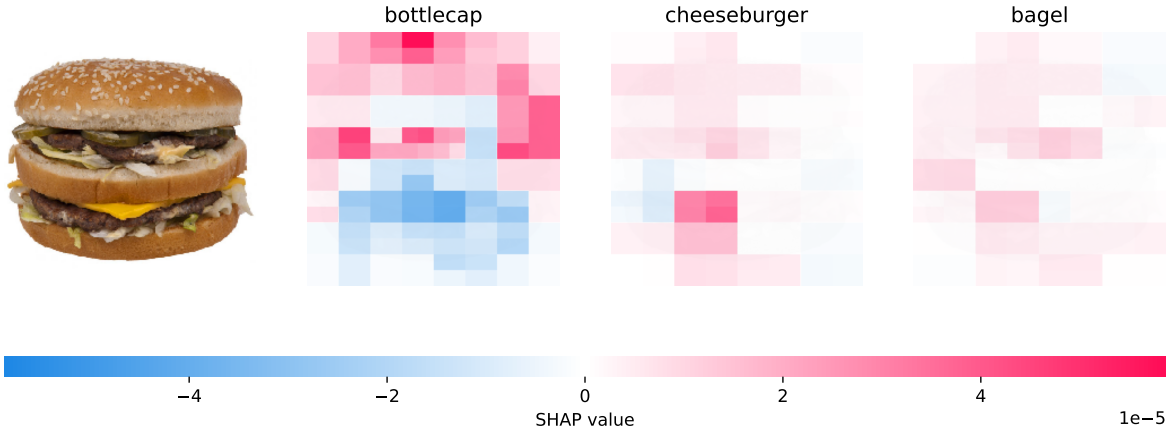
inpaint_telea



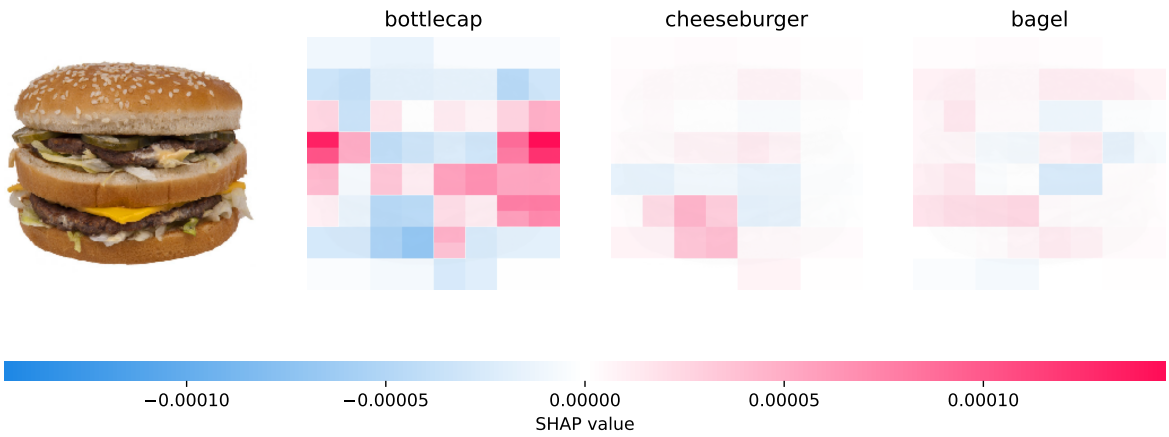
inpaint_ns



`blur(128, 128)`



`blur(16, 16)`



The masker choice does influence the explanation of the cheeseburger classification. The mistaken classification identified the cheeseburger as a bottle cap. All maskers concurred that the most influencing pixels were at the top, except for the blur(16, 16) masker, which highlighted the side pixels. Nonetheless, the blur(16, 16) masker appears somewhat unreliable, as it doesn't obscure much. Even to me, the bottom part of the image is recognizable as a burger, indicating that the features haven't been significantly obscured. In all instances, the pixels at the bottom of the burger seemed to counter the bottle cap classification. For more information on maskers, refer to the maskers chapter in the Appendix.

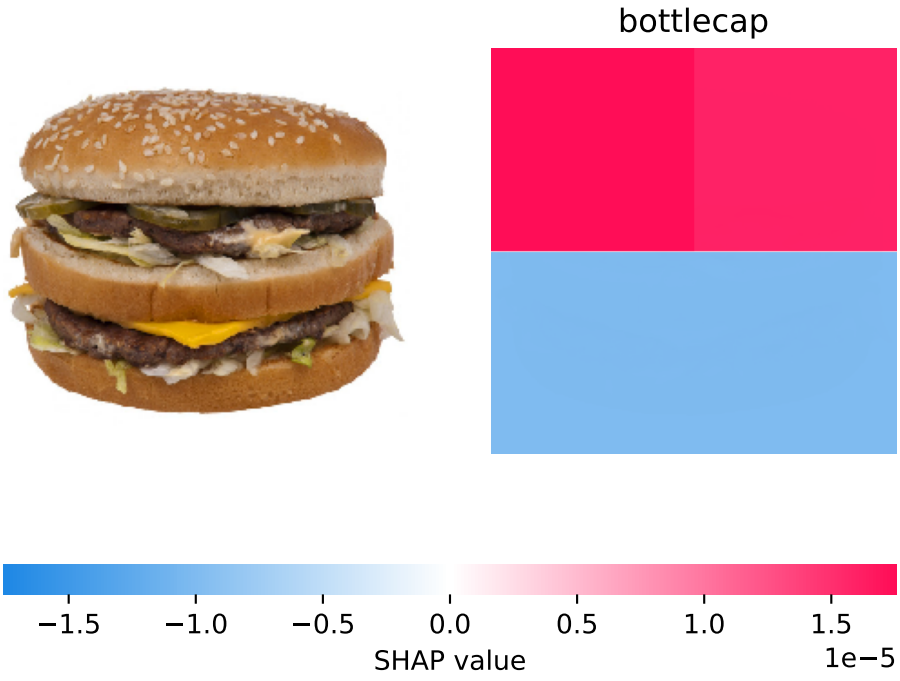
Another intriguing hyperparameter is the number of evaluation steps.

13.4 Effect of increasing the evaluation steps

The number of evaluations in SHAP for images affects the detail of the explanations. Given that the Image explainer is based on the Partition Explainer, more evaluations result in finer partitions and more localized superpixels. Let's investigate the impact of increasing the number of evaluation steps, beginning with just 10 evaluations:

```
topk = 1
masker = shap.maskers.Image(
```

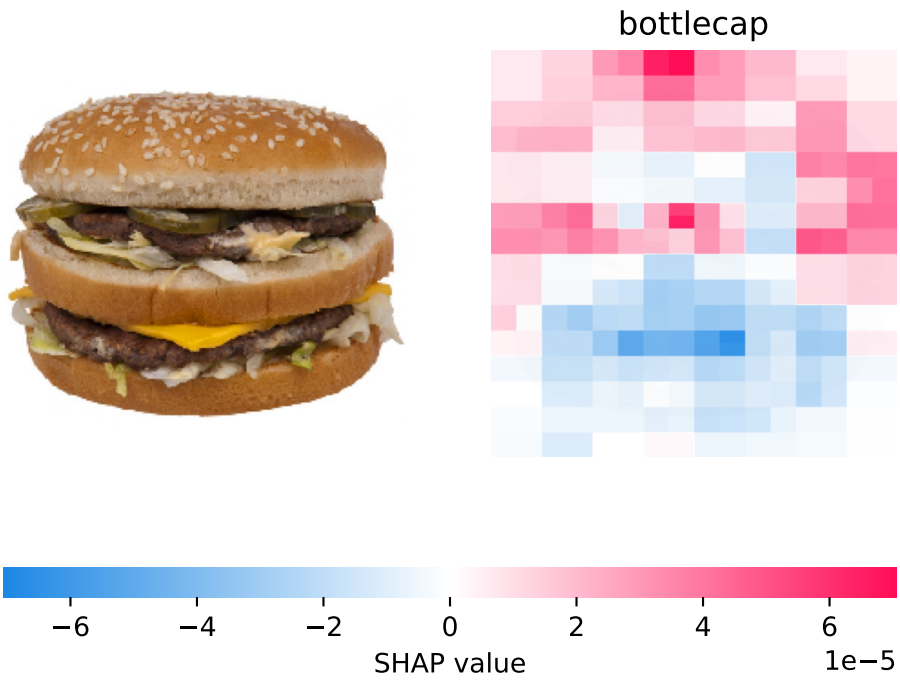
```
    'blur(128, 128)', shape=sh
)

explainer = shap.Explainer(predict, masker, output_names=class_names)
shap_values = explainer(
    X[[21]],
    max_evals=10,
    batch_size=50,
    outputs=shap.Explanation.argsort.flip[:topk]
)
shap.image_plot(shap_values, pixel_values=X[[21]]/255)
```



Then we increase the number of evaluations to 1000:

```
shap_values = explainer(
    X[[21]],
    max_evals=1000,
    batch_size=50,
    outputs=shap.Explanation.argsort.flip[:topk]
)
shap.image_plot(shap_values, pixel_values=X[[21]]/255)
```

```
  0%|          | 0/998 [00:00<?, ?it/s]
```



As we increase the number of evaluations, we notice:

- We acquire more detailed superpixels, which are associated with the Partition Explainer. With only 10 evaluations, we see 4 leaves, requiring a tree depth of 2, which leads to 4 individual SHAP values and 2 for the first split.

- The SHAP values decrease as the partitions and the number of pixels get smaller. This is because masking fewer pixels tends to have a lower impact on the prediction.
- The computation time grows linearly with the number of evaluations.

However, the Partition Explainer isn't the only option for image classifiers. In the next chapter, we'll discuss how to generate pixel-level explanations.

14 Deep and Gradient Explainer

By the end of this chapter, you will be able to:

- Apply SHAP on a pixel level for image models.
- Understand the differences between pixel-based SHAP and larger patch-based SHAP.
- Explain the Deep Explainer and the Gradient Explainer.

In the previous chapter, we explored the Partition Explainer, which treated larger image patches as features for SHAP. In this chapter, we will explain image classifier classifications using a different approach, akin to tabular data. This involves two key aspects:

- Assigning one SHAP value for each input pixel.
- Simulating feature absence (pixel absence) by sampling pixels from a background dataset.

Since we are working with a neural network, two model-specific tools are available:

- The Gradient Explainer, as neural networks often rely on gradients.
- The Deep Explainer, which utilizes neural network layers to backpropagate SHAP values.

Both methods are discussed in greater detail in the Estimation Appendix. For this example, we will use the MNIST dataset. The MNIST dataset contains 70,000 handwritten digits (0-9), each represented as a 28x28 pixel grayscale image. The goal of the MNIST task is to create a machine learning algorithm that can accurately classify these images into their corresponding digit categories. This well-established benchmark problem has been used to evaluate the performance

of various algorithms, including neural networks, decision trees, and support vector machines. Researchers in machine learning, computer vision, and pattern recognition have extensively used the MNIST dataset.

Why did I choose the MNIST dataset instead of ImageNet, as in the previous example? Because using pixel-wise explanations with Gradient or Deep Explainer requires sampling absent features using a background dataset.

Imagine using the ImageNet dataset, where we have an image of a burger and replace the "absent" pixels with pixels from a dog image – it would result in a strange outcome. However, for the MNIST dataset, this approach is more reasonable, as digits are more similar to each other, and replacing some pixels of a "2" with those of a "3" won't generate bizarre images. I acknowledge that this is a somewhat vague argument, but generally, explanations for images can be more challenging.

14.1 Training the neural network

Let's start by training a neural network from scratch using TensorFlow. The code is partially based on this shap notebook[1].

```
import tensorflow as tf
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import (
    Dense, Dropout, Flatten, Conv2D, MaxPooling2D
)
from tensorflow.keras.utils import to_categorical

# Load the MNIST dataset
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# Preprocess the data
x_train = x_train.reshape(x_train.shape[0], 28, 28, 1)
x_test = x_test.reshape(x_test.shape[0], 28, 28, 1)
```

[1] https://shap.readthedocs.io/en/latest/example_notebooks/image_examples/image_classification/Multi-input%20Gradient%20Explainer%20MNIST%20Example.html

```
x_train = x_train.astype('float32') / 255
x_test = x_test.astype('float32') / 255
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)

# Define model architecture
model = Sequential()
model.add(Conv2D(
  32,
  kernel_size=(3, 3),
  activation='relu',
  input_shape=(28, 28, 1)
))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(10, activation='softmax'))

# Compile model
model.compile(
  loss='categorical_crossentropy',
  optimizer='adam',
  metrics=['accuracy']
)

# Train model
model.fit(
  x_train,
  y_train,
  batch_size=128,
  epochs=5,
  validation_data=(x_test, y_test)
)
score = model.evaluate(x_test, y_test, verbose=0)
```

Next, we evaluate the model's performance:

```
# Evaluate model on test set
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

```
Test loss: 0.044951524585485 46
Test accuracy: 0.9843999743461609
```

14.2 Computing SHAP values with the Gradient Explainer

After training and evaluating our model, we'll explain its classifications using SHAP:

```
import shap
import time

# x_test is large, take a sample
x_sample = shap.sample(x_train, 500)

# Pass list of inputs to explainer since we have two inputs
explainer = shap.GradientExplainer(model, data=x_sample)

# Explain model's predictions on first three test set samples
start_time = time.time()
shap_values = explainer.shap_values(x_test[:3])
gradient_time = time.time() - start_time
```

The output, `shap_values`, is a list with a length equal to the number of classes (10 in this case):

```
print(len(shap_values))
```

```
10
```

For each model output, we obtain the SHAP values:

```
print(shap_values[0].shape)
```

```
(3, 28, 28, 1)
```

The first dimension represents the number of images for which we computed the SHAP values. The remaining dimensions contain the SHAP values in the form of an image, because the input data was an image.

Now, let's plot the SHAP values:

```
shap.image_plot(shap_values, x_test[:3])
```

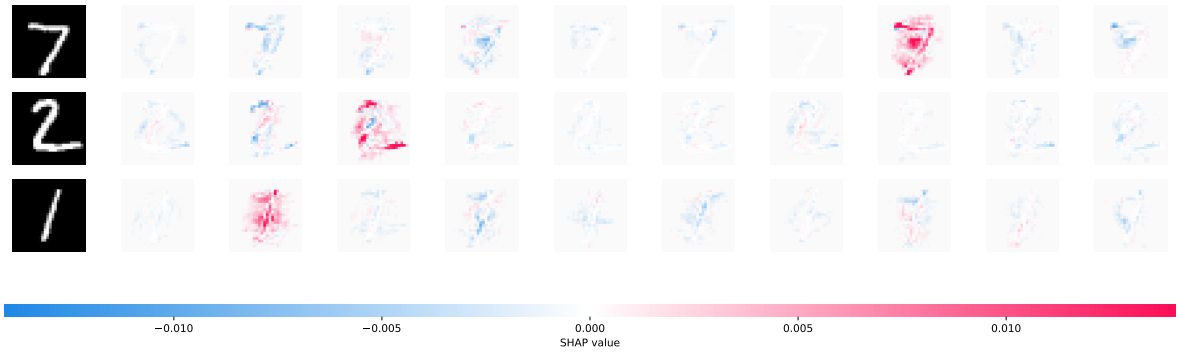


Figure 14.1: SHAP values for the input pixels of different input images (one per row). The first column shows the input image, then each column shows the SHAP values for the classes from 1 to 9.

In the plot, red pixels contributed positively to the respective class, while blue pixels contributed negatively. Grey pixels have a near zero SHAP value. The first row shows a 7 and, for example, in the 8th column we see positive contributions

of the pixels that make up a 7. The second row shows the image of a 2 and we can see that especially the start and end of the "2" contributed positively to the class "2" (3rd column). The start of the "2" and the slope in the middle contributed negatively to a prediction of "1".

14.3 SHAP with the Deep Explainer

We'll use the same process as above, but this time we'll use the DeepExplainer:

```
explainer = shap.DeepExplainer(model, data = x_sample)
start_time = time.time()
shap_values = explainer.shap_values(x_test[:3])
deep_time = time.time() - start_time
```

Then, we plot the results again:

```
shap.image_plot(shap_values, x_test[:3])
```

The SHAP values are very similar to the Gradient Explainer. This makes sense, since in both cases the result should be the same SHAP values and the difference is only due to the fact that both are approximations.

14.4 Time Comparison

We measured the time for both the gradient and deep explainer.

Let's examine the results:

```
print('Gradient explainer: ', round(gradient_time, 2))
print('Deep explainer: ', round(deep_time, 2))
```

```
Gradient explainer:  0.97
Deep explainer:  5.3
```

In theory, to obtain a reliable time comparison, you should repeat the calls several hundred times and avoid using other programs simultaneously. However, this comparison was conducted only once on my MacBook with an M1 chip. Keeping this in mind, we see a bit of a difference. Since only the CPU is used, the efficiency of calling the model could improve if a GPU were involved. But, I'd rather use the Gradient Explainer.

15 Explaining Language Models

By the end of this chapter, you will be able to:

- Apply SHAP for text classification and text generation models.
- Select a tokenization level for interpretation.

Let's explore text-based models. All models in this chapter have one thing in common: their inputs are text. However, we'll encounter two distinct types of outputs:

- Scores, such as in classification and sentiment analysis.
- Text, such as in text generation, translation, and summarization.

While they might seem different at first glance, they're quite similar upon closer inspection. We'll start with the simpler case where a model outputs a single score for a text input, like in classification, for instance, determining the category of a news article.

In this chapter, we'll mainly work with transformers, which are state-of-the-art for text-based machine learning. However, keep in mind, SHAP values are model-agnostic. So it doesn't matter if the underlying model is a transformer neural network or a support vector machine that works with engineered features, like TF-IDF (Term Frequency-Inverse Document Frequency).

15.1 How SHAP for text works

As with other applications of SHAP, the aim is to attribute the prediction to the inputs. SHAP requires a scalar output, which is straightforward when dealing with a text classifier. For sequence-to-sequence models, the output isn't inherently

scalar, but we can make it so by examining the score for a specific token instead of the word.

The features in both text classification and text-to-text models are text-based. However, it's not as simple as it sounds, because it's not words that are fed into the neural network but numbers. In the case of state-of-the-art neural networks, these numbers are represented as embedding tokens. Tokens are typically smaller than words, and there are numerous methods to tokenize text.

15.2 Defining players in text

As seen in the Image Chapter, we can use different levels of granularity for SHAP inputs than those used for the model. This provides us with multiple options for computing SHAP values:

- By character
- By token
- By word
- By sentence
- By paragraph
- And everything in between

The choice depends on the specific application, and we'll explore various examples throughout this chapter. Consider the task of sentiment analysis. The sentence "I returned the item as it didn't work." might have a predicted score of -0.5 indicating a negative sentiment.

The aim of SHAP is to attribute this score to the input words. If you choose to attribute the prediction at the word level, you will obtain one SHAP value for each word: ["I", "returned", "the", "item", "as", "it", "didn't", "work"].

Each word acts as a team player, and the -0.5 score is fairly distributed among them.

15.3 Removing players in text-based scenarios

An interesting question arises: how do you simulate the absence of players/features in text? In theory, you have multiple options:

- Remove the word.
- Replace it with a fixed token (e.g., "…").
- Replace it with a draw from background data.

SHAP implements options 1 and 2. Options 1 and 2 are more reasonable than option 3, as option 3 could introduce significantly different texts. For example, to assess the impact of the word "returned" on the negative sentiment prediction, we would include it in different teams. Assuming that missing words are replaced with "…", and multiple adjacent words are replaced with a single "…", we get the following coalitions for computing marginal contributions.

- "I … the … as it didn't work" -> "I returned the … as it didn't work"
- "… the item … work" -> "… returned the item … work"
- "…" -> "… returned …"

With this theoretical knowledge in hand, let's proceed to a text classification example.

15.4 Text classification

Let's start with a straightforward example of a classification task. We've already discussed classification, but now our input is text. A classic example is sentiment analysis, which, while it has its own name, is essentially just classification with predetermined labels: positive and negative.

We'll use Hugging Face transformers, which simplifies the implementation:

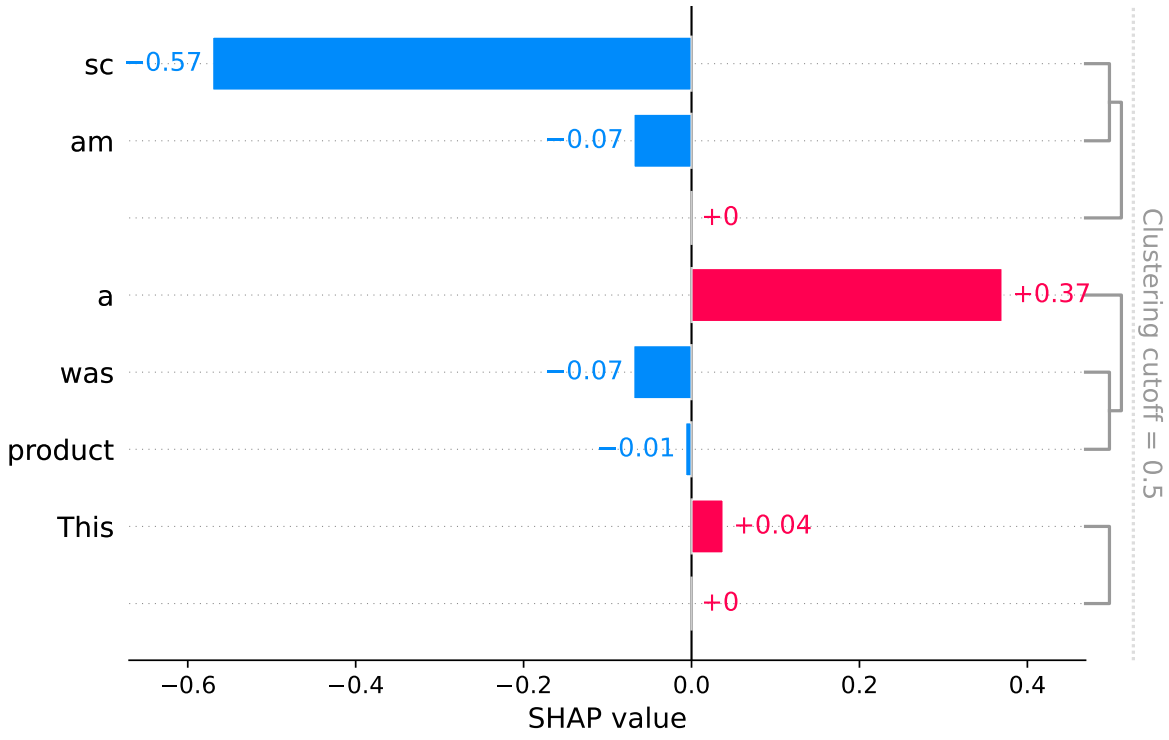
```
from transformers import pipeline
model = pipeline('sentiment-analysis', return_all_scores=True)
s = ['This product was a scam']
print(model(s)[0][1])
```

```
{'label': 'POSITIVE', 'score': 0.00033915787935256 96}
```

As anticipated, the statement “this product was a scam” is classified as negative. The keyword “scam” would naturally lead us to this conclusion, but let’s find out whether this was the reason for the model’s classification and if SHAP can offer any insight.

```
import shap
explainer = shap.Explainer(model)
shap_values = explainer(s)
print("expected: %.2f" % shap_values.base_values[0][1])
print("prediction: %.2f" % model(s)[0][1]['score'])
shap.plots.bar(shap_values[0, :, 'POSITIVE'])
```

```
expected: 0.31
prediction: 0.00
```



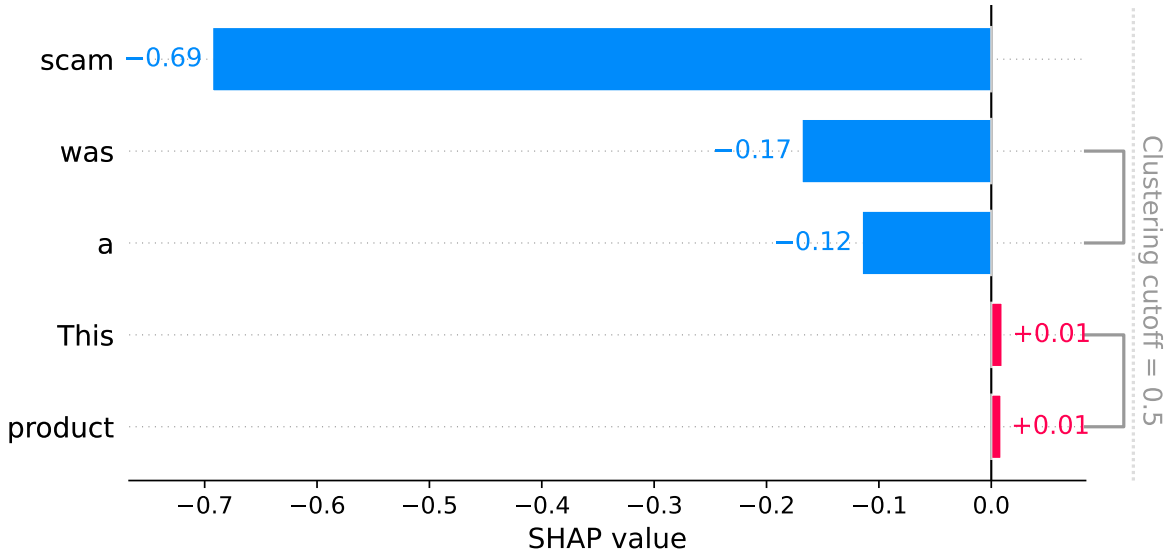
That was straightforward, wasn't it? The term "sc" appears to contribute the most to the negative sentiment. However, the splitting of "scam" into "sc" and "am" isn't ideal for interpretation. This issue emerges from our masking of the input, in which the choice of tokenizer influences how the text is masked. We can see that the Partition explainer was used since the clustering is displayed here as well. So we can just add up the two SHAP values of "sc" and "am" to get the SHAP value for "scam", but it would be more elegant to compute SHAP values based on better tokenization.

15.5 Experimenting with the masker

Let's modify the tokenizer as previously discussed. In the following code, I present a custom tokenizer that partitions the text into words. This is achieved using maskers, the SHAP abstraction that simulates the absence of features. Next, we supply both the model and the masker to SHAP.

Let's review the results:

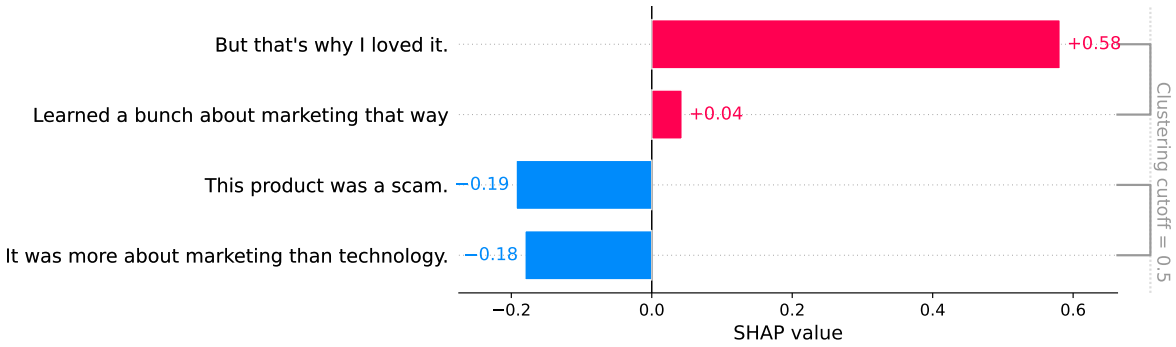
```
masker = shap.maskers.Text(tokenizer=r"\W+")
explainer = shap.Explainer(model, masker=masker)
shap_values = explainer(s)
shap.plots.bar(shap_values[0, :, 'POSITIVE'])
```



Now it's evident that "scam" is the most relevant term for a negative classification. The tokenizer is highly adaptable. To illustrate this, let's consider another example using SHAP values calculated on sentences. In this scenario, the tokenizer is simple and breaks the input at periods ".". We'll experiment with a lengthier input text to observe the contribution of each sentence.

```
s2 = "This product was a scam." + \
     "It was more about marketing than technology." + \
     "But that's why I loved it." + \
     "Learned a bunch about marketing that way."
masker = shap.maskers.Text(tokenizer=r"\.", mask_token=" ")
explainer = shap.Explainer(model, masker=masker)
shap_values = explainer([s2])
print("expected: %.2f" % shap_values.base_values[0][1])
print("prediction: %.2f" % model(s2)[0][1]['score'])
shap.plots.bar(shap_values[0, :, 'POSITIVE'])
```

```
expected: 0.75
prediction: 1.00
```



"But that's why I loved it" has a huge positive contribution to the sentiment, more than the two negative sentences combined. I've used a whitespace " " as the masking token, which seems suitable for dropping a sentence. The default replacement token for text is "...", but generally, if a tokenizer is provided, the .mask_token attribute is utilized, assuming the tokenizer has this attribute.

To illustrate "extreme" masking, let's replace a removed sentence with a specific one instead of leaving it blank. The `collapse_mask_token=True` argument ensures that if two tokens in a row are replaced by the mask_token, the token is only added once. In the ensuing example, sentences are replaced with "I love it", but only once consecutively.

Consider the sentence: "This product was a scam. It was more about marketing than technology. But that's why I loved it. Learned a bunch about marketing that way." Let's analyze the marginal contribution of "Learned a bunch about marketing" when added to an empty set, by comparing these two sentences:

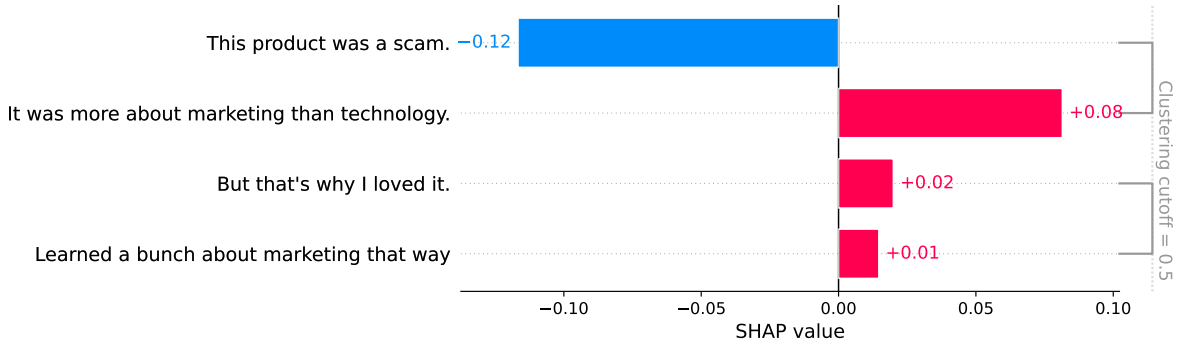
"I love it. Learned a bunch about marketing that way." versus "I love it."

If `collapse_mask_token=False`, we would compare "I love it. I love it. I love it. Learned a bunch about marketing that way." with "I love it. I love it. I love it. I love it." Therefore, it often makes sense to set `collapse_mask_token` to True. In theory, you could also create a custom masker.

```
masker = shap.maskers.Text(tokenizer=r"\.", mask_token="I love it",
                           collapse_mask_token=True)
explainer = shap.Explainer(model, masker=masker)
shap_values = explainer([s2])
```

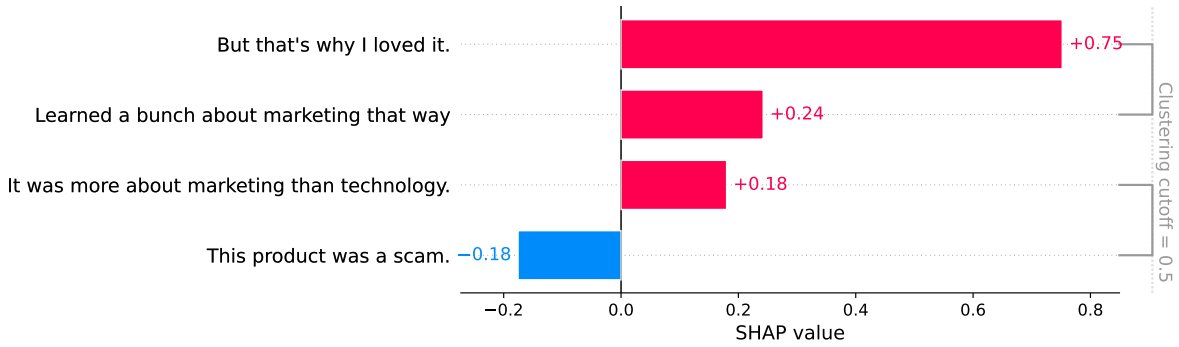
```
print("expected: %.2f" % shap_values.base_values[0][1])
print("prediction: %.2f" % model(s2)[0][1]['score'])
shap.plots.bar(shap_values[0, :, 'POSITIVE'])
```

```
expected: 1.00
prediction: 1.00
```




```
masker = shap.maskers.Text(
  tokenizer=r"\.",
  mask_token='I hate it',
  collapse_mask_token=True
)
explainer = shap.Explainer(model, masker=masker)
shap_values = explainer([s2])
print("expected: %.2f" % shap_values.base_values[0][1])
print("prediction: %.2f" % model(s2)[0][1]['score'])
shap.plots.bar(shap_values[0, :, 'POSITIVE'])
```

```
expected: 0.00
prediction: 1.00
```



Here, the replacement acts as a reference. In one scenario, any sentence removed from the coalition is replaced with "I love it," and in the other scenario, it's replaced with "I hate it."

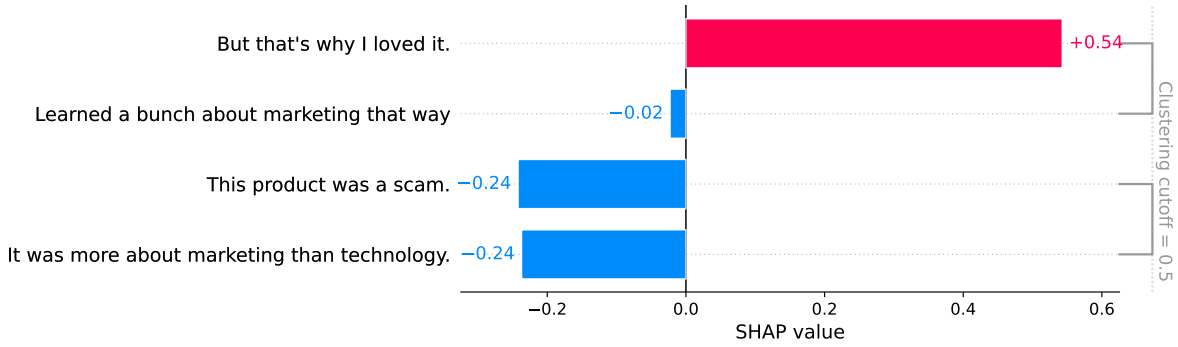
What changes are the base values; they shift from strongly positive to negative, as can be inferred from the difference in the base value. Every sentence is now interpreted in contrast to the replacement. This was also true earlier, but previously we replaced it with an empty string, which is more neutral than the sentences provided.

> **Warning**
>
> Avoid using extreme masking tokens as they might not make sense. However, more specific tokens can be beneficial. This highlights the importance of masking, which serves as background data. Consider the replacement carefully and test alternatives if necessary.

There's a difference when replacing tokens with " " or "...":

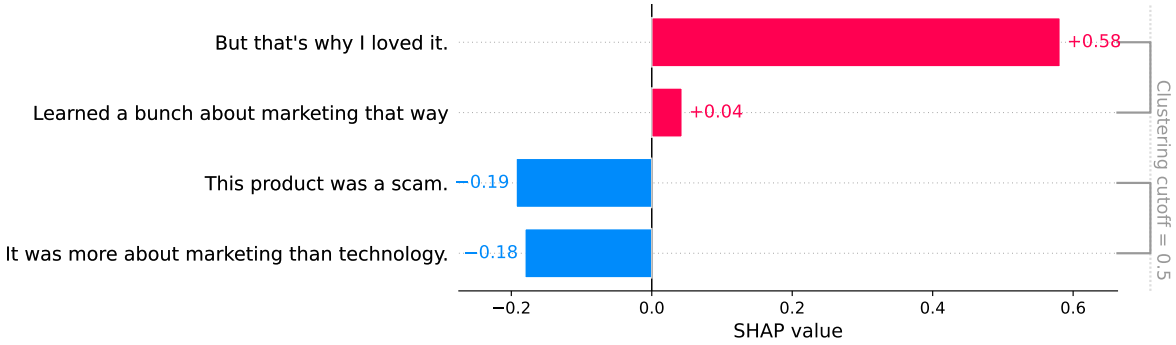
```
masker = shap.maskers.Text(
  tokenizer=r"\.", mask_token='...', collapse_mask_token=True
)
explainer = shap.Explainer(model, masker=masker)
shap_values = explainer([s2])
print("expected: %.2f" % shap_values.base_values[0][1])
print("prediction: %.2f" % model(s2)[0][1]['score'])
shap.plots.bar(shap_values[0, :, 'POSITIVE'])
```

```
expected: 0.96
prediction: 1.00
```




```
masker = shap.maskers.Text(
  tokenizer=r"\.", mask_token=' ', collapse_mask_token=True
)
explainer = shap.Explainer(model, masker=masker)
shap_values = explainer([s2])
print("expected: %.2f" % shap_values.base_values[0][1])
print("prediction: %.2f" % model(s2)[0][1]['score'])
shap.plots.bar(shap_values[0, :, 'POSITIVE'])
```

```
expected: 0.75
prediction: 1.00
```



Despite the overall attribution not changing significantly, except for the sign change in the "marketing" sentence, which was close to zero, the base value changes considerably.

Experiment with it, generate some text, and make a qualitative judgment about whether it makes sense. To understand more about maskers, refer to the maskers chapter in the Appendix.

15.6 Using logits instead of probabilities

The output falls in the probability space between 0 and 1, necessitating a logit transformation. Additive explanations perform better on linear scales, such as logits, which occur just before the 0 to 1 squeezing. SHAP provides a wrapper for transformers that allows specifying whether to use logits or probabilities:

```
model2 = shap.models.TransformersPipeline(
  model, rescale_to_logits=True
)
```

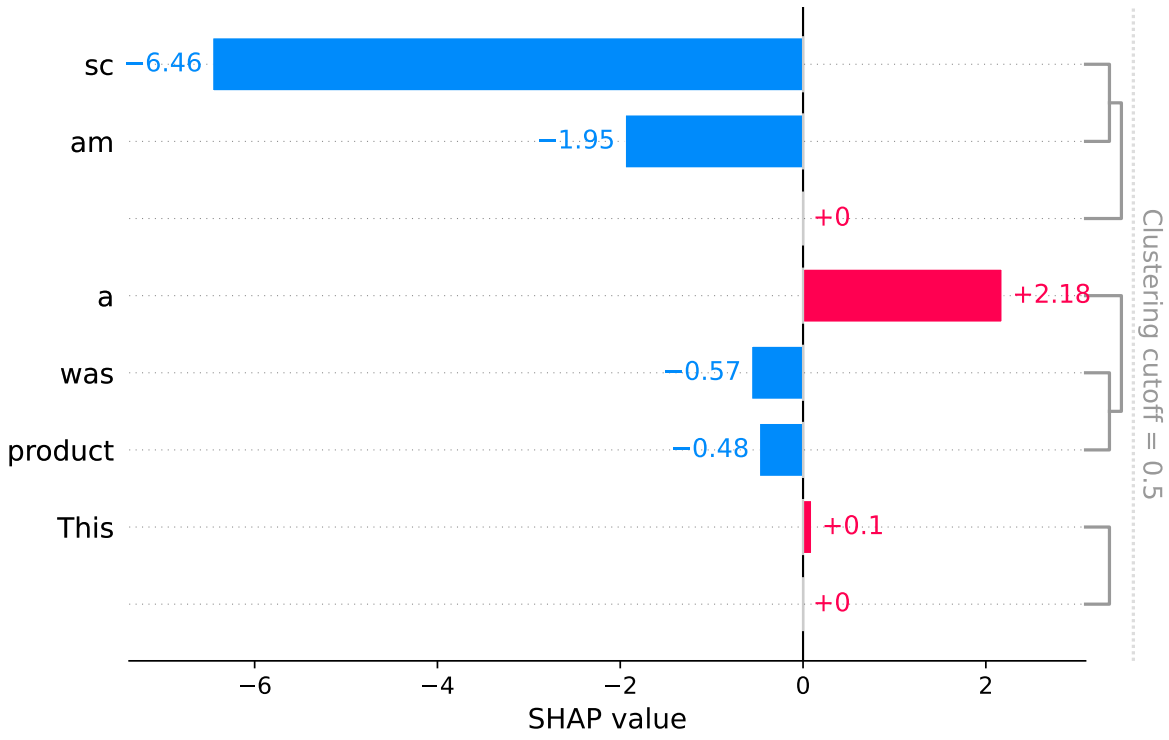
Like the original transformer, you can make predictions with this model:

```
model2(s)
```

```
array([[ 7.9887276 , -7.98870562]])
```

Now let's see how this impacts the explanations with SHAP:

```
explainer2 = shap.Explainer(model2)
shap_values2 = explainer2(s)
shap.plots.bar(shap_values2[0,:, 'POSITIVE'])
```



This result is quite similar.

15.7 How SHAP interacts with text-to-text models

Text-to-text models are unique in that they generate multiple outputs. Each token produced by the neural network is treated as an individual prediction.

This can be viewed as a classification task where the goal is to determine the next token based on some text input. Consider early large language models that aimed to produce the next words given an input text. For example:

Input text: "Is this the Krusty Krab?" Output text: "No! This is Patrick!"

In the context of text-to-text models, each output token is considered an individual prediction, much like in multi-class classification. We can compute SHAP values for each token.

If the tokenized input has a length of n and the tokenized output length is m, we derive $n \cdot m$ SHAP values. The level of input tokenization is user-controllable. In the example above, if the user opts for word-level tokenization for the input, the first token of the output, i.e., "No", receives n SHAP values. The next token "!" gets n SHAP values, and so on.

15.8 Explaining a text-to-text model

Let's explore text generation, a fairly general task. Again, this is implemented in the Hugging Face transformers package. It's a large language model similar to those that power GPT-3 and GPT-4. For a locally runnable example that doesn't require an account, we'll use a less powerful model. This doesn't change the general interface of text input and output. The key difference is that we get not only the words but also their scores, crucial for calculating SHAP values. In this case, we'll use GPT-2, automatically selected by the transformers library for the "text-generation" task at the time of writing. The following code is partially based on this shap notebook[1].

```
from transformers import AutoTokenizer, AutoModelForCausalLM

tokenizer = AutoTokenizer.from_pretrained('gpt2')
model = AutoModelForCausalLM.from_pretrained('gpt2')
```

Next, we decide on the text to complete.

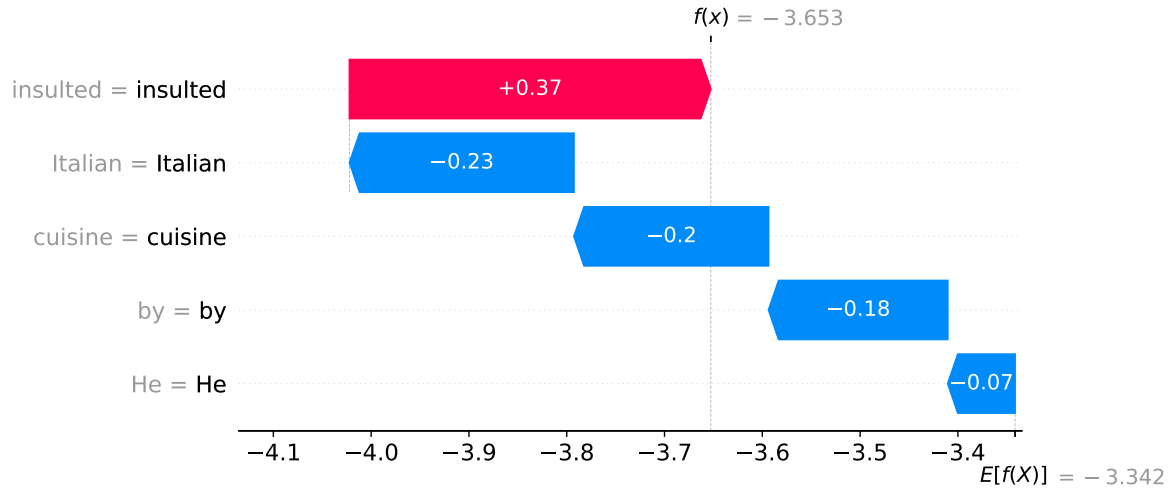
[1] https://shap.readthedocs.io/en/latest/example_notebooks/text_examples/text_generation/Open%20Ended%20GPT2%20Text%20Generation%20Explanations.html

```
import torch

# Set seed for consistent results
torch.manual_seed(0)

input_text = 'He insulted Italian cuisine by'

# Encode input text
input_ids = tokenizer.encode(input_text, return_tensors='pt')

# Sample instead of returning the most likely token
model.config.do_sample=True
# Set maximum length
model.config.max_new_tokens = 30

# Generate text with stop_token set to "."
output = model.generate(input_ids)

# Decode output text
output_text = tokenizer.decode(output[0], skip_special_tokens=True)
print('The result: "' + output_text + '"')
```

The result: “He insulted Italian cuisine by throwing spaghetti at the tables and throwing rocks at the local biker’s peloton, and he did something about it by showing that she needs a change”

Now, we can obtain the SHAP explanations for the first generated word “throwing”.

```
torch.manual_seed(0)
# Setting the model to decoder to prevent input repetition.
model.config.is_decoder = True
explainer = shap.Explainer(model, tokenizer)
shap_values = explainer([input_text])
shap.plots.waterfall(shap_values[0, :, 5 + 1])
```



The word "insulted" positively contributed to this word, while all others had a negative contribution.

15.9 Other text-to-text tasks

Here are other examples of text-to-text tasks for which notebooks are available:

- Question Answering[2]
- Summarization[3]
- Machine Translation[4]

As general text-to-text models can now handle these tasks, the given example should suffice.

[2] https://shap.readthedocs.io/en/latest/example_notebooks/text_examples/question_answering/Explaining%20a%20Question%20Answering%20Transformers%20Model.html
[3] https://shap.readthedocs.io/en/latest/example_notebooks/text_examples/summarization/Abstractive%20Summarization%20Explanation%20Demo.html
[4] https://shap.readthedocs.io/en/latest/example_notebooks/text_examples/translation/Machine%20Translation%20Explanations.html

16 Limitations of SHAP

By the end of this chapter, you will be able to:

- Recognize the limitations of SHAP values.
- Resist the allure of hype.

As a statistician holding both a Bachelor's and a Master's degree, I feel obligated to critically examine methods and resist hype. In this chapter, we will scrutinize SHAP. While SHAP is popular, it is not without its shortcomings. We have already discussed issues such as the correlation problem and the complexity of interpreting interactions. Let's dive further into these limitations.

16.1 Computation time can be excessive

Kernel and Sampling estimators can be particularly slow. Depending on your use case, computing SHAP values might be unnecessary. If you only need to determine the global effect of a feature, it would be more efficient and cost-effective to compute partial dependence plots (PDPs)[1]. However, model-specific versions like the Tree Estimator are relatively fast, making them a viable option if feasible.

16.2 Interactions can be perplexing

The argument that interactions can be confusing comes from Kumar et al. (2020), who claimed that the additivity axiom can be counterintuitive. They studied the

[1] https://christophm.github.io/interpretable-ml-book/pdp.html

function $f(x) = \prod_{j=1}^{p} x_j$, which served as the "model" but was actually a predefined function for studying SHAP values. This function is purely multiplicative. All features were independent and had an expected value of zero. Given this information, what would you anticipate for the SHAP importance of each feature? I would expect all SHAP importances to be equal. However the two features have different scales, such as X_1 ranging from -1 to +1 and X_2 ranging from -1000 to +1000. While X_2 appears to have a greater influence on the prediction due to its larger scale, the SHAP importance for both features is the same. This is because knowing X_1 is as important as knowing X_2. X_2 has a much larger range, but it doesn't impact the SHAP value because X_1 can set the prediction to zero or flip the sign. This shows that interpreting interactions is not always intuitive. In the Interaction Chapter, we also observed how interactions are divided between features and often exhibit a combination of local and global effects. Keep this peculiarity in mind.

16.3 No consensus on what an attribution should look like

Another issue, which extends beyond SHAP, is the lack of a cohesive understanding of what an attribution method should entail. There is uncertainty about how interactions should be attributed or what "importance" truly represents.

In interpretable machine learning, the process is often reversed: We use a somewhat mathematically coherent method that generates an "explanation" for a model and then attempt to decipher its meaning. Ideally, we would specify what we want in terms of an explanation and then select or design the method based on our needs. However, this is rarely the case in the real world.

With SHAP values, one could argue that the axioms perfectly define them. After all, SHAP values are the only solution that satisfies the axioms. The question now is: Are these axioms useful for interpretation? Did we define the right game?

As demonstrated in the Linear Chapter and the Additive Chapter, at least for these more restricted models, the results align with our expectations and what similar metrics (such as coefficients) would indicate.

Bear in mind that SHAP values are but one example of an attribution method. And even attribution methods are just a subset of methods in interpretable machine learning.

16.4 SHAP values don't always provide human-friendly explanations.

Human-friendly explanations should be concise, contrastive, and focus on "abnormal" causes (Miller 2019). However, SHAP values are not concise because they attribute significance to all feature values. They are not inherently contrastive either. While they provide contrast to some background data, this can be difficult to comprehend due to their averaging nature over numerous coalitions, which tends to dilute the contrastiveness. The fundamental building block of SHAP values, the marginal contribution (effect of adding a feature value to a coalition), is contrastive when considering a single background data point. As for focusing on the "abnormal," SHAP values don't inherently possess this quality, which would require a sense of how likely a particular feature combination is.

SHAP values don't always provide human-friendly explanations. Kumar et al. (2020) demonstrated that SHAP values might not align with human expectations. Hence, it's not advisable to present them to end-users as straightforward information; they are complex concepts requiring explanations for proper usage and understanding.

16.5 SHAP values don't enable user actions

This limitation is closely related to the lack of contrastiveness. For instance, consider using SHAP values to explain a model that predicts corn yield based on multiple inputs, such as weather and fertilizer use. An explanation for a field with a low prediction might state that fertilizer use had a slightly positive impact. However, it doesn't answer the question of how to improve the field's yield. Should fertilizer use be increased? SHAP values don't provide a solution, as they only explain how the current value affected the prediction compared to the background data, without suggesting how to modify it. However, they do indicate which features need to be preserved to avoid regression to the mean.

For actionable recommendations, consider using counterfactual explanations[2]. Additionally, ensure the model itself offers actionable advice by using representative data samples and modeling causal relationships.

16.6 SHAP values can be misinterpreted

The SHAP value of a feature is not the difference in predicted value after removing the feature from model training. Instead, the SHAP value interpretation is as follows: 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 SHAP value. SHAP is not the ideal explanation method for sparse explanations (explanations with few features), as it always uses all the features.

16.7 SHAP values aren't a surrogate model

Unlike LIME, the SHAP value provides a single value per feature but no prediction model. Thus, it cannot be used to make statements about changes in predictions for changes in input, such as, "If I were to earn €300 more a year, my credit score would increase by 5 points."

16.8 Data access is necessary

Another limitation is that data access is required to calculate the SHAP value for a new data instance. Access to the prediction function alone is insufficient, as you need the data to replace parts of the instance of interest with values from randomly drawn instances. This can only be avoided if you can create data instances that resemble real data instances but are not actual instances from the training data.

[2] christophm.github.io/interpretable-ml-book/counterfactual.html

16.9 You can fool SHAP

It's possible to create intentionally misleading interpretations with SHAP, which can conceal biases (Slack et al. 2020), at least if you use the marginal sampling version of SHAP. If you are the data scientist creating the explanations, this is not an issue (it could even be advantageous if you are an unscrupulous data scientist who wants to create misleading explanations). However, for the recipients of a SHAP explanation, it is a disadvantage as they cannot be certain of the explanation's truthfulness.

16.10 Unrealistic data when features are correlated

See Correlation Chapter.

17 Building SHAP Dashboards with Shapash

By the end of this chapter, you will be able to:

- Create dashboards displaying insights from SHAP values.
- Install and use Shapash.

Shapash is a package that utilizes SHAP (or LIME) to compute contributions and visualize them in a dashboard or report. The dashboard is more convenient for exploring explanations than iterating through explanations in a Jupyter notebook or Python script, enabling exploratory data analysis.

17.1 Installation

You can install Shapash using pip:

```
pip install shapash==2.3.0
```

17.2 A quick example with Shapash

We'll build a simple dashboard using the California housing data.

```
import sklearn
import shap
import shapash
```

```
import pandas as pd
from sklearn.model_selection import train_test_split
from lightgbm import LGBMRegressor

# Load the California housing dataset from Shap
X, y = shap.datasets.california()

X_train, X_test, y_train, y_test = train_test_split(
  X, y, train_size=0.8, random_state=1
)

# Train a model
model = LGBMRegressor(n_estimators=100).fit(X_train, y_train)

y_pred = pd.DataFrame(
  model.predict(X_test), columns=['pred'], index=X_test.index
)

xpl = shapash.SmartExplainer(model=model)

explainer = shap.Explainer(model, X_train)

xpl.compile(y_pred=y_pred, x=X_test)
```

To start the app, simply run:

```
app = xpl.run_app(title_story='California Housing', port=8020)
```

This displays a link to `http://0.0.0.0:8020/`, where the Shapash app is accessible.

17.3 Dashboard overview

The interactive dashboard contains several sections:

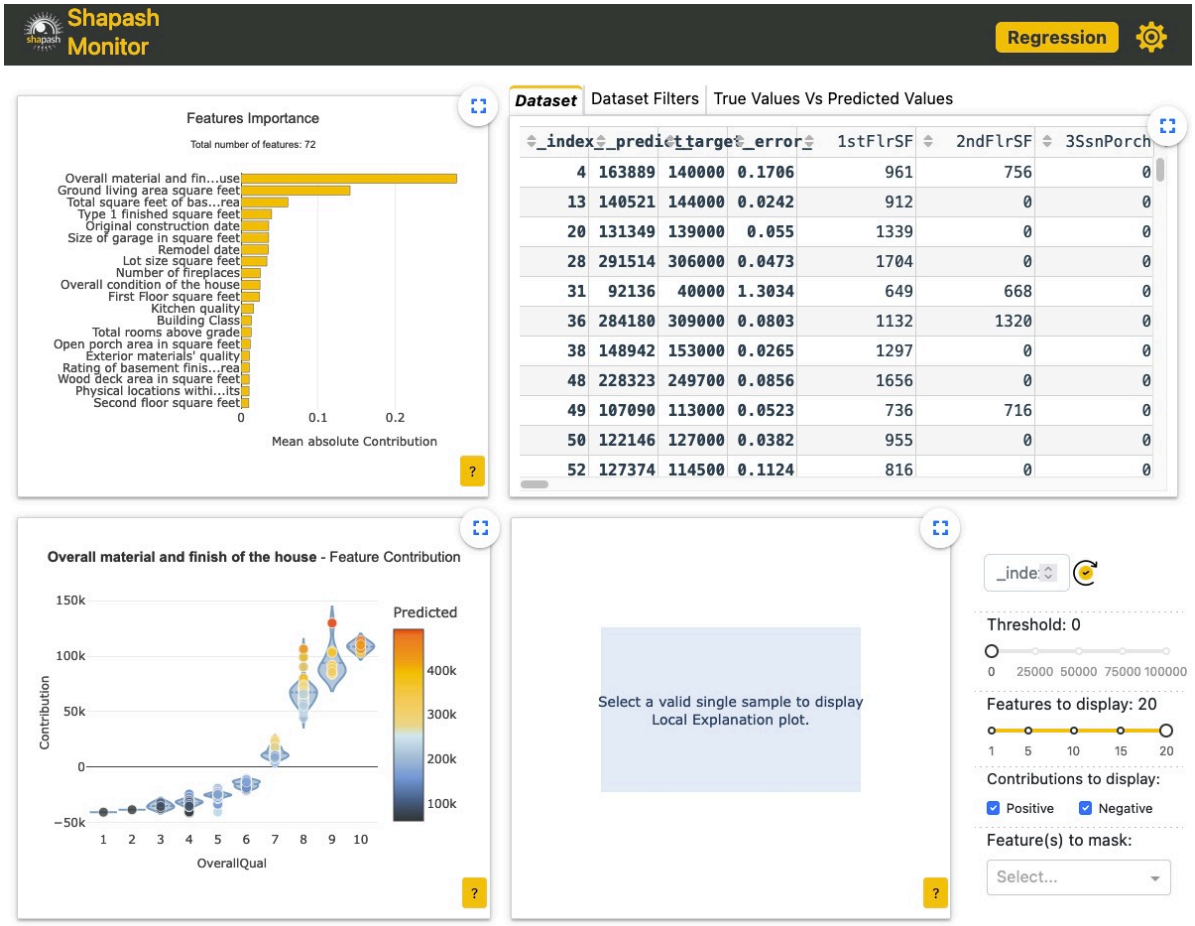


Figure 17.1: Shapash Webapp

- Top left: Displays feature importance.
- Bottom left: Feature dependence plots. The feature can be changed by clicking on the importance graph.
- Top right: Contains multiple tabs, including raw data values, filters for a subset view on the importance graph, and a true versus predicted plot.
- Bottom right: Presents a bar plot of SHAP values. Select an instance in the top-right graph using the dataset picker or in the bottom-left feature contribution figure. The graph can be customized, for example by displaying only the most influential features.

To stop the app, use:

```
app.kill()
```

18 Alternatives to the `shap` Library

Several alternative implementations of `shap` and SHAP values are available.

In Python:

- Captum[1](Kokhlikyan et al. 2020): A comprehensive model interpretability library, providing KernelShap, Sampling estimator, GradientShap, and Deep Shap implementations.
- shapley[2](Rozemberczki et al. 2022): Offers the exact estimator, several linear explanation methods, and Monte Carlo permutation sampling.

Python packages that utilize `shap` internally:

- DALEX[3](Baniecki et al. 2021)
- AIX360[4](Arya et al. 2019)
- InterpretML[5](Nori et al. 2019) encompasses multiple methods including SHAP.
- OmniXAI[6](Yang et al. 2022), a library dedicated to explainable AI.
- shapash[7], designed for dashboards and reports, as discussed in this chapter.

In R:

- DALEX[8](Biecek 2018)
- kernelshap[9]
- shapr[10](Sellereite and Jullum 2019)

[1] https://github.com/pytorch/captum
[2] https://github.com/benedekrozemberczki/shapley
[3] https://github.com/ModelOriented/DALEX
[4] https://github.com/Trusted-AI/AIX360
[5] https://github.com/interpretml/interpret/
[6] https://github.com/salesforce/OmniXAI
[7] https://github.com/MAIF/shapash
[8] https://github.com/ModelOriented/DALEX
[9] https://github.com/ModelOriented/kernelshap
[10] https://github.com/NorskRegnesentral/shapr

- ShapleyR[11]
- shapper[12], which depends on the Python shap package.
- shapviz[13] reproduces many SHAP plots from the original Python `shap` package and includes additional ones.
- treeshap[14]
- iml[15](Molnar et al. 2018).

[11] https://github.com/redichh/ShapleyR
[12] https://github.com/ModelOriented/shapper
[13] https://github.com/ModelOriented/shapviz
[14] https://github.com/ModelOriented/treeshap
[15] https://github.com/christophM/iml

19 Extensions of SHAP

This chapter outlines some extensions of SHAP values, with a focus on adaptations for specific models or data structures, while preserving their role in explaining predictions.

19.1 L-Shapley and C-Shapley for data with a graph structure

Although the SHAP values in the `shap` library are intended for i.i.d data, many structured data types consist of interconnected elements that form a graph. To accommodate this, Chen et al. (2018) introduced L-Shapley (local) and C-Shapley (connected) for data with a graph structure. These constrained versions of Shapley values are especially helpful in explaining image and text classifiers.

For text, L-Shapley only takes into account neighboring words, rather than all word interactions. For instance, when calculating the SHAP value of the 6th word in a text, only neighboring words are considered for word absence, based on the assumption that distant words have minimal interaction. C-Shapley, conversely, focuses on n-grams with $n \leq 4$, preventing coalitions between distant words.

19.2 Group SHAP for grouped features

Group SHAP (Lin and Gao 2022) is a simple method that groups features together and calculates a single SHAP value for each group instead of each feature value. This approach is less computationally taxing and provides a potential solution to the correlation issue. The `shap` library's Partition explainer facilitates this by allowing custom groupings.

19.3 n-Shapley values

The n-Shapley values (Bordt and Luxburg 2022) link SHAP values with GAMs (generalized additive models), with n signifying the interaction depth considered during the Shapley value calculation, depending on the GAM's training. A standard GAM includes no interaction (n=1), but interactions can be incorporated. The paper illustrates the relationship between n-Shapley values and functional decomposition, providing the nshap package in Python[1] for implementation. If the model being explained is a GAM, SHAP recovers all non-linear components, as outlined in the Additive Chapter.

19.4 Shapley Interaction Index

While SHAP values allocate predictions to individual players fairly, what happens with interactions between players? Intrinsically, these interactions are split among the participating players. However, measuring the interaction effect directly can be beneficial. Shapley Taylor (Sundararajan et al. 2020) and Faith-Shap (Tsai et al. 2023) are two examples of such interaction indices.

19.5 Causality and SHAP Values

Without additional assumptions, SHAP values don't represent causal attributions. Although they attribute features to the model prediction, this doesn't necessarily correspond to observable effects in reality. Additional assumptions are required to ensure causal effects.

Several extensions for causal interpretation have been proposed, including Causal Shapley Values (Heskes et al. 2020), Shapley Flow (Wang et al. 2021), and Asymmetric Shapley Values (Frye et al. 2020). These methods typically use directed acyclic graphs to identify (or assume) causal structures or apply the do-calculus. Occasionally, they modify or extend the original Shapley axioms, which can change the resulting Shapley game.

[1] https://github.com/tml-tuebingen/nshap

19.6 Counterfactual SHAP

As mentioned in the Limitations Chapter, SHAP values may not be the best choice for counterfactual explanations. A counterfactual explanation is a contrastive explanation that clarifies why the current prediction was made instead of a counterfactual outcome. This is crucial in recourse situations when someone affected by a decision wants to challenge a prediction, such as a creditworthiness classifier. Counterfactual SHAP (or CF-SHAP) (Albini et al. 2022) introduces this approach to SHAP values through careful selection of background data.

19.7 Explanation Shifts[2]

Explanation shifts (Mougan et al. 2022) refer to differences in how predictions are explained by machine learning models on training data compared to new data. By analyzing variations in SHAP values, we get insights into how the relationship between models and features evolves. Explanation shifts offer advantages over traditional methods that focus only on distribution shifts (like monitoring performance), as explanations provide a deeper understanding of changes in the model's behavior. Changes that might otherwise be overlooked. Explanation shifts is implemented within the skshift[3] Python package which also includes tutorials.

19.8 Fairness Measures via Explanations Distributions: Equal Treatment

Mougan et al. (2023) proposed a new fairness metric based on liberalism-oriented philosophical principles: "Equal Treatment". This measure considers the influence of feature values on predictions, defines distributions of SHAP explanations, and compares explanation distributions between populations with different protected characteristics. While related work focused on model predictions to measure demographic parity, the measure of equal treatment is more fine-grained,

[2]The preceding paragraph and the one about fairness were authored by Carlos Mougan, the researcher responsible for these extended uses of SHAP (and slightly edited by me).

[3]https://skshift.readthedocs.io/en/latest/

accounting for the usage of attributes by the model via explanation distributions. Consequently, equal treatment implies equal outcomes, but the converse is not necessarily true. Equal treatment is implemented within the explanationspace[4] package, which includes tutorials.

19.9 And many more

This summary is not comprehensive. At the time of writing, approximately 12k papers cited the original SHAP paper (Lundberg and Lee 2017b), with around 10k of these also mentioning "shap" or "shapley" in their title or abstract. Although many of these are review or application papers, a notable number extend SHAP values, making it unfeasible to cover all of them here.

[4] https://explanationspace.readthedocs.io/en/latest/Python

20 Other Uses of Shapley Values in Machine Learning

This chapter explores the various applications of Shapley values in tasks within the machine learning field beyond prediction explanations. While the Extensions Chapter focused on extensions for explaining predictions, this chapter introduces other tasks in machine learning and data science that can benefit from the use of Shapley values. The information in this chapter is based on the overview paper by Rozemberczki et al. (2022).

20.1 Feature importance determination based on loss function

SAGE (Covert et al. 2020) is a model-agnostic method for quantifying the predictive power of individual features while taking into account feature interactions. In this approach, model training is seen as the "game," individual features are the "players," and the overall performance is the "payout." SAGE differs from SHAP importance plots as it assesses features at a global level, instead of combining individual effects. The interpretation is based on the loss function, rather than the absolute prediction output. SAGE can be utilized to understand suboptimal models and identify corrupted or incorrectly encoded features. In a similar manner, Redell (2019) proposed a fair distribution of a model's R-squared using Shapley values.

20.2 Feature selection

Feature selection, a process closely related to SAGE, involves identifying the features that best enhance model performance. Shapley values can be incorporated into the modeling process, as suggested by Guyon and Elisseeff (2003) and Fryer et al. (2021). By repeatedly training the model using different features, it is possible to estimate each feature's importance to performance. However, Fryer et al. (2021) argue that Shapley values may not be ideally suited for this task, as they excel in attribution rather than selection. Despite this limitation, each feature is treated as a player, with the model performance being the payoff. Unlike SHAP values for prediction, the contribution of each feature is evaluated globally for the entire model. The ultimate goal is to ascertain each feature's contribution to the model's performance.

20.3 Data valuation

Data valuation refers to the process of evaluating the importance of each input data in a machine learning model. Shapley values can be used to address this problem, as illustrated by "Data Shapley"(Ghorbani and Zou 2019). In this approach, data points in the training set are considered players, and the payoff is determined by evaluation metrics or the model's goodness of fit on the test data. This method is similar to deletion diagnostics such as Cook's Distance. An application of Data Shapley is presented in a paper by Bloch et al.(Bloch et al. 2021), where it was used to select patient data for an Alzheimer detection model.

20.4 Model valuation in ensembles

In an ensemble of multiple models, the overall performance of the ensemble ideally exceeds that of each individual model. Each model is considered a player, and the payout is determined by the ensemble's total performance.

20.5 Federated learning

Federated learning is a method to train a machine learning model on private data distributed across multiple entities, such as hospitals. In this context, each hospital is a player that contributes to the training of the model. Federated learning facilitates training models across various private datasets while maintaining privacy. The payout is determined by the model's goodness of fit or other performance metrics, which estimate the contribution of each entity.

20.6 And many more

This list is far from exhaustive, and it is possible to expand it even further. Shapley values are a versatile method. Whenever you need to distribute a numerical outcome among several entities, consider using Shapley values, whether in machine learning, data science, or other fields.

21 Acknowledgments

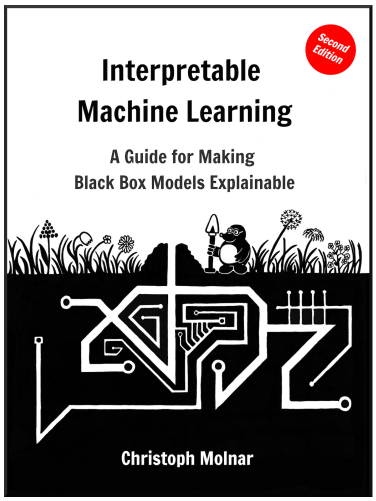
This book stands on the shoulders of giants. In particular, these giants are the researchers working on Shapley values and SHAP, with a special mention of Scott Lundberg who played a pivotal role in bringing Shapley values to machine learning.

I would also like to thank all my beta readers for their invaluable feedback. In no particular order: Carlos Mougan, Bharat Raghunathan, Junaid Butt, Joshua Le Cornu, Vaibhav Krishna Irugu Guruswamy, Liban Mohamed, Tim Triche, Ronald Richman, Germán García, Jeff Herman, Zachary Duey, Sven Kruschel, Joaquín Bogado, Shino Chen, Sairam Subramanian, Kerry Pearn, Gavin Parnaby, Robert Martin, Andrea Ruggerini, Arved Niklas Fanta, Saman Parvaneh, Simon Prince, Marouane Il Idrissi, Kranthi Kamsanpalli, Enrico Roletto, David Cortés, Valentino Zocca, HaveF, and Johannes Widera.

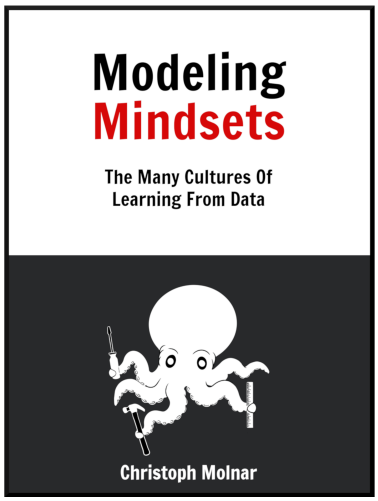
And of course a big thanks goes to Heidi, my wife, who always has to put up with my ramblings when I learn something new and just have to tell someone. The cover art (the team of meerkats) was created by jeeshiu from Fiverr[1].

[1] https://www.fiverr.com/jeeshiu

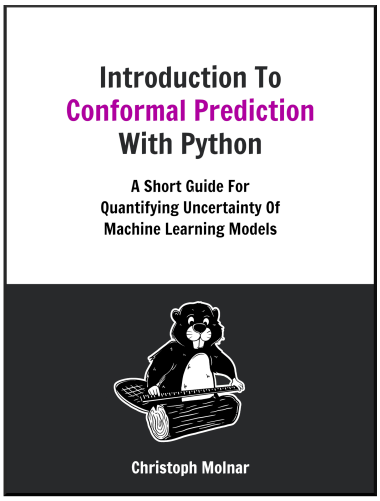
More From The Author



Interpretable Machine Learning covers a wide range of explainable AI techniques, from counterfactuals and PDPs to Permutation Feature Importance and SHAP.



In less than 100 pages, **Modeling Mindsets** elucidates the worldviews behind various statistical modeling and machine learning mindsets.



Introduction To Conformal Prediction With Python is the quickest way to learn an easy-to-use and very general technique for uncertainty quantification.

Available from leanpub.com and amazon.com.

References

Aas K, Jullum M, Løland A (2021) Explaining individual predictions when features are dependent: More accurate approximations to shapley values. Artificial Intelligence 298:103502

Albini E, Long J, Dervovic D, Magazzeni D (2022) Counterfactual shapley additive explanations. In: 2022 ACM conference on fairness, accountability, and transparency. pp 1054–1070

Arya V, Bellamy RKE, Chen P-Y, et al (2019) One explanation does not fit all: A toolkit and taxonomy of AI explainability techniques[2]

Bach S, Binder A, Montavon G, et al (2015) On pixel-wise explanations for non-linear classifier decisions by layer-wise relevance propagation. PloS one 10:e0130140

Baniecki H, Kretowicz W, Piatyszek P, et al (2021) Dalex: Responsible machine learning with interactive explainability and fairness in python[3]. Journal of Machine Learning Research 22:1–7

Biecek P (2018) DALEX: Explainers for complex predictive models in r[4]. Journal of Machine Learning Research 19:1–5

Bloch L, Friedrich CM, Initiative ADN (2021) Data analysis with shapley values for automatic subject selection in alzheimer's disease data sets using interpretable machine learning. Alzheimer's Research & Therapy 13:1–30

Bordt S, Luxburg U von (2022) From shapley values to generalized additive models and back. arXiv preprint arXiv:220904012

Caruana R, Lou Y, Gehrke J, et al (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. pp 1721–1730

Chen H, Janizek JD, Lundberg S, Lee S-I (2020) True to the model or true to the data? arXiv preprint arXiv:200616234

[2] https://arxiv.org/abs/1909.03012
[3] http://jmlr.org/papers/v22/20-1473.html
[4] http://jmlr.org/papers/v19/18-416.html

Chen H, Lundberg S, Lee S-I (2021) Explaining models by propagating shapley values of local components. Explainable AI in Healthcare and Medicine: Building a Culture of Transparency and Accountability 261–270

Chen J, Song L, Wainwright MJ, Jordan MI (2018) L-shapley and c-shapley: Efficient model interpretation for structured data. arXiv preprint arXiv:180802610

Covert I, Lundberg SM, Lee S-I (2020) Understanding global feature contributions with additive importance measures. Advances in Neural Information Processing Systems 33:17212–17223

Deng J, Dong W, Socher R, et al (2009) Imagenet: A large-scale hierarchical image database. In: 2009 IEEE conference on computer vision and pattern recognition. Ieee, pp 248–255

Elish MC, Watkins EA (2020) Repairing innovation: A study of integrating AI in clinical care. Data & Society

Frye C, Rowat C, Feige I (2020) Asymmetric shapley values: Incorporating causal knowledge into model-agnostic explainability. Advances in Neural Information Processing Systems 33:1229–1239

Fryer D, Strümke I, Nguyen H (2021) Shapley values for feature selection: The good, the bad, and the axioms. IEEE Access 9:144352–144360

García MV, Aznarte JL (2020) Shapley additive explanations for NO2 forecasting. Ecological Informatics 56:101039

Ghorbani A, Zou J (2019) Data shapley: Equitable valuation of data for machine learning. In: International conference on machine learning. PMLR, pp 2242–2251

Grinsztajn L, Oyallon E, Varoquaux G (2022) Why do tree-based models still outperform deep learning on tabular data? arXiv preprint arXiv:220708815

Guyon I, Elisseeff A (2003) An introduction to variable and feature selection. Journal of machine learning research 3:1157–1182

He K, Zhang X, Ren S, Sun J (2016) Deep residual learning for image recognition. In: Proceedings of the IEEE conference on computer vision and pattern recognition. pp 770–778

Heskes T, Sijben E, Bucur IG, Claassen T (2020) Causal shapley values: Exploiting causal knowledge to explain individual predictions of complex models. Advances in neural information processing systems 33:4778–4789

Jabeur SB, Mefteh-Wali S, Viviani J-L (2021) Forecasting gold price with the XGBoost algorithm and SHAP interaction values. Annals of Operations Research 1–21

Janzing D, Minorics L, Blöbaum P (2020) Feature relevance quantification in

explainable AI: A causal problem. In: International conference on artificial intelligence and statistics. PMLR, pp 2907–2916
Johnsen PV, Riemer-Sørensen S, DeWan AT, et al (2021) A new method for exploring gene–gene and gene–environment interactions in GWAS with tree ensemble methods and SHAP values. BMC bioinformatics 22:1–29
Kim Y, Kim Y (2022) Explainable heat-related mortality with random forest and SHapley additive exPlanations (SHAP) models. Sustainable Cities and Society 79:103677
Kokhlikyan N, Miglani V, Martin M, et al (2020) Captum: A unified and generic model interpretability library for PyTorch[5]
Kumar IE, Venkatasubramanian S, Scheidegger C, Friedler S (2020) Problems with shapley-value-based explanations as feature importance measures. In: International conference on machine learning. PMLR, pp 5491–5500
Lin K, Gao Y (2022) Model interpretability of financial fraud detection by group SHAP. Expert Systems with Applications 210:118354
Lundberg SM, Erion G, Chen H, et al (2020) From local explanations to global understanding with explainable AI for trees. Nature machine intelligence 2:56–67
Lundberg SM, Lee S-I (2017b) A unified approach to interpreting model predictions. Advances in neural information processing systems 30:
Lundberg SM, Lee S-I (2017a) A unified approach to interpreting model predictions[6]. In: Guyon I, Luxburg UV, Bengio S, et al. (eds) Advances in neural information processing systems 30. Curran Associates, Inc., pp 4765–4774
Miller T (2019) Explanation in artificial intelligence: Insights from the social sciences. Artificial intelligence 267:1–38
Mitchell R, Cooper J, Frank E, Holmes G (2022) Sampling permutations for shapley value estimation
Molnar C (2022) Interpretable machine learning: A guide for making black box models explainable[7], 2nd edn.
Molnar C, Casalicchio G, Bischl B (2018) Iml: An r package for interpretable machine learning. Journal of Open Source Software 3:786
Mougan C, Broelemann K, Kasneci G, et al (2022) Explanation shift: Detecting distribution shifts on tabular data via the explanation space. arXiv preprint arXiv:221012369
Mougan C, State L, Ferrara A, et al (2023) Demographic parity inspector: Fair-

[5] https://arxiv.org/abs/2009.07896
[6] http://papers.nips.cc/paper/7062-a-unified-approach-to-interpreting-model-predictions.pdf
[7] https://christophm.github.io/interpretable-ml-book

ness audits via the explanation space. arXiv preprint arXiv:230308040
Nori H, Jenkins S, Koch P, Caruana R (2019) InterpretML: A unified framework for machine learning interpretability. arXiv preprint arXiv:190909223
Parsa AB, Movahedi A, Taghipour H, et al (2020) Toward safer highways, application of XGBoost and SHAP for real-time accident detection and feature analysis. Accident Analysis & Prevention 136:105405
Redell N (2019) Shapley decomposition of r-squared in machine learning models. arXiv preprint arXiv:190809718
Ribeiro MT, Singh S, Guestrin C (2016) " why should i trust you?" explaining the predictions of any classifier. In: Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining. pp 1135–1144
Rodríguez-Pérez R, Bajorath J (2020) Interpretation of machine learning models using shapley values: Application to compound potency and multi-target activity predictions. Journal of computer-aided molecular design 34:1013–1026
Rozemberczki B, Watson L, Bayer P, et al (2022) The shapley value in machine learning. arXiv preprint arXiv:220205594
Rudin C (2019) Stop explaining black box machine learning models for high stakes decisions and use interpretable models instead. Nature machine intelligence 1:206–215
Scholbeck CA, Molnar C, Heumann C, et al (2020) Sampling, intervention, prediction, aggregation: A generalized framework for model-agnostic interpretations. In: Machine learning and knowledge discovery in databases: International workshops of ECML PKDD 2019, würzburg, germany, september 16–20, 2019, proceedings, part i. Springer, pp 205–216
Sellereite N, Jullum M (2019) Shapr: An r-package for explaining machine learning models with dependence-aware shapley values. Journal of Open Source Software 5:2027. https://doi.org/10.21105/joss.02027
Shapley LS et al (1953) A value for n-person games
Shrikumar A, Greenside P, Kundaje A (2017) Learning important features through propagating activation differences. In: International conference on machine learning. PMLR, pp 3145–3153
Slack D, Hilgard S, Jia E, et al (2020) Fooling lime and shap: Adversarial attacks on post hoc explanation methods. In: Proceedings of the AAAI/ACM conference on AI, ethics, and society. pp 180–186
Smith M, Alvarez F (2021) Identifying mortality factors from machine learning using shapley values–a case of COVID19. Expert Systems with Applications

176:114832
Staniak M, Biecek P (2018) Explanations of model predictions with live and breakDown packages. arXiv preprint arXiv:180401955
Štrumbelj E, Kononenko I (2010) An efficient explanation of individual classifications using game theory. The Journal of Machine Learning Research 11:1–18
Štrumbelj E, Kononenko I (2014) Explaining prediction models and individual predictions with feature contributions. Knowledge and information systems 41:647–665
Sundararajan M, Dhamdhere K, Agarwal A (2020) The shapley taylor interaction index. In: International conference on machine learning. PMLR, pp 9259–9268
Sundararajan M, Najmi A (2020) The many shapley values for model explanation. In: International conference on machine learning. PMLR, pp 9269–9278
Sundararajan M, Taly A, Yan Q (2017) Axiomatic attribution for deep networks. In: International conference on machine learning. PMLR, pp 3319–3328
Tsai C-P, Yeh C-K, Ravikumar P (2023) Faith-shap: The faithful shapley interaction index. Journal of Machine Learning Research 24:1–42
Wang D, Thunéll S, Lindberg U, et al (2022) Towards better process management in wastewater treatment plants: Process analytics based on SHAP values for tree-based machine learning methods. Journal of Environmental Management 301:113941
Wang J, Wiens J, Lundberg S (2021) Shapley flow: A graph-based approach to interpreting model predictions. In: International conference on artificial intelligence and statistics. PMLR, pp 721–729
Yang W, Le H, Savarese S, Hoi S (2022) OmniXAI: A library for explainable AI. https://doi.org/10.48550/ARXIV.2206.01612

A SHAP Estimators

This chapter presents the various SHAP estimators in detail.

A.1 Exact Estimation: Computing all the coalitions

> **Exact Estimation**
>
> This method computes the exact SHAP value. It's model-agnostic and is only meaningful for low-dimensional tabular data (<15 features).

The exact estimation theoretically computes all 2^p possible coalitions, from which we can calculate all possible feature contributions for each feature, as discussed in the Theory Chapter. It also uses all of the background data, not just a sample. This means the computation has no elements of randomness – except that the data sample is random. Despite the high computational cost, which depends on the number of features and the size of the background data, this method uses all available information and provides the most accurate estimation of SHAP values compared to other model-agnostic estimation methods.

Here is how to use the exact method with the `shap` package:

```
explainer = shap.Explainer(model, background)
```

However, `shap` limits the marginal contributions of features to coalitions of size 0, 1, p-2, and p-1, which means it only covers interactions of a maximum size of 2. As the documentation states[1], it should be used for less than 15 features.

[1] https://github.com/slundberg/shap/blob/master/shap/explainers/_exact.py

Because of this enumeration, the exact estimation uses an optimization method called Gray code. Gray code is an effective ordering of coalitions where adjacent coalitions only differ in one feature value, which can be directly used to compute marginal contributions. This method is more efficient than enumerating all possible coalitions and adding features to them, as Gray code reduces the number of model calls through more effective computation.

Exact SHAP values are often not feasible, but this issue can be addressed through sampling.

A.2 Sampling Estimator: Sampling the coalitions

> **i** Sampling Estimator
>
> This method works by sampling coalitions. It's model-agnostic.

The first versions of the Sampling Estimator were proposed by Štrumbelj and Kononenko (2014) and modified by Štrumbelj and Kononenko (2010). The sampling process involves two dimensions: sampling from the background data and sampling the coalitions.

To calculate the exact SHAP value, all possible coalitions (sets) of feature values must be evaluated with and without the j-th feature. However, the exact solution becomes problematic as the number of features increases due to the exponential increase in the number of possible coalitions. Štrumbelj and Kononenko (2014) proposed an approximation using Monte Carlo integration:

$$\hat{\phi}_j = \frac{1}{M} \sum_{m=1}^{M} \left(f(x^{(i)}_{S \cup j} \cup x^{(m)}_{C \setminus j}) - f(x^{(i)}_S \cup x^{(m)}_C) \right)$$

Implicitly, this averaging operation weights samples according to the probability distribution of X. Repeat this process for each feature to obtain all SHAP values.

In `shap`, the estimation method is implemented as follows:

```
explainer = shap.explainers.Sampling(model, background)
```

- The default number of samples (`nsamples`) is `auto`, which equates to 1,000 times the number of features.
- The Sampling Estimator exclusively accepts the identity link (relevant for classifiers). The Sampling Estimator, while more effective than the exact estimation for larger datasets, is not the most efficient method for estimating SHAP values. Let's now discuss the Permutation Estimator.

A.3 Permutation Estimator: Sampling permutations

> **i** Permutation Estimator
>
> This method samples permutations of feature values and iterates through them in both directions. It is model-agnostic.

The Permutation Estimator distinguishes itself from the Sampling Estimator by sampling entire permutations of features, rather than random coalitions of features. These permutations are then traversed in both directions.

The Permutation Estimator is explained in detail in the Estimation Chapter.

A.3.1 Interlude: Why one permutation suffices for 2-way interactions

Here's some intuition as to why one permutation is sufficient, or rather, an example (feel free to skip this if you're already convinced that one permutation suffices for detecting 2-way interactions).

Example:

- The prediction function is $f(x) = 2x_3 + 3x_1x_3$, which includes an interaction between features x_1 and x_3.
- We'll explain the data point $(x_1 = 4, x_2 = 1, x_3 = 1, x_4 = 2)$.

- Our background data consists of only one data point for simplicity: $(x_1 = 0, x_2 = 0, x_3 = 0, x_4 = 0)$.
- We'll examine two permutations and demonstrate that both result in the same marginal contributions for feature x_3.
- Note, this doesn't prove that all 2-way interactions are recoverable by permutation, but it provides some insight into why it might work.
- The first permutation: (x_2, x_3, x_1, x_4).
  - We have two marginal contributions: x_3 to $\{x_2\}$ and x_3 to $\{x_1, x_4\}$.
  - We denote $f_{2,3}$ as the prediction where x_2 and x_3 values come from the data point to be explained, and x_1 and x_4 values come from the background data.
  - Thus: $f_{2,3} = f(x_1 = 0, x_2 = 1, x_3 = 1, x_4 = 0) = 2 \cdot 1 + 3 \cdot 0 \cdot 1 = 2$
- The marginal contributions are $f_{2,3} - f_2 = 2 - 0 = 2$ and $f_{1,3,4} - f_{1,4} = 14$.
- Now let's consider a different permutation: (x_1, x_2, x_3, x_4).
  - This is the original feature order, but it is also a valid permutation.
  - For this, we'll compute different marginal contributions.
  - $f_{1,2,3} - f_{1,2} = 14$.
  - $f_{3,4} - f_4 = 2$.
  - And, unsurprisingly, these are the same marginal contributions as for the other permutation.
- So, even with only a 2-way interaction, we had two different permutations that we iterated forward and backward, and we obtained the same marginal contributions.
- This suggests that adding more permutations doesn't provide new information for the feature of interest.
- This isn't a proof, but it gives an idea of why this method works.

This type of sampling is also known as antithetic sampling and performs well compared to other sampling-based estimators of SHAP values (Mitchell et al. 2022).

End of interlude

Here's how to use the Permutation Estimator in `shap`:

```
explainer = shap.explainers.Permutation(model, background)
```

- The Permutation method is the default estimation method for model-agnostic explanations. If you set `algorithm=auto` when creating an `Explainer` (which is the default) and there's no model-specific SHAP estimator for your model, the permutation method will be used.
- The current SHAP implementation iterates 10 default permutations forward and backward.
- This implementation also supports hierarchical data structures with partition trees, which aren't implemented in the Kernel Estimator (see later) or the Sampling Estimator.
- Since the permutations are sampled, it's recommended to set a seed in the `Explainer` for reproducibility.

A.4 Linear Estimator For linear models

We'll now move from model-agnostic estimation methods to model-specific ones. These estimators take advantage of the model's internal structure. For the Linear Estimator, we use the linear equation common to linear regression models. Linear regression models can be expressed in the following form:

$$f(x) = \beta_0 + \beta_1 x_1 + ... + \beta_p x_p,$$

The β's represent the weights or coefficients by which the features are multiplied to generate the prediction. The intercept β_0 is a unique coefficient that determines the output when all feature values are zero, implying that there are no interactions[2] and no non-linear relations. The SHAP values are simple to compute in this case, as discussed in the Linear Chapter. They are defined as:

$$\phi_j^{(i)} = \beta_j \cdot (x_j - \mathbb{E}(X_j))$$

This formula also applies if you have a non-linear link function. It means that the model isn't entirely linear, as the weighted sum is transformed before making

[2]While you can generally add interactions to a linear model, this option is not available for the Linear Estimator.

the prediction. This model class is known as generalized linear models (GLMs). Logistic regression is an example of a GLM, and it's defined as:

$$f(x) = \frac{1}{1 + exp(-(\beta_0 + \beta_1 x_1 + ... + \beta_p x_p))}$$

GLMs have the following general form:

$$f(x) = g(\beta_0 + \beta_1 x_1 + ... + \beta_p x_p)$$

Even though the function is fundamentally linear, the result of the weighted sum is non-linearly transformed. In this case, SHAP can still use the coefficients, and the Linear Estimator remains applicable. However, it operates not on the level of the prediction but on the level of the inverse of the function g, namely g^{-1}. For logistic regression, it means that we interpret the results at the level of log odds. Remember that this adds some complexity to the interpretation.

Here are some notes on implementation:

```
shap.explainers.Linear(model, background)
```

- To use the link function, set `link` in the explainer. The default is the identity link. Learn more in the Classification Chapter.
- The SHAP implementation allows for accounting for feature correlations when `feature_perturbation` is set to "correlation_dependent". However, this will result in a different "game" and thus different SHAP values. Read more in the Correlation Chapter.

A.5 Additive Estimator: For additive models

> **Note**
>
> This model-specific estimator takes advantage of the lack of feature interaction in additive models.

The Additive Estimator is a generalization of the Linear Estimator. While it still assumes no interactions between features, it allows the effect of a feature to be non-linear. This model class is represented as follows:

$$f(x) = \beta_0 + f_1(x_1) + ... + f_p(x_p)$$

This is also known as a generalized additive model (GAM). Each f_j is a potentially non-linear function of a feature. Examples include splines to model a smooth, non-linear relationship between a feature and the target. However, we can still use the fact that there are no interactions between the features. In other words, the Additive Estimator assumes that the model only has first-order effects.

How does the absence of interaction (additivity) assist with the computation of SHAP values? It helps significantly. When considering marginal contributions and coalitions, an absence of interaction means that the effect of a feature value on the prediction is independent of the features already in the coalition. The marginal contribution of a feature is always constant. Thus, to compute the SHAP values of a feature, it's enough to add it to just one coalition, for example, the coalition where all other features are absent. This means we require exactly $(p + 1) \cdot n_{bg}$ calls to the model, where n_{bg} represents the size of the background data, or specifically, the amount we sample from it. The SHAP value for feature X_j can be computed as follows:

$$\phi_j^{(i)} = f_j(x_j^{(i)}) - \frac{1}{n}\sum_{k=1}^{n} f_j(x_j^{(k)})$$

This equation is similar to the one used in the Linear Estimator. The first term denotes the effect of the feature value $x_j^{(i)}$, while the second term centers it at the expected effect of the feature X_j. However, different assumptions about the shape of the effect are required due to the use of different models (linear versus additive).

Like the Linear Estimator, the Additive Estimator can also be extended to non-linear link functions:

$$f(x) = g(\beta_0 + f_1(x_1) + ... + f_p(x_p))$$

When a link function is used, interpretation happens at the level of the linear predictor, which isn't on the scale of the prediction but on the inverse of the link function g^{-1}.

The implementation details are as follows:

```
shap.explainers.Additive
```

- Only compatible with the Tabular Masker.
- Can be used in conjunction with clustered features.

A.6 Kernel Estimator: The deprecated original

The Kernel SHAP method estimates the SHAP values by sampling coalitions and conducting a weighted linear regression.

> **Kernel Estimator**
>
> The Kernel Estimator is no longer widely used in SHAP, although it's still available. Instead, the Permutation Estimator is now the preferred option. This section remains for historical reasons. The Kernel Estimator was the original SHAP implementation, proposed in Lundberg and Lee (2017b), and it drew parallels with other attribution methods such as LIME (Ribeiro et al. 2016) and DeepLIFT (Shrikumar et al. 2017).

The Kernel Estimator comprises five steps:

- Sample coalitions $z'_k \in \{0,1\}^M, \quad k \in \{1,\dots,K\}$ (1 = feature present in coalition, 0 = feature absent).
- Calculate a prediction for each z'_k by converting z'_k to the original feature space and then applying model $f : f(h_x(z'_k))$.
- Determine the weight for each z'_k using the SHAP kernel.
- Fit a weighted linear model.
- Return SHAP values $\phi_j^{(i)}$, which are the coefficients from the linear model.

A random coalition is created by repeatedly flipping a coin until we have a sequence of 0's and 1's. For instance, the vector (1,0,1,0) denotes 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. ("Wait a minute!" you might say, "The model hasn't been trained on these binary coalition data and thus can't make predictions for them.") To convert coalitions of feature values into valid data instances, we need a function $h_x(z') = z$ where $h_x : \{0,1\}^M \rightarrow \mathbb{R}^p$. The function h_x maps 1's to the corresponding value from the instance x that we wish to explain and 0's to values from the original instance.

For SHAP-compliant weighting, Lundberg et al. propose the SHAP kernel:

$$\pi_x(z') = \frac{(M-1)}{\binom{M}{|z'|}|z'|(M-|z'|)}$$

Here, M represents the maximum coalition size, and $|z'|$ signifies the number of features present in instance z'. Lundberg and Lee illustrate that using this kernel weight for linear regression yields SHAP values. LIME, a method that functions by fitting local surrogate models, operates similarly to Kernel SHAP. If you were to employ the SHAP kernel with LIME on the coalition data, LIME would also generate SHAP values!

A.6.1 More strategic sampling of coalitions

We can adopt a more strategic approach to sampling coalitions: The smallest and largest coalitions contribute most to the weight. By allocating a portion of the sampling budget K to include these high-weight coalitions, as opposed to random sampling, we can achieve better SHAP value estimates. We start with all possible coalitions containing 1 and M-1 features, resulting in a total of 2M coalitions. If the budget permits (current budget is K - 2M), we can include coalitions featuring 2 and M-2 elements, and so on. For the remaining coalition sizes, we sample using recalibrated weights.

A.6.2 Fitting a weighted linear regression model

We possess the data, the target, and the weights; everything necessary to construct our weighted linear regression model:

$$g(z') = \phi_0 + \sum_{j=1}^{M} \phi_j^{(i)} z'_j$$

We train the linear model g by optimizing the following loss function L:

$$L(f, g, \pi_x) = \sum_{z' \in Z} (f(h_x(z')) - g(z'))^2 \pi_x(z')$$

Here, Z is the training data. This equation represents the familiar sum of squared errors that we typically optimize for linear models. The estimated coefficients of the model, the $\phi_j^{(i)}$'s, are the SHAP values.

As we are in a linear regression context, we can rely on standard tools for regression. For instance, we can incorporate regularization terms to make the model sparse. By adding an L1 penalty to the loss L, we can create sparse explanations.

Implementation details in `shap`:

- Regularization is employed, which helps reduce variance and noise, but introduces bias to the SHAP value estimates.
- The Kernel Estimator has been largely superseded by the Permutation Estimator.

A.7 Tree Estimator: Designed for tree-based models

> **Tree Estimator**
>
> This estimation method is specific to tree-based models such as decision trees, random forests, and gradient boosted trees.

The Tree Estimator contributes significantly to the popularity of SHAP. While other estimation methods may be relatively slow, the Tree Estimator is a fast, model-specific estimator designed for tree-based models. It is compatible with decision trees, random forests, and gradient boosted trees such as LightGBM and

XGBoost. Boosted trees are particularly effective for tabular data, and having a quick method to compute SHAP values positions the technique advantageously. Moreover, it is an exact method, meaning you obtain the correct SHAP values rather than mere estimates, at least with respect to the coalitions. It remains an estimate in relation to the background data, given that the background data itself is a sample.

The Tree Estimator makes use of the tree structure to compute SHAP values, and it comes in two versions: Interventional and Tree-Path Dependent Estimator.

A.7.1 Interventional Tree Estimator

The Interventional Estimator calculates the usual SHAP values but takes advantage of the tree structure for the computation. The estimation is performed by recursively traversing the tree(s). Here is a basic outline of the algorithm for explaining one data point and one background data point. Bear in mind that this explanation applies to a single tree; the strategy for an ensemble will be discussed subsequently.

- We start with a background data point, which we'll refer to as z, and the data point we want to explain, known as x.
- The process begins at the top of the tree, tracing the paths for x and z.
- However, we don't just trace the paths of x and z as they would merely terminate at two or possibly the same leaf nodes.
- Instead, at each crossroad, we ask: what if the decision was based on the feature values of x and z?
- If they differ, both paths are pursued.
- This combined path tracing is done recursively.
- Ah, recursion — always a mind boggler!
- Upon reaching the terminal nodes, or leaves, predictions from these leaf nodes are gathered and weighted based on how many feature differences exist in relation to x and z.
- These weights are recursively combined.

Here's a simpler explanation: We form a coalition of feature values, which includes present players (feature values from x) and absent players (feature values from z). Despite there being 2^p coalitions, the tree only allows a limited number of possible predictions. Instead of starting with all coalitions, we can reverse the

process and explore the tree paths to determine which coalitions would yield different predictions since many feature changes may have no impact on the prediction. The tricky part, which is addressed in SHAP, is accurately weighting and combining predictions based on the altered features. For further details, refer to p.25, Algorithm 3 of this paper[3] by Lundberg et al. (2020).

For ensembles of trees, we can average the SHAP values, with each tree's prediction contribution influencing the final ensemble's weight. Thanks to the additivity of SHAP values, the Shapley values of a tree ensemble are the (weighted) average of the individual trees' Shapley values.

The complexity of the Tree Estimator (over a background set of size n_{bg}) is $\mathcal{O}(Tn_{bg}L)$, where T is the number of trees and L is the maximum number of leaves in any tree.

A.8 Tree-path-dependent Estimator

There's an alternative method for calculating SHAP values for trees. This method is mainly of historical interest as it's not the default approach and has certain issues. This second method depends on the tree paths and doesn't require background data. While the first Tree SHAP method treats features X_j and X_{-j} as either independent or combined as if they were, this alternative method leans more towards the conditional expectation $E_{X_j|X_{-j}}(f(x)|x_j)$. Still, it's not precisely that (Aas et al. 2021), so its exact nature is somewhat ambiguous. Since the features are altered in a conditional rather than marginal manner, the resulting SHAP values vary from those obtained using the Interventional Tree Estimator method. Although these are still valid SHAP values, changing the conditioning represents a different value function and subsequently a different game being played and attributed to the feature values. One problem with conditional expectation is that features with no effect on the prediction function f may still receive a non-zero TreeSHAP estimate (Janzing et al. 2020; Sundararajan and Najmi 2020). This can happen when the feature is correlated with another feature that plays a significant role in the prediction. I'll provide some insight into how the Tree-Path Dependent Estimator computes the expected prediction for a single tree, an instance x, and a feature subset S. If we condition on all features, which

[3] https://arxiv.org/abs/1905.04610

means that S contains all features, the prediction from the node where instance x falls would be the expected prediction. Conversely, if we don't condition the prediction on any feature, meaning S is empty, we use the weighted average of predictions from all terminal nodes. If S includes some, but not all features, we disregard predictions of unreachable nodes. A node is deemed unreachable if the decision path to it contradicts values in x_S. From the remaining terminal nodes, we average the predictions weighted by node sizes, which refers to the number of training samples in each node. The mean of the remaining terminal nodes, weighted by the number of instances per node, yields the expected prediction for x given S. The challenge lies in applying this procedure for each possible subset S of the feature values.

The fundamental concept of the path-dependent Tree Estimator is to push all possible subsets S down the tree simultaneously. For each decision node, we need to keep track of the number of subsets. This depends on the subsets in the parent node and the split feature. For instance, when the first split in a tree is on feature x3, all subsets containing feature x3 will go to one node (the one where x goes). Subsets that do not include feature x3 go to both nodes with reduced weight. Unfortunately, subsets of different sizes carry different weights. The algorithm must keep track of the cumulative weight of the subsets in each node, which complicates the algorithm.

The tree estimation method is implemented in `shap`:

```
shap.explainers.Tree(
  model, data, feature_perturbation='interventional'
)
shap.explainers.Tree(
  model, feature_perturbation='tree_path_dependent'
)
```

- The implementation is in C++ for enhanced speed.
- It supports tree-based models like XGBoost, LightGBM, CatBoost, PySpark, and most tree-based models found in scikit-learn, such as RandomForestRegressor.
- There are two feature_perturbation methods available: `interventional` and `tree_path_dependent`.
- The `tree_path_dependent` option requires no background data.

A.9 Gradient Estimator: For gradient-based models

> **Gradient Estimator**
>
> The Gradient Estimator is a model-specific estimation method tailored for gradient-based models, such as neural networks, and can be applied to both tabular and image data.

Many models, including several neural networks, are gradient-based. This means that we can compute the gradient of the loss function with respect to the model input. When we can compute the gradient with respect to the input, we can use this information to calculate SHAP values more efficiently.

Gradient SHAP is defined as the expected value of the gradients times the inputs minus the baselines.

$$\text{GradientShap}(x) = \mathbb{E}\left((x_j - \tilde{x}_j) \cdot \frac{\delta g(\tilde{x} + \alpha \cdot (x - \tilde{x}))}{\delta x_j} d\alpha\right)$$

It's estimated with:

$$\text{GradientShap}(x) = \frac{1}{n_{bg}} \sum_{i=1}^{n_{bg}} (x_j - \tilde{x}_j^{(i)}) \cdot \frac{\delta g(\tilde{x} + \alpha_i \cdot (x - \tilde{x}^{(i)}))}{\delta x_j} d\alpha$$

So, what does this formula do? For a given feature value x_j, this estimation method cycles through the background data of size n_{bg}, computing two terms:

- The distance between the data point to be explained x_j and the sample from the background data.
- The gradient g of the prediction with respect to the j-th feature, calculated not at the position of the point to be explained, but at a random location of feature X_j between the data point of interest and the background data. The α_i is uniformly sampled from $[0, 1]$.

These terms are multiplied and averaged over the background data to approximate SHAP values. There's a connection between the Gradient Estimator and a

method called Integrated Gradients (Sundararajan et al. 2017). Integrated Gradients is a feature attribution method also based on gradients that outputs the integrated path of the gradient with respect to a reference point as an explanation. The difference between Integrated Gradients and SHAP values is that Integrated Gradients use a single reference point, while Shapley values utilize a background data set. The Gradient Estimator can be viewed as an adaptation of Integrated Gradients, where instead of a single reference point, we reformulate the integral as an expectation and estimate that expectation with the background data.

Integrated gradients are defined as follows:

$$IG(x) = (x_j - \tilde{x}_j) \cdot \int_{\alpha=0}^{1} \frac{\delta g(\tilde{x} + \alpha \cdot (x - \tilde{x}))}{\delta x_j} d\alpha$$

These are the components of the equation:

- x: the data point to be explained.
- $\tilde{x}$: the reference data point. For images, this could be a completely black or gray image.
- g: the gradient function of the gradient-based model with respect to the input feature x_j in our case.
- The integral is along the path between x_j and $\tilde{x}_j$.

The SHAP Gradient Estimator extends this concept by using multiple data points as references and integrating over an entire background dataset.

Here are the implementation details in `shap`:

```
shap.GradientExplainer(model, data)
```

- Compatible with PyTorch, TensorFlow, and Keras.
- Data can be numpy arrays, pandas DataFrames, or torch.tensors.
- The Gradient Estimator is highly versatile, allowing the use of gradients based on parameters, which enables SHAP values to attribute predictions to layers within a neural network. See this example[4].

[4] https://shap.readthedocs.io/en/latest/example_notebooks/image_examples/image_classification/Explain%20an%20Intermediate%20Layer%20of%20VGG16%20on%20ImageNet%20(PyTorch).html?highlight=Gradient

A.10 Deep Estimator: for neural networks

The Deep Estimator is specifically designed for deep neural networks (Chen et al. 2021). This makes the Deep Estimator more model-specific compared to the Gradient Estimator, which can be applied to all gradient-based methods in theory. The Deep Estimator is inspired by the DeepLIFT algorithm (Shrikumar et al. 2017), an attribution method for deep neural networks. To understand how the Deep Estimator works, we first need to discuss DeepLIFT. DeepLIFT explains feature attribution in neural networks by calculating the contribution value Δf for each input feature x_j, comparing the prediction for x with the prediction for a reference point z. The user chooses the reference point, which is usually an "uninformative" data point, such as a blank image for image data. The difference to be explained is $\Delta f(x) - \Delta f(\tilde{x})$. DeepLIFT's attributions, called contribution scores $C_{\Delta x_j \Delta f}$, add up to the total difference: $\sum_{j=1}^{n} C_{\Delta x_j \Delta f} = \Delta f$. This process is similar to how SHAP values are calculated. DeepLIFT does not require x_j to be the model inputs; they can be any neuron layer along the way. This feature is not only a perk of DeepLIFT but also a vital aspect, as DeepLIFT is designed to backpropagate the contributions through the neural network, layer by layer. DeepLIFT employs the concept of "multipliers," defined as follows:

$$m_{\Delta x \Delta f} = \frac{C_{\Delta x \Delta f}}{\Delta x}$$

A multiplier represents the contribution of Δx to Δf divided by Δx. Like a partial derivative ($\frac{\partial f}{\partial x}$) when Δx approaches a very small value, this multiplier is a finite distance. Like derivatives, these multipliers can be backpropagated through the neural network using the chain rule: $m_{\Delta x_j \Delta f} = \sum_{j=1}^{n} m_{\Delta x_j \Delta y_j} m_{\Delta y_j \Delta f}$, where x and y are two consecutive layers of the neural network. DeepLIFT then defines a set of rules for backpropagating the multipliers for different components of the neural networks, using the linear rule for linear units, the "rescale rule" for nonlinear transformations like ReLU and sigmoid, and so on. Positive and negative attributions are separated, which is crucial for backpropagating through nonlinear units.

However, DeepLIFT does not yield SHAP values. Deep SHAP is an adaptation of the DeepLIFT procedure to produce SHAP values. Here are the changes the Deep Estimator incorporates:

- The Deep Estimator uses background data, a set of reference points, instead of a single reference point.
- The multipliers are redefined in terms of SHAP values, which are backpropagated instead of the original DeepLIFT multipliers. Informally: $m_{\Delta x_j \Delta f} = \frac{\phi}{x_j} - \mathbb{E}(X_j)$.
- Another interpretation of the Deep Estimator: it computes the SHAP values in smaller parts of the network first and combines those to obtain SHAP values for the entire network, explaining the prediction from the input, similar to our usual understanding of SHAP.

> **Note**
>
> How large should the background data be for the Deep Estimator? According to the SHAP author[a], 100 is good, and 1000 is very good.
>
> [a] https://shap-lrjball.readthedocs.io/en/latest/generated/shap.DeepExplainer.html#shap.DeepExplainer.shap_values

Implementation details for the `shap` library:

- Compatible with PyTorch and TensorFlow/Keras.
- Supports only pre-implemented neural network operations.
- Using unusual or custom operations may result in errors from the Deep Explainer.
- Complexity increases linearly with the number of background data rows.

A.11 Partition Estimator: For hierarchically grouped data

The Partition Estimator is built on a hierarchy of features, similar to tree-based structures like hierarchical clustering. It iterates through this hierarchy recursively.

How is the hierarchy interpreted? Consider a tree of depth 1 with multiple groups, for example, four groups. Initially, we compute four SHAP values, one for each group. This implies that features grouped together behave as a single entity.

The SHAP value for each group can then be attributed to its individual features. Alternatively, if the hierarchy further splits into subgroups, we attribute the SHAP value at the subgroup level.

Why is this useful? There are instances where we are more interested in a group of features rather than individual ones. For example, multiple feature columns may represent a similar concept, and we're interested in the attribution of the concept, not the individual features. Let's say we're predicting the yield of fruit trees, and we have various soil humidity measurements at different depths. We might not care about the individual attributions to different depths but instead want a SHAP value attributed to the overall soil humidity. The results are not SHAP values but Owen values. Owen values are another solution to the attribution problem in cooperative games. They are similar to SHAP values but assigned to feature groups instead of individual features. Owen values only allow permutations defined by a coalition structure. The computation is identical to SHAP values, except that it imposes a hierarchy.

Partition Estimator also proves useful for image inputs, where image pixels can be grouped into larger regions.

Implementation details:

```
shap.PartitionExplainer(model, partition_tree=None)
```

- A partition tree, a hierarchical clustering of the input features, is a required input. It should follow a format similar to `scipy.cluster.hierarchy`, essentially a matrix.
- Alternatively, you can use `masker.clustering` to utilize a built-in feature clustering in SHAP, which will be the default when `partition_tree=None`.

B The Role of Maskers and Background Data

> **Note**
>
> Maskers are the technical solution for "removing" feature values to compute marginal contributions and SHAP values.

SHAP values are computed by considering many possible contributions. Forming a coalition of features means some features are absent. In games, playing with fewer players can be straightforward, but for machine learning, simulating the absence of feature values requires more effort. This problem is solved in `shap` using maskers.

A masker is a function that takes a data instance and a binary mask, indicating which inputs to mask. The output is the same data point with the designated inputs masked. Depending on the input data type (tabular, text, or image data), maskers employ different strategies for masking. However, they all take a feature values input and a binary vector. This binary vector, like [0,1,...,1], also known as a binary mask, indicates which feature values to mask (the 1s) and which not to (the 0s). For non-masked features, the feature vector position remains untouched. For the masked or absent features (the 1s), feature values are replaced according to the specific masker used. We will explore various maskers in this chapter.

A binary mask can be viewed as a team description, indicating which players participate and which don't. The masker's job is to take a coalition and produce something that can be input into the model. The output can be multiple data points when using background data. These data points from the masker are then input to the model. The model's predictions are obtained and averaged. A masker can be a function that performs this mapping or directly an array or pandas DataFrame, providing a shortcut.

B.1 Masker for tabular data

> **Note**
>
> Typical maskers for tabular data replace absent feature values with samples from a background dataset.

For tabular data, we can provide a background dataset, and the masker replaces missing values with samples from the background data. There are two choices for maskers: Independent masker and Partition masker.

B.1.1 Independent masker

The independent masker replaces missing features with the background data. Technically, the tabular data in SHAP is implemented with the `Independent` masker in `shap`:

```
masker = shap.maskers.Independent(data=X_train)
explainer = shap.LinearExplainer(model, masker=masker)
```

Here is an illustration of how the Independent masker operates, using a simulated background dataset.

```
import shap
import pandas as pd
import numpy as np

# create a dictionary with the column names and data
data = {'f1': np.random.randint(100, 200, 10),
        'f2': np.random.randint(1, 10, 10),
        'f3': np.random.randint(10, 20, 10)}

# create the pandas DataFrame
df = pd.DataFrame(data)
```

```
# print the DataFrame
print(df)
```

```
    f1  f2  f3
0  104   6  16
1  157   3  13
2  160   1  19
3  134   3  15
4  186   2  18
5  140   8  17
6  128   6  12
7  196   4  16
8  125   7  11
9  113   6  17
```

This dataframe is our background dataset. Next, we create a masker and apply it to data point (f1=0, f2=0, f3=0).

```
np.random.seed(2)
m = shap.maskers.Independent(df, max_samples=3)
mask = np.array([1, 0, 1], dtype=np.bool)
print(m(mask=mask, x=np.array([0, 0, 0])))

mask = np.array([False, False, True])
print(m(mask=mask, x=np.array([0, 0, 0])))
```

```
   f1  f2  f3
0   0   1   0
1   0   7   0
2   0   2   0
    f1  f2  f3
0  160   1   0
1  125   7   0
2  186   2   0
```

In this example, we have two masks. The first mask eliminates features f1 and f3, keeping only the feature values f1=0 and f3=0 from x, and drawing f2 from the base data. In the second instance, only feature f3 is preserved.

Masks for tabular data can also be defined through integer vectors that indicate which feature (based on column index) to mask.

B.2 Partition masker

The Partition masker uses both the base dataset and a hierarchical structure of the data. It is deployed with the Partition explainer and includes a max_samples argument that controls the number of data points selected from the base data to compute SHAP values. The clustering type argument specifies the partitioning method, a distance metric for clustering feature values. This can either be a string representing the correlation metric (default is "correlation"), or any distance function from the `scipy.spatial.distance.pdist` module, such as "cosine", "euclidean", or "chebyshev". Alternatively, a numpy array defining the clusters directly can also be provided.

The second option using the array is worth considering as it can be helpful when providing your own feature groupings.

B.3 Maskers for text data

> **Note**
>
> Maskers for text replace tokens with a user-defined token.

For text data, the input is often represented as tokens within models, especially neural networks. These tokens can be represented by (learned) embeddings, a vectorized representation of a token, or other internal representations such as bag-of-words counts or hand-written rules flagging specific words. However, when calculating SHAP values, we are not concerned with the internal representation. The "team" consists of the text fed into the model, and the payout is the model output, whether it's a sentiment score or a probability for the next word in a translation task. The individual players, which can be characters, tokens, or

words, are determined by the user. The granularity at which text is divided into smaller units is termed tokenization, and the individual units are known as tokens. Tokenization can be performed at various levels, such as by character, subword, word, sentence, paragraph, or even custom methods like using n-grams or stopping at specific characters. The choice of tokenization method depends on the goal of the model interpretation.

> **Tip**
>
> Word tokenization is often a good first choice for interpretation with SHAP.

Assuming the input is tokenized by word for the following discussions, the next question is: How do we represent the "absence" of a word? This decision is up to the user. By default, the word is replaced by an empty string, but it could also be replaced with "..." (which is the default in `shap`) or even a randomly chosen word from background data or based on a grammar tool. The chosen method will influence the prediction, as shown in the text chapter.

```
import shap
m = shap.maskers.Text()
s = 'Hello is this the Krusty Krab?'
print(m(s=s, mask = [1,1,1,1,1,1]))
print(m(s=s, mask = [1,0,0,1,0,1]))
print(m(s=s, mask = [1,1,1,1,0,0]))
print(m(s=s, mask = [0,0,0,0,0,0]))
```

```
(array(['Hello is this the Krusty Krab'], dtype='<U29'),)
(array(['Hello ...the ...Krab'], dtype='<U20'),)
(array(['Hello is this the ...'], dtype='<U21'),)
(array(['...'], dtype='<U3'),)
```

B.4 Maskers for image data

> **Note**
>
> For images, maskers substitute missing pixels with blurred versions or employ inpainting techniques.

Similar to text data, the representation of an image for SHAP can be independent of its representation for the model, as long as there is a mapping between the SHAP version and the model version to calculate the marginal contributions.

For images, we have two options for players: individual pixels or larger units containing multiple pixels. The composition of these units is flexible, and they can be created based on a grid, such as dividing a 224x224 image into 196 rectangles of size 16x16. In this case, the number of players would be 196 instead of 224x224 = 50,176.

So, what does the masker do? It replaces parts of the image. In theory, you could use data from a background dataset, but that would be strange. Alternatively, you could replace parts with gray pixels or another neutral color, but that could also result in unusual images. SHAP implements blurring and inpainting methods to remove or guess content from the rest of the image.

The absence of a team member is addressed in this manner. This masker is implemented in SHAP's Image masker.