Chapter 9. Interpretability

Model interpretability helps you develop a deeper understanding of the workings of your models.

Interpretability itself does not have a mathematical definition. Biran and Cotton provided a good definition of interpretability. They wrote that systems, or in this case models, *"are interpretable if their operations can be understood by a human, either through introspection or through a produced explanation."* In other words, if there is some way for a human to figure out why a model produced a certain result, the model is interpretable.

The term *explainability* is also often used, but the distinction between interpretability and explainability is not well-defined. In this chapter, we will primarily refer to both as *interpretability*.

Interpretability is becoming both increasingly important and increasingly difficult as models become more and more complex. But the good news is that the techniques for achieving interpretability are improving as well.

Explainable AI

Interpretability is part of a larger field known as *Responsible AI*. The development of AI, and the successful application of AI to more and more

problems, has resulted in rapid growth in the ability to perform tasks that were previously not possible. This has created many great new opportunities. But there are questions about how much trust we should place in the results of these models. Sometimes there also are questions about how responsibly models handle a number of factors that influence people and can cause harm.

Interpretability is important for Responsible AI because we need to understand how models generated their results. The results generated by a model can be explained in different ways. One of the most dependable techniques is to create a model architecture that is inherently explainable. A simple example of this is decision tree–based models, which by their nature are explainable. But there are increasingly advanced and complex model architectures that can now also be designed to be inherently explainable.

Why is interpretability in AI so important? Well, fundamentally it's because we need to explain the results and the decisions that are made by our models. This is especially true for models with high sensitivity, including natural language models, which when confronted with certain examples can generate wildly wrong (or offensive, dangerous, or misleading) results.

Interpretability is also important for assessing vulnerability to attacks (discussed next), which we need to evaluate on an ongoing basis, and not just after an attack has already happened. Fairness is a key issue as well, since we want to make sure we are treating every user of our model fairly. A

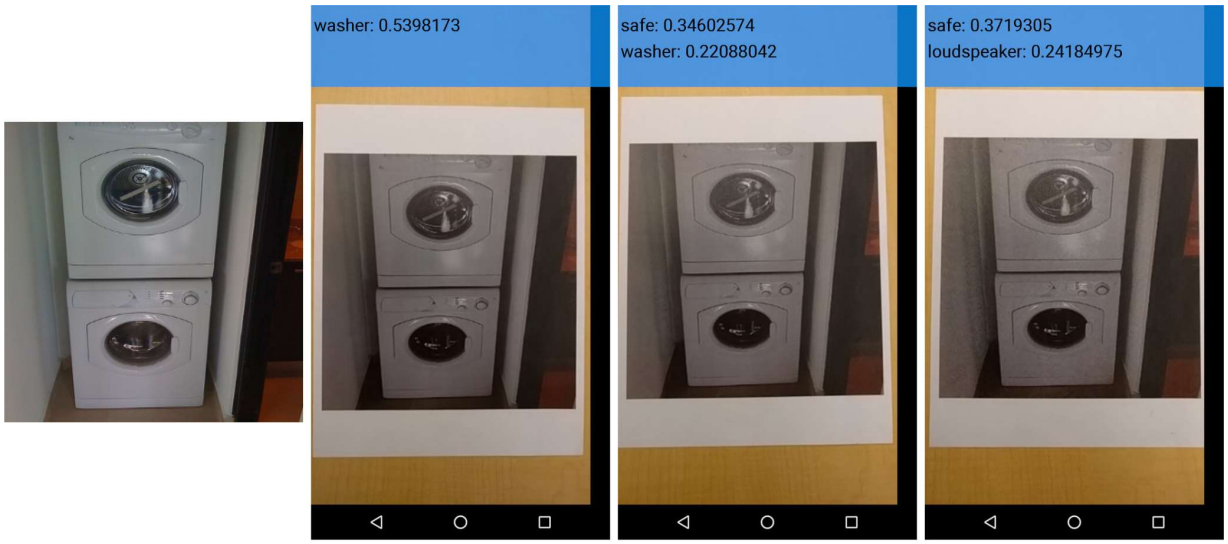
lack of fairness can also impact our reputation and branding. This is especially true in cases where customers or other stakeholders may question or challenge our model's decisions, but really it's true in any case where we generate a prediction. And of course, there are legal and regulatory concerns, especially when someone is so unhappy that they challenge us and our model in court, or when our model's results lead to an action that causes harm.

Deep neural networks (DNNs) can be fooled into misclassifying inputs to produce results with no resemblance to the true category. This is easiest to see in examples of image classification, but fundamentally it can occur with any model architecture. The example in Figure 9-1 demonstrates a black-box attack in which the attack is constructed without access to the model. The example is based on a phone app for image classification using physical adversarial examples.

Figure 9-1 shows a clean image of a stackable washing machine and dryer from the dataset (image A on the left) that is used to generate one clean and two adversarial images with various degrees of perturbation. Images B, C, and D show the clean and adversarial images, and the results of using a TensorFlow Camera Demo app to classify them.

Image B is recognized correctly as a "stackable washing machine and dryer," while increasing the adversarial perturbation in images C and D results in greater misclassification. The key result is that in image D the

model thinks the appliance is either a safe or a loudspeaker, but definitely not a stackable washing machine and dryer. Looking at the image, would you agree with the model? Can you even see the adversarial perturbation that was applied? It's not easy.



(a) Image from dataset (b) Clean image (c) Adv. image, $\epsilon = 4$ (d) Adv. image, $\epsilon = 8$

Figure 9-1. Misclassifying appliances (source: Kurakin et al., 2017)

Figure 9-2 shows what is perhaps the most famous example of this kind of model attack. By adding an imperceptibly small amount of well-crafted noise, an image of a panda can be misclassified as a gibbon—with a *99.3% confidence!* This is much higher than the original confidence that the model had that it was a panda.

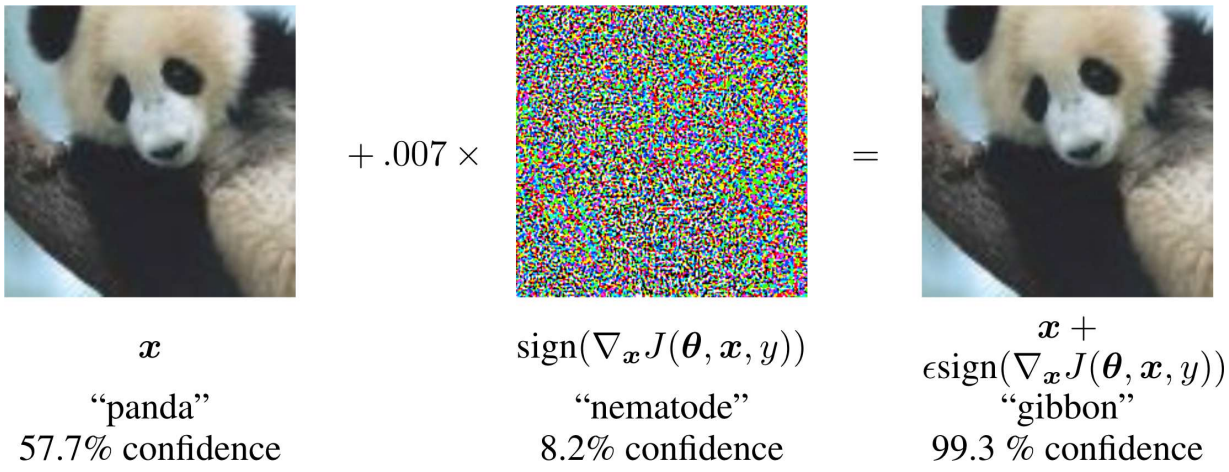


Figure 9-2. Misclassifying a panda (Goodfellow et al., 2015)

Developing a robust understanding of how a model makes predictions through tools and techniques designed for model interpretation is one part of guarding against attacks such as these. The process of discovery while studying a model can also reveal vulnerabilities to attacks before they become fire drills.

Model Interpretation Methods

Let’s look now at some of the basic ways to interpret models. There are two broad, overlapping categories: techniques that can be applied to models in general and techniques that can be applied to model architectures that are inherently interpretable. Practically speaking, the level of effort required needs to be feasible as well, and one measure of the interpretability of models is the amount of effort or analysis required to understand a given result.

Ideally, you would like to be able to query the model to understand why and how it reached a particular decision. Why did the model behave in a certain way? You would like to be able to identify and validate the relevant features driving the model's outputs. Doing so will help you develop trust in the reliability of the predictive system, even in unforeseen circumstances. This diagnosis will help ensure accountability and confidence in the safety of the model.

Ideally, you should also be able to validate any given data point to demonstrate to business stakeholders and peers that the model works as expected, but in practice this can be difficult. Being able to do this will help assure stakeholders, including your users and the public, of the transparency of the model.

What information can the model provide to avoid prediction errors? In a perfect world, you should be able to query and understand latent variable interactions to evaluate and understand, in a timely manner, what features are driving predictions, but in practice this can be difficult. Tools like the Learning Interpretability Tool (LIT; see Chapter 8) can help you visualize, explore, and understand your model.

Method Categories

There are some criteria that can be used for categorizing model interpretation methods. For example, interpretability methods can be

grouped based on whether they're intrinsic or post hoc. They can also be model specific or model agnostic. And they can be grouped according to whether they are local or global. Let's discuss each of these criteria.

Intrinsic or post hoc?

One way to group model interpretability methods is by whether the model itself is intrinsically interpretable or whether it must be interpreted as a black box. Model architectures that are intrinsically interpretable have been around for a long time, and the classic examples of this are linear models and tree-based models. More recently, however, more advanced model architectures such as lattice models have been developed to enable both interpretability and a high degree of accuracy on complex modeling problems. Lattice models, for example, can match, or in some cases exceed, the accuracy of neural networks. In general, an intrinsically interpretable model provides a higher degree of certainty than a post hoc method does as to why it generated a particular result.

Post hoc methods treat models as black boxes, and they often don't distinguish between different model architectures. They tend to treat all models the same, and you apply them after training to try to examine particular results so that you can understand what caused the model to generate them. There are some post hoc methods, especially for convolutional networks, that do inspect the layers within the network to try to understand how results were generated. However, there is always some

level of uncertainty about whether the interpretation of the reasons for certain results is correct or not, since post hoc methods don’t evaluate the actual sequence of operations that led to the generation of the results. Examples of post hoc analyses include feature importance and partial dependency plots.

The various interpretation methods can also be roughly classified according to the types of results they produce. Some methods create a summary of feature statistics. Some methods return a single value for a feature; for example, feature importance returns a single number per feature. A more complex example would be pairwise feature interaction strength, which associates a number with each pair of features.

Some methods, such as partial dependence plots, rely on visualization to summarize features. Partial dependence plots are curves that show a feature and its average predicted output. In this case, visualizing the curve is more meaningful and intuitive than simply representing the values in a table.

Some model-specific methods look at model internals. The interpretation of intrinsically interpretable models falls into this category. For example, for less complex models, such as linear models, you can look at their learned weights to produce an interpretation. Similarly, the learned tree structure in tree-based models serves as an interpretation. In lattice models, the parameters of each layer are the output of that layer, which makes it relatively easy to analyze, understand, and debug each part of the model.

Some methods examine particular data points. One such method is counterfactual explanations. Counterfactual explanations are used to explain the prediction of a datapoint. With this method, you find another data point by changing some features so that the predicted output changes in a relevant way. The change should be significant. For example, the new data point should be of a different predicted class.

Model specific or model agnostic?

Model-specific methods are limited to specific model types. For example, the interpretation of regression weights in linear models is model specific. By definition, the techniques for interpreting intrinsically interpretable models are model specific. But model-specific methods are not limited to intrinsically interpretable models. There are also tools that specifically focus on neural network interpretation.

Model-agnostic methods are not specific to any particular model and can be applied to any model after it is trained. Essentially they are post hoc methods. These methods do not have access to the internals of the model, such as the weights and parameters. They usually work by analyzing feature input and output pairs and trying to infer relationships.

Local or global?

In addition to grouping interpretation methods as model agnostic or model specific, they can be grouped by whether they generate interpretations that are local or global.

Interpretability methods can be local or global based on whether the method explains an individual prediction or the entire model behavior. Sometimes the scope can be in between local and global.

A local interpretability method explains an individual prediction. For example, it can explain feature attribution in the prediction of a single example in the dataset. Feature attributions measure how much each feature contributed to the predictions for a given result.

Figure 9-3 shows a feature attribution using a library called SHAP (we will discuss SHAP in detail later in this chapter), for the prediction of a single example by a regression model trained on the diabetes dataset. The model predicts the disease progression one year after the baseline. The diagram shows the contribution of features in pushing model output from the base value toward the actual model output. The plot shows the balance of the forces reaching equilibrium at 197.62. Forces on the left side push that equilibrium higher, and forces on the right side push it lower.

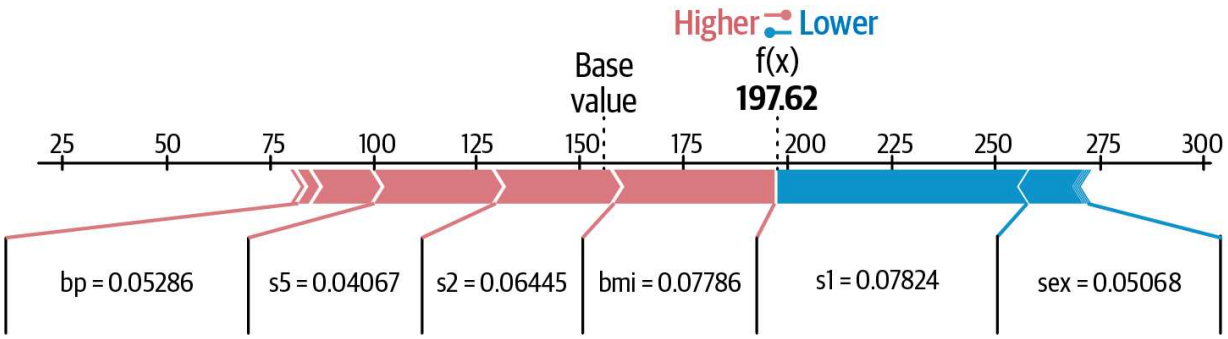


Figure 9-3. SHAP feature attribution

Interpretability methods can also be global. Global interpretability methods explain the entire model behavior. For example, if the method creates a summary of feature attributions for predictions on the entire test dataset, it can be considered global.

Figure 9-4 shows an example of a global explanation created by the SHAP library. It shows feature attributions (the SHAP value) of every feature, for every sample, for predictions in the diabetes prediction dataset. The color represents the feature value.[1] As S1 (total serum cholesterol) increases, it tends to lead to a decrease in the likelihood of diabetes. Since this explanation shows an overview of attributions of all features on all instances in the dataset, it should be considered global.

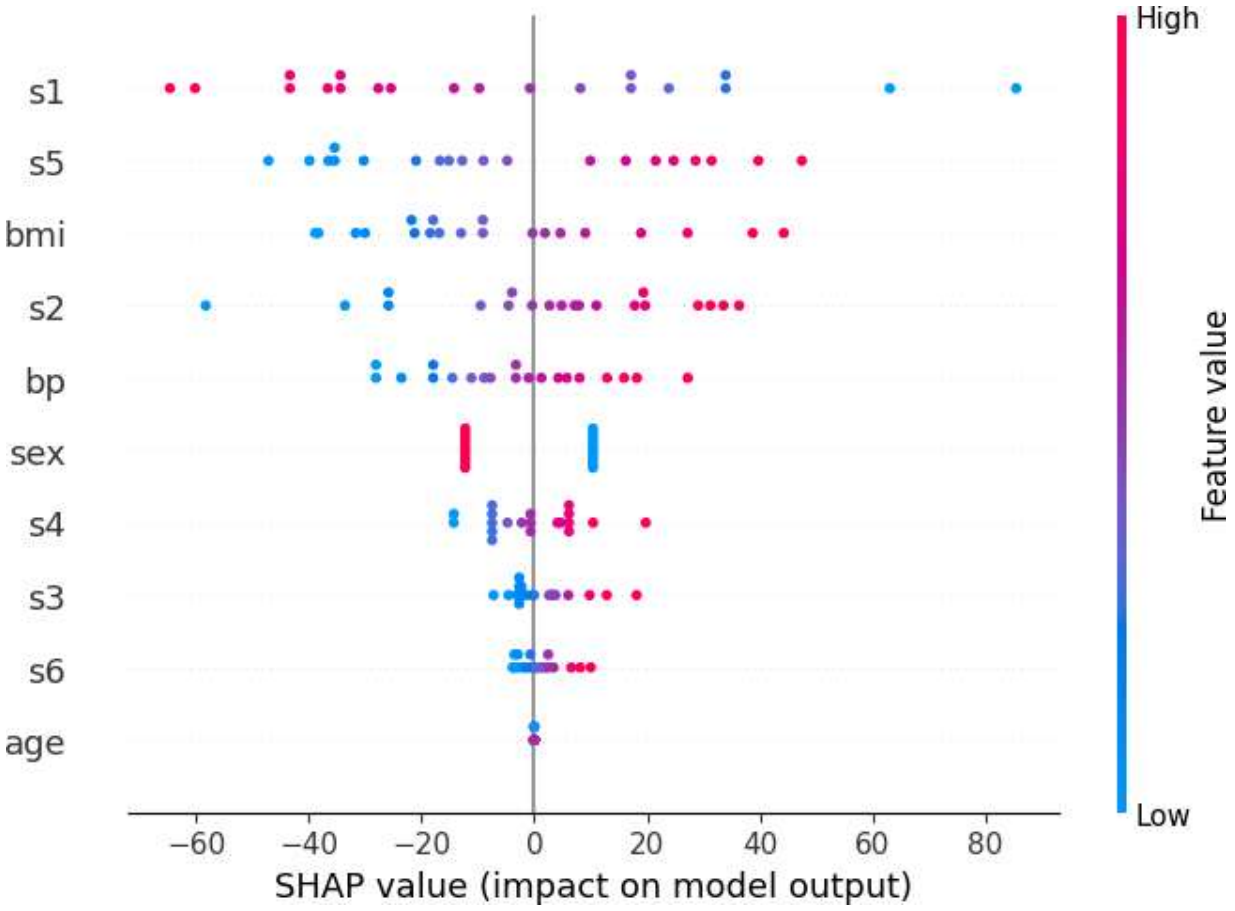


Figure 9-4. SHAP global explanation[2]

Intrinsically Interpretable Models

Since the early days of statistical analysis and ML, there have been model architectures that are intrinsically interpretable. Let’s look at those now, along with more recent advances, and learn how they can help improve interpretability.

What exactly do we mean by an intrinsically interpretable model?

One definition is that the workings of the model are transparent enough and intuitive enough that they make it relatively easy to understand how the model produced a particular result by examining the model itself. Many classic models are highly interpretable, such as tree-based models and linear models.

Although we've seen neural networks that are able to produce really amazing results, one of the issues with them is that they tend to be very opaque, especially the larger, more complex architectures, which makes them black boxes when we're trying to interpret them. That limits our ability to interpret their results and requires us to use post hoc analysis tools to try to understand how they reached a particular result.

However, newer architectures have been created that are designed specifically for interpretability, and yet they retain the power of DNNs. This continues to be an active field of research.

One key characteristic that helps improve interpretability is when features are monotonic. *Monotonic* means that contribution of the feature toward the model result either consistently increases, decreases, or stays even as the feature value changes. This matches the domain knowledge for many features in many kinds of problems, so when you're trying to understand a model result, if the features are monotonic, it matches your intuition about the reality of the world you're trying to model.

For example, say you're trying to create a model to predict the value of a used car. When all other features are held constant, the more miles the car has on it the lower its value should be. You don't expect a car with more miles to be worth more than it was when it had fewer miles, all other things being equal. This matches your knowledge of the world, and so your model should match it too, and the mileage feature should be monotonic. In Figure 9-5, two of the curves are monotonic, while one is not because it does not consistently increase, decrease, or remain the same.

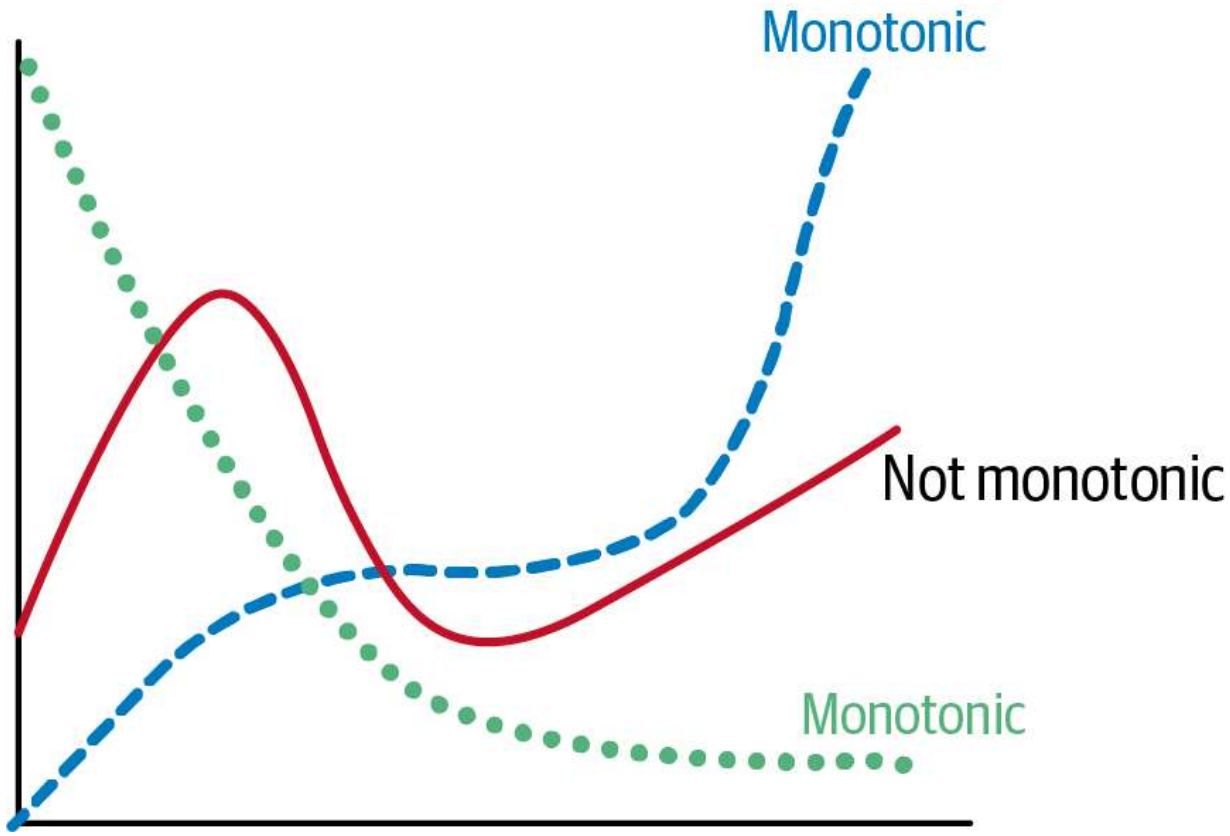


Figure 9-5. An example of monotonicity

Let's look at a few architectures that are considered interpretable. First, linear models are very interpretable because linear relationships are easy to

understand and interpret, and the features of linear models are always monotonic. Some other model architectures have linear aspects to them. For example, when used for regression, rule fit models are linear. And in all cases, TensorFlow Lattice models use linear interpolation between lattice points, which we'll learn about soon.

Some models can automatically include feature interactions, or include constraints on feature interactions. In theory, you can include feature interaction in all models through feature engineering. Interactions that match our domain knowledge tend to make models more interpretable.

Depending on the characteristics of the loss surface you are trying to model, more complex model architectures can achieve higher accuracy. This often comes at a price in terms of interpretability. For many of the reasons discussed earlier, interpretability can be a strict requirement of models, and so you need to find a balance between models that you can interpret and models that generate the accuracy you need. Again, some newer architectures have been created that deliver far greater accuracy as well as good interpretability. TensorFlow Lattice is one example of this kind of architecture.

Probably the ultimate in interpretability is our old friend, linear regression. For most developers, linear regression will be the first model they learn about. It's very easy to understand the relationship between feature contributions, even for multivariate linear regression. As feature values

increase or decrease, their contribution to the model results also increases or decreases.

The example in Figure 9-6 models the number of chirps per minute that a cricket will make based on the temperature of the air. This is a very simple linear relationship, and so linear regression models it well. By the way, this also means that when you're out at night, if you listen carefully to the crickets and count how many chirps they make, you can measure the temperature of the air. Check out Dolbear's law to learn more.

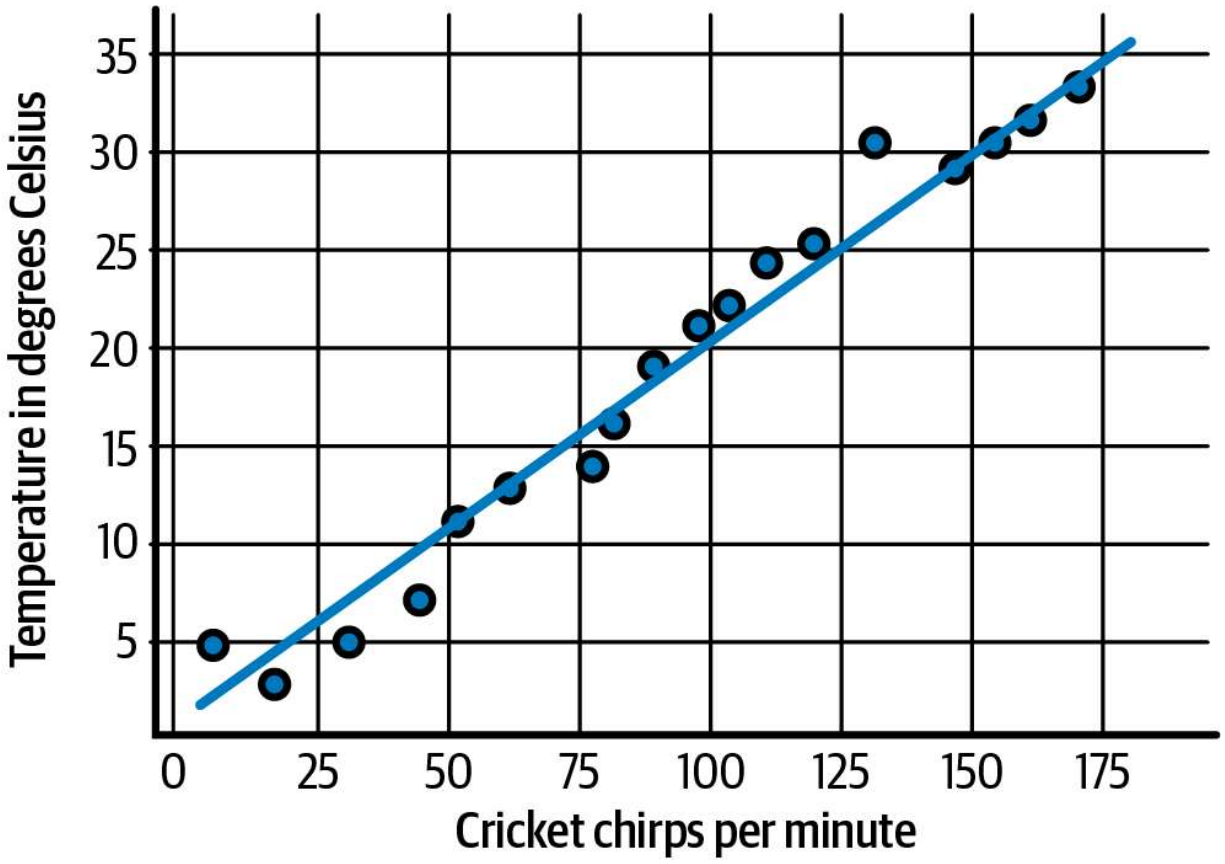


Figure 9-6. A linear model of cricket chirps

Feature importance

Of course, the actual contribution of a feature to the result of the model will depend on its weight. This is especially easy to see for linear models. For numerical features, an increase or decrease of one unit in a feature increases or decreases the prediction based on the value of the corresponding weight. For binary features, the prediction is increased or decreased by the value of the weight, based on whether the feature's value is a 1 or a 0. Categorical features are usually divided into several individual features with one-hot encoding, each of which has a weight. With one-hot encoding, only one of the categories will be set, so only one of the weights will be included in the model result.

How can we determine the relevance of a given feature for making predictions?

Feature importance tells us how important a feature is for generating a model result. The more important a feature is, the more we want to include it in our feature vector. But feature importance for different models is calculated differently, because different models calculate results differently.

For linear regression models, the absolute value of a feature's t-statistic is a good measure of that feature's importance. The *t-statistic* is the learned or estimated weight of the feature, scaled by its standard error. So, the importance of a feature increases as its weight increases. But the more

variance the weight has (i.e., the less certain we are about the correct value of the weight), the less important the feature is.

Lattice models

A lattice model, as shown in Figure 9-7, overlays a grid onto the feature space and sets the values of the function that it's trying to learn at each of the vertices of the grid. As prediction requests come in, if they don't fall directly on a vertex, the result is interpolated using linear interpolation from the nearest vertices of the grid.

One of the benefits of using a lattice model is that you can regularize the model and greatly reduce sensitivity, even to examples that are outside the coverage of the training data, by imposing a regular grid on the feature space.

TensorFlow Lattice models go beyond simple lattice models. TensorFlow Lattice further allows you to add constraints and inject domain knowledge into the model. The graphs in Figure 9-8 show the benefits of regularization and domain knowledge. Compare the one on the top left to the one on the bottom right, and notice how close the model is to the ground truth compared to other kinds of models.

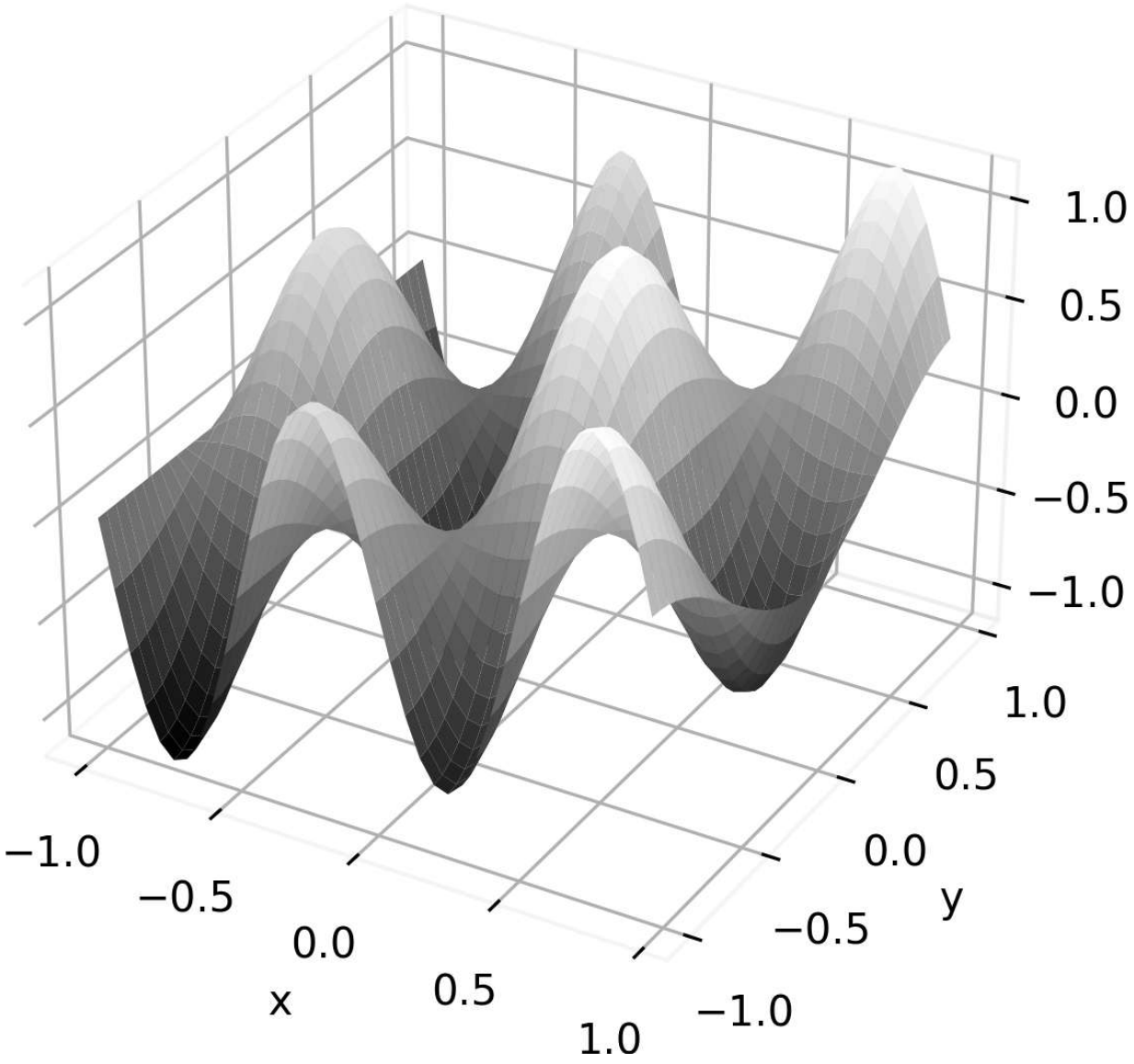


Figure 9-7. A lattice model

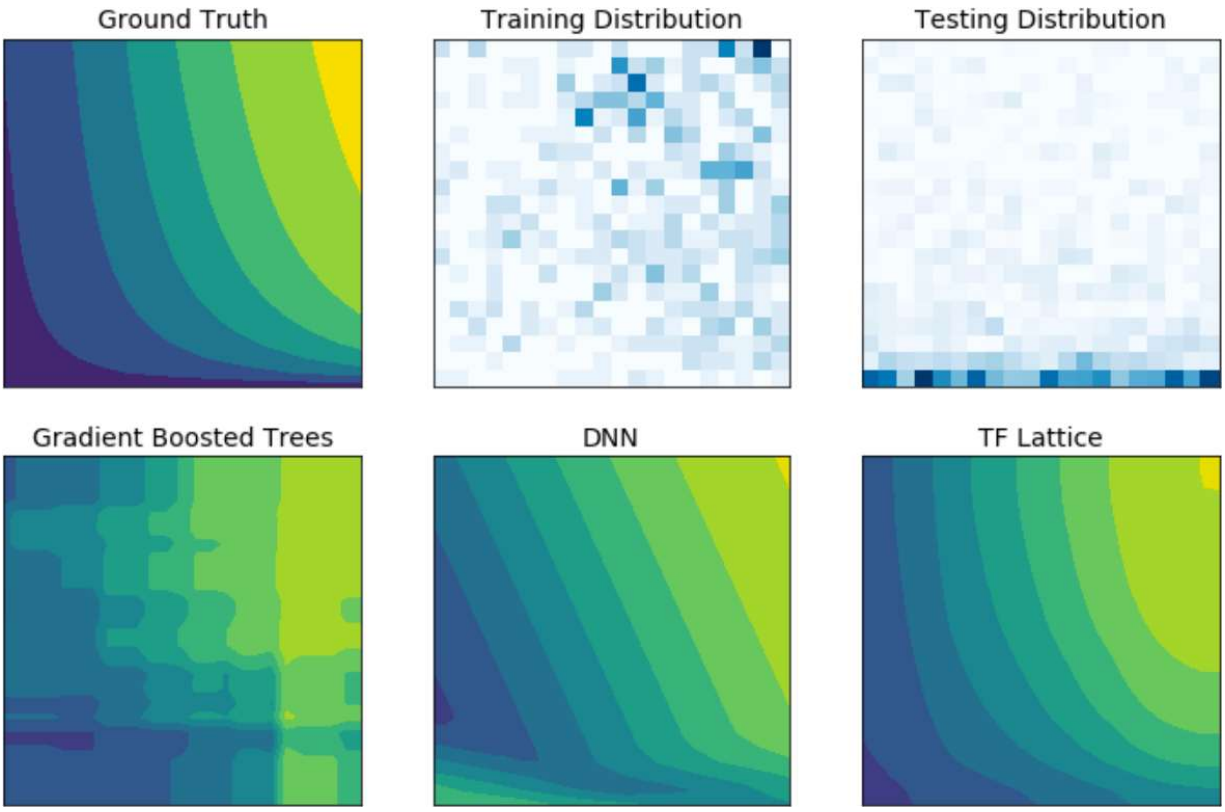


Figure 9-8. Modeling with TensorFlow Lattice

When you know that certain features in your domain are monotonic, or convex, or that one or more features interact, TensorFlow Lattice enables you to inject that knowledge into the model as it learns. For interpretability, this means feature values and results are likely to match your domain knowledge for what you expect your results to look like.

You can also express relationships or interactions between features to suggest that one feature reflects trust in another feature. For example, a higher number of reviews makes you more confident in the average star rating of a restaurant. You might have considered that yourself when shopping online. All of these constraints, based on your domain knowledge

or what you know about the world you're trying to model, help the model produce results that make sense, which helps make them interpretable. Also, since the model uses linear interpolation between vertices, it has many of the benefits of linear models in terms of interpretability.

Along with all the benefits of adding constraints based on domain knowledge, TensorFlow Lattice models also have a level of accuracy on complex problems that is similar to DNNs, with the added benefit that TensorFlow Lattice models are easier to interpret than neural networks.

However, lattice models do have a weakness. Dimensionality is their kryptonite.

The number of parameters of a lattice layer increases exponentially with the number of input features, which creates problems with scaling for datasets with a large number of features. As a rough rule of thumb, you're probably OK with no more than 20 features, but this will also depend on the number of vertices you specify. There is another way to deal with this dimensionality kryptonite, however, and that is to use ensembling, but that is beyond the scope of this discussion.

Model-Agnostic Methods

Unfortunately, you can't always work with models that are intrinsically interpretable. For a variety of reasons, you may be asked to try to interpret

the results of models that are not inherently easy to interpret. Fortunately, there are several methods available that are not specific to particular types of models—in other words, they are model agnostic.

Model-agnostic methods separate the explanations from the model. These methods can be applied to any model after it's been trained. For example, they can be applied to linear regression or decision trees, and even black-box models like neural networks.

The desirable characteristics of a model-agnostic method include model flexibility and explanation flexibility. The explanations shouldn't be limited to a certain type. The method should be able to provide an explanation as a formula, or in some cases explanations can be graphical, perhaps for feature importance.

These methods also need to have representation flexibility. The feature representations used should make sense in the context of the model being explained. Let's take the example of a text classifier that uses word embeddings. It would make sense for the presence of individual words to be used in the explanation in this case.

There are many model-agnostic methods that are currently being used—too many to go into in detail here. So we will only discuss two: partial dependence plots and permutation feature importance.

Partial dependence plots

Partial dependence plots (PDPs) help you understand the effect that particular features have on the model results you're seeing, as well as the relationship between those features and the targets or labels in your training data. PDPs typically concentrate on the marginal impact caused by one or two features on the model results. Those relationships could be linear and/or monotonic, or they could be of a more complex type. For example, for a linear regression model, a PDP will always show a linear, monotonic relationship. Partial dependence plotting is a global method, since it considers all instances and evaluates the global relationship between the features and the results. The following formula shows how the average marginal effect on the result for given values of the features is calculated:

$$\widehat{f}_{x_S}(x_S) = \frac{1}{n}\sum_{i=1}^{n}\widehat{f}\left(x_S, x_C^{(i)}\right)$$

In the preceding formula, the partial function $\widehat{f}_{x_S}$ is estimated using the Monte Carlo method. The equation shows the estimation of the partial function, where n is the number of examples in the training dataset, S is the features that we're interested in, and C is all the other features.

The partial function tells us what the average marginal effect on the result is for given values of the features in S. In this formula, $x_{\mathbf{c}}^{(\mathbf{i})}$ are feature values for the features we're not interested in.

The PDP makes the assumption that the features in C are not correlated with the features in S.

Figure 9-9 shows a random forest model trained on a bike rentals dataset to predict the number of bikes rented per day, given a set of features that include temperature, humidity, and wind speed. These are the PDPs for temperature, humidity, and wind speed. Notice that as the temperature increases up to about 15°C (59°F), more people are likely to rent a bike. This makes sense, because people like to ride bikes when the weather is nice, and at that temperature, we'd say it's just starting to get nice. But notice that this trend first levels off and then starts to fall off above about 25°C (77°F). You can also see that humidity is a factor, and that above about 60% humidity people start to get less interested in riding bikes. How about you? Do these plots match your bike riding preferences?

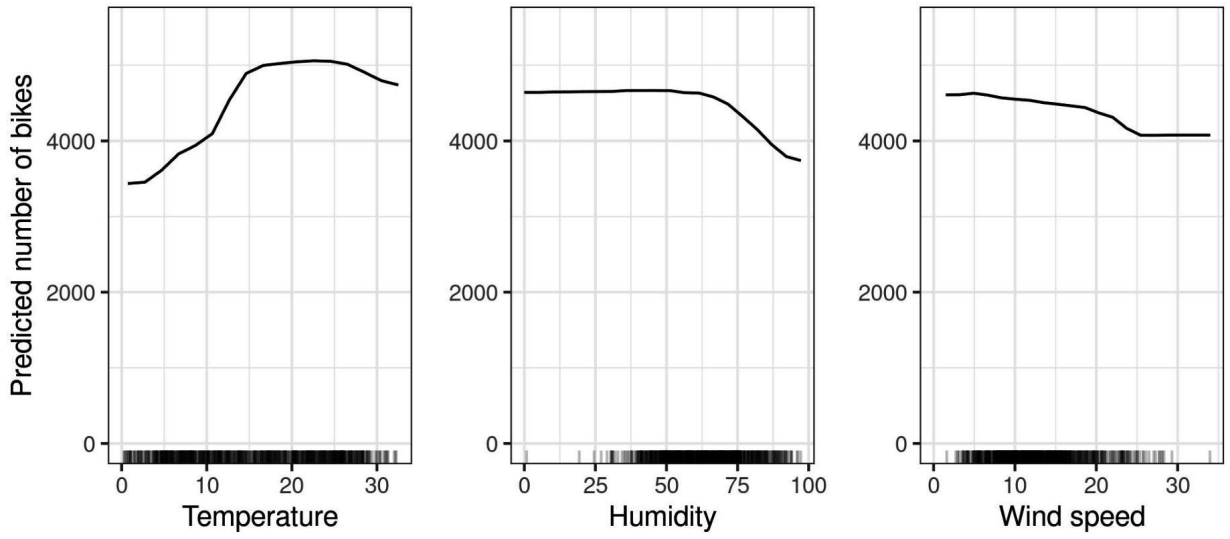


Figure 9-9. PDP plots for bike rentals (with permission from Christoph Molnar, *Interpretable Machine Learning*, 2024)

To calculate a PDP for categorical features, we force all instances to have the same category value. Figure 9-10 shows the plot for the categorical feature `Season` in the bike rentals dataset. It has four possible values: `Spring`, `Summer`, `Fall`, and `Winter`. To calculate the PDP for `Summer` we force all instances in the dataset to have `value = 'summer'` for the `Season` feature.

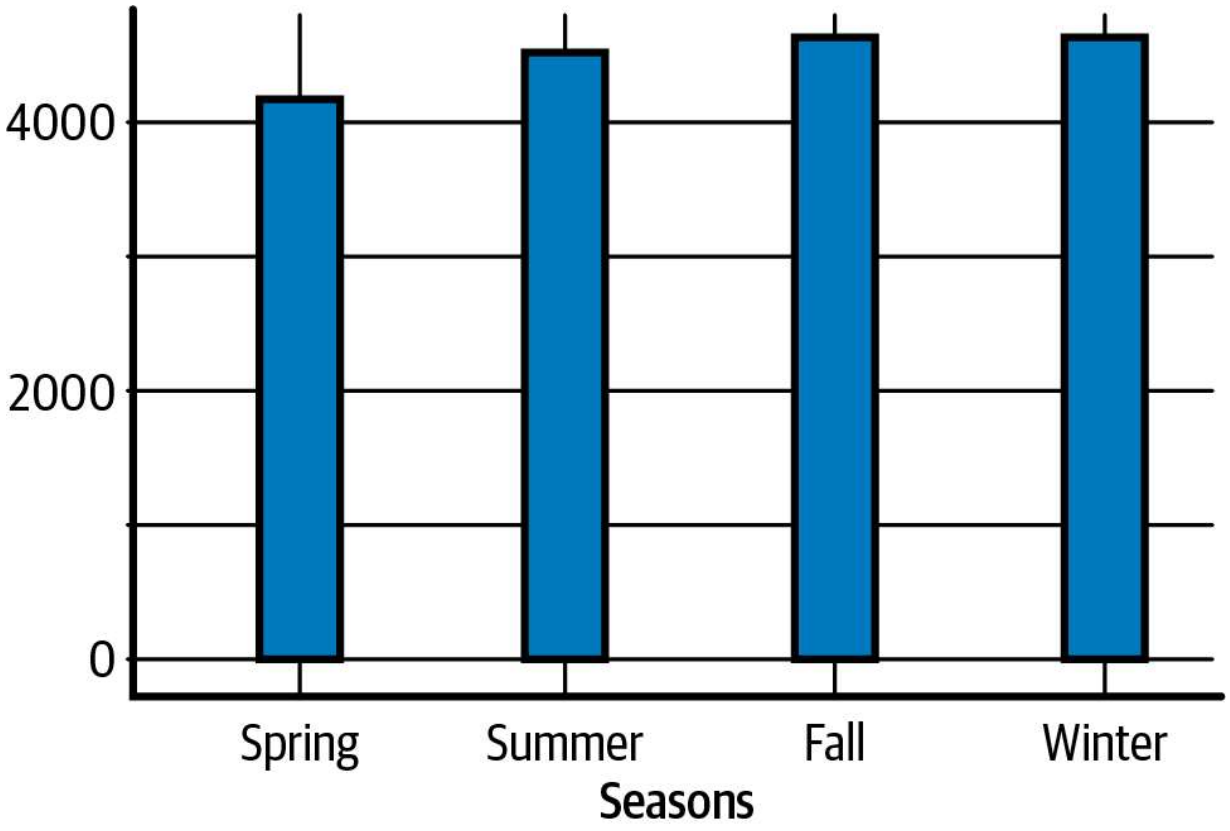


Figure 9-10. A PDP for a categorical feature (with permission from Christoph Molnar, *Interpretable Machine Learning*, 2024)

Notice that there isn't much of an effect of change in seasons on bike rentals, except in spring when the number of rentals is somewhat lower.

Frankly, we wouldn’t expect that to be the case, but that’s what the data is telling us! You can’t always trust your intuition.

There are some clear advantages to using a PDP. First, the results tend to be intuitive, especially when the features are not correlated. When they’re not correlated, a PDP shows how the average prediction changes when a feature is changed. The interpretation of a PDP is also usually causal in the sense that if we change a feature and measure the changes in the results, we expect the results to be consistent. Finally, a PDP is fairly easy to implement with some of the growing list of open source tools that are available.

Like most things, however, there are some disadvantages to a PDP. Realistically, you can only really work with two features at a time, because humans have a hard time visualizing more than three dimensions. We’re not sure we would blame PDPs for that.

A more serious limitation is the assumption of independence. A PDP assumes the features you’re analyzing (C, in the preceding formula) aren’t correlated with other features (S). As we learned in our discussion of feature selection, it’s a good idea to eliminate correlated features anyway. But if you do still have correlated features, a PDP doesn’t work quite right.

For example, suppose you want to predict how fast a person walks, given the person’s height and weight. If height is in C and weight is in S, a PDP

will assume that height and weight aren’t correlated, which is obviously a false assumption. As a result, we might include a person with a height of 6 feet, 5 inches (2 meters) and a weight of 110 lb (50 kg), which is a bit unrealistic even for fashion models, although when we searched this online we were shocked to learn that some are actually pretty close. Anyway, you get the idea: correlated features are bad.

Permutation feature importance

Permutation feature importance is a way of measuring the importance of a feature. Permuting a feature breaks the relationship between a feature and the model result, essentially by assigning a nearly random value to the feature.

For permutation feature importance, we measure the importance of a feature by measuring the increase in the prediction error after permuting the feature. A feature is “important” if shuffling its values increases the model error, because in this case the model relies on the feature for the prediction. A feature is “unimportant” if shuffling its values leaves the model error unchanged. Again, if we find that we have unimportant features, we should really consider removing them from our feature vector. The amount by which the feature changes the model error gives us a value for the feature importance.

This is the basic algorithm. The inputs are the model, the features, the labels or targets, and our error metric. You start by measuring the model error with all the true values of the features. Next, you start an iterative process for each feature where:

1. You first permute the values of the feature you're examining and measure the change in the model error.
2. You express the feature importance either as a ratio of the permuted error to the original error or as the difference between the two errors.
3. You then sort by feature importance to determine the least important features.

Permutation feature importance has a nice interpretation because feature importance is the increase in model error when the feature's information is destroyed. It's a highly compressed, global insight into the model's behavior. Since by permuting the feature you also destroy the interaction effects with other features, it also shows the interactions between features. This means it accounts for both the main feature effect and the interaction effects on model performance. And a big advantage is that it doesn't require retraining the model. Some other methods suggest deleting a feature, retraining the model, and then comparing the model error. Since retraining a model can take a long time and require significant resources, not requiring that is a big advantage.

There are, however, some disadvantages inherent to using permutation feature importance. For one, it's unclear whether you should use your training data or your test data to measure permutation feature importance, as there are concerns with both options. Measuring permutation feature importance using your training data means your measure can reflect the model's overfitting on features, not the true predictive value of those features. On the other hand, measuring permutation feature importance using your test data means you will have a smaller test set to work with (if you use a subset of test data solely for measuring permutation feature importance) or you will bias your model performance measurement. And like with PDPs, correlated features are once again a problem. You also need to have access to the original labeled training dataset, so if you're getting the model from someone else and they don't give you that, you can't use permutation feature importance.

Local Interpretable Model-Agnostic Explanations

Local Interpretable Model-agnostic Explanations (LIME) is a popular and well-known framework for creating local interpretations of model results. The idea is quite intuitive. First, forget about the training data, and imagine you only have the black-box model where you can input data points and get the predictions of the model. You can probe the box as often as you want. Your goal is to understand why the model made a certain prediction. LIME is one of the techniques included in LIT (see Chapter 8).

LIME tests what happens to the predictions when you give variations of your data to the model. LIME generates a new dataset consisting of permuted samples and the corresponding predictions of the model. With this new dataset, LIME then trains an interpretable model, which is weighted by the distance from the sampled instances to the result you're interpreting. The interpretable model can be anything that is easily interpretable, like a linear model or a decision tree.

The new model should be a reasonably good approximation of the model results locally, but it does not have to be a good global approximation. This kind of accuracy is also called *local fidelity*. You then explain the prediction by interpreting the new local model, which as we said is easily interpretable.

Shapley Values

The Shapley value is a concept from cooperative game theory. It was named after Lloyd Shapley. He introduced the concept in 1951 and later won the Nobel Prize for the discovery.

Imagine that a group of players cooperates, and this results in an overall gain because of their cooperation. Since some players may contribute more than others, or may have different amounts of bargaining power, how should we distribute the gains among the players? Or phrased differently, how important is each player to the overall cooperation, and what payoff

can the player reasonably expect? The Shapley value provides one possible answer to this question.

For ML and interpretability, the “players” are the features of the dataset, and we’re using the Shapley value to determine how much each feature contributes to the results. Knowing how the features contribute will help you understand how important they were in generating the model’s result. Because the Shapley value is not specific to any particular type of model, it can be used regardless of the model architecture.

That was a quick overview of the ideas behind the concept of the Shapley value. Let’s now focus on a concrete example. Suppose you trained a model to predict truck prices. You need to explain why the model predicts a $42,000 price for a truck. What data do we have to work with? Well, in this example the car is a pickup truck, is fully electric, and has a half-ton capacity.

The average prediction of all half-ton pickup trucks is $36,000, but the model predicts $42,000 for this particular truck. Why?

Shapley values come from game theory, so let’s clarify how to apply them to ML interpretability. The “game” is the prediction task for a single instance of the dataset. The “gain” is the actual prediction for this instance, minus the average prediction for all instances. The “players” are the feature values of the instance that collaborate to produce the gain. In the truck

example, the feature values engine = electric and capacity = ½ ton worked together to achieve the prediction of $42,000.

Our goal is to explain the difference between the actual prediction ($42,000) and the average prediction ($36,000), which is a gain of $6,000.

One possible explanation could be that the half-ton capacity contributed $36,000 and the electric engine contributed $6,000. The contributions add up to $6,000: the final prediction minus the mean predicted truck price. You could think of that as the absolute value, $6,000, or you could also think of it as the percentage of the mean, which is about 16%.

Unlike perhaps any other method of interpreting model results, Shapley values are based on a solid theoretical foundation. Other methods make intuitive sense, which is an important factor for interpretability, but they don’t have the same rigorous theoretical foundation. This is one of the reasons Shapley was awarded a Nobel Prize for his work. The theory defines four properties that must be satisfied: Efficiency, Symmetry, Dummy, and Additivity.

One key advantage of Shapley values is that they are fairly distributed among the feature values of an instance. Some have argued that Shapley might be the *only* method to deliver a full explanation. In situations where the law requires interpretability—such as the European Union’s “right to explanations”—some feel that the Shapley value might be the only legally

compliant method, because it is based on a solid theory and distributes the effects fairly.

The Shapley value also allows contrastive explanations. Instead of comparing a prediction to the mean prediction of the entire dataset, you could compare it to a subset, or even to a single data point. This ability to contrast is something that local models like LIME do not have.

Like any method, Shapley has some disadvantages. Probably the most important is that it’s computationally expensive, which in a large percentage of real-world cases means it’s only feasible to calculate an approximate solution. It can also be easily misinterpreted. The Shapley value is not the difference of the predicted value after removing the feature from the model training. It’s the contribution of a feature value to the difference between the actual prediction and the mean prediction.

Unlike some other methods, Shapley does not create a model. This means you can’t use it to test changes in the input, such as “If I change to a hybrid truck, how does it change the prediction?”

And finally, like many other methods, it does not work well when the features are correlated. But you already know you should have removed correlated features from your feature vector when you were doing feature selection, so that’s not a problem for you, right? Well, hopefully anyway, but it’s something to be aware of.

If you want to only explain a few of your features or if model interpretability isn’t super critical, Shapley is probably the wrong method to use. Shapley always uses all the features. Humans often prefer selective explanations, such as those produced by LIME and similar methods, so those might be a better choice for explanations that laypersons have to deal with. Another solution is to use SHAP, which is based on the Shapley value but can also provide explanations with only a few features. We’ll discuss SHAP next.

The SHAP Library

Now let’s take a look at the open source SHAP library, which is a powerful tool for working with Shapley values and other similar measures.

SHAP, which is short for SHapley Additive exPlanations, is a game-theoretic approach to explain the output of any ML model, which makes it model agnostic. It connects optimal credit allocation with local explanations using the classic Shapley values from game theory, and their related extensions, which have been the subject of several recent papers. Remember that Shapley created his initial theory in 1951, and more recently researchers have been extending his work.

SHAP assigns each feature an importance value for a particular prediction and includes some very useful extensions, many of which are based on this recent theoretical work. These include:

- TreeExplainer, a high-speed exact algorithm for tree ensembles
- DeepExplainer, a high-speed approximation algorithm for SHAP values in deep learning models
- GradientExplainer, which combines ideas from integrated gradients, SHAP, and SmoothGrad into a single expected value equation
- KernelExplainer, which uses a specially weighted local linear regression to estimate SHAP values for any model

SHAP also includes several plots to visualize the results, which helps you interpret the model.

You can visualize Shapley values as "forces," as shown in Figure 9-11. Each feature value is a force that either increases or decreases the prediction. The prediction starts from the baseline, which for Shapley values is the average of all predictions. In a force plot, each Shapley value is displayed as an arrow that pushes the prediction to increase or decrease. These forces meet at the prediction to balance each other out.

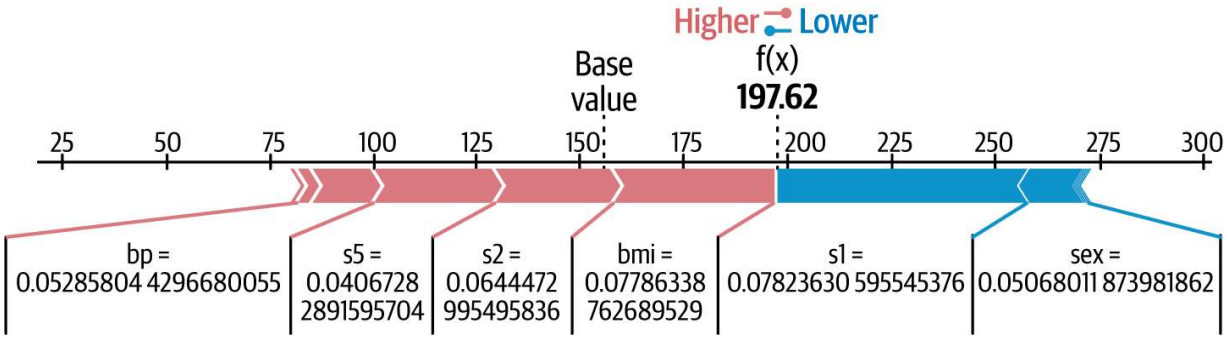


Figure 9-11. A SHAP force plot

A summary plot combines feature importance with feature effects. As shown in Figure 9-12, each point on the summary plot is a Shapley value for a feature and an instance. Overlapping points are jittered in the y-axis direction, so we get a sense of the distribution of the Shapley values per feature, and features are ordered according to their importance. So in Figure 9-12, we can quickly see that the two most important features are s1 (total serum cholesterol) and s5 (log of serum triglycerides level).

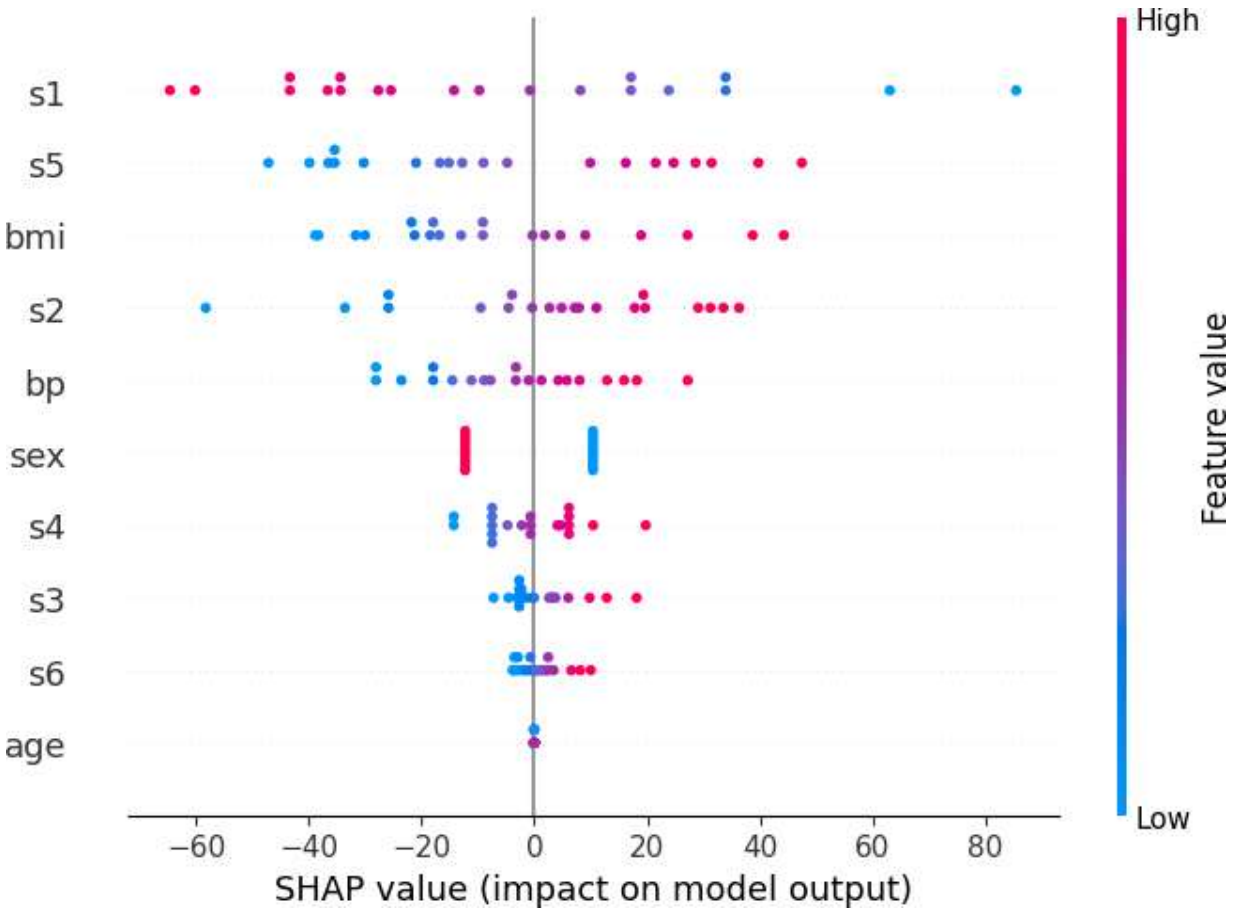


Figure 9-12. A SHAP summary plot[3]

As shown in Figure 9-13, in a SHAP dependence plot, a feature value is plotted on the x-axis and the SHAP value is plotted on the y-axis. From the plot in this example, you can see that the correlation between BMI and blood pressure (bp).

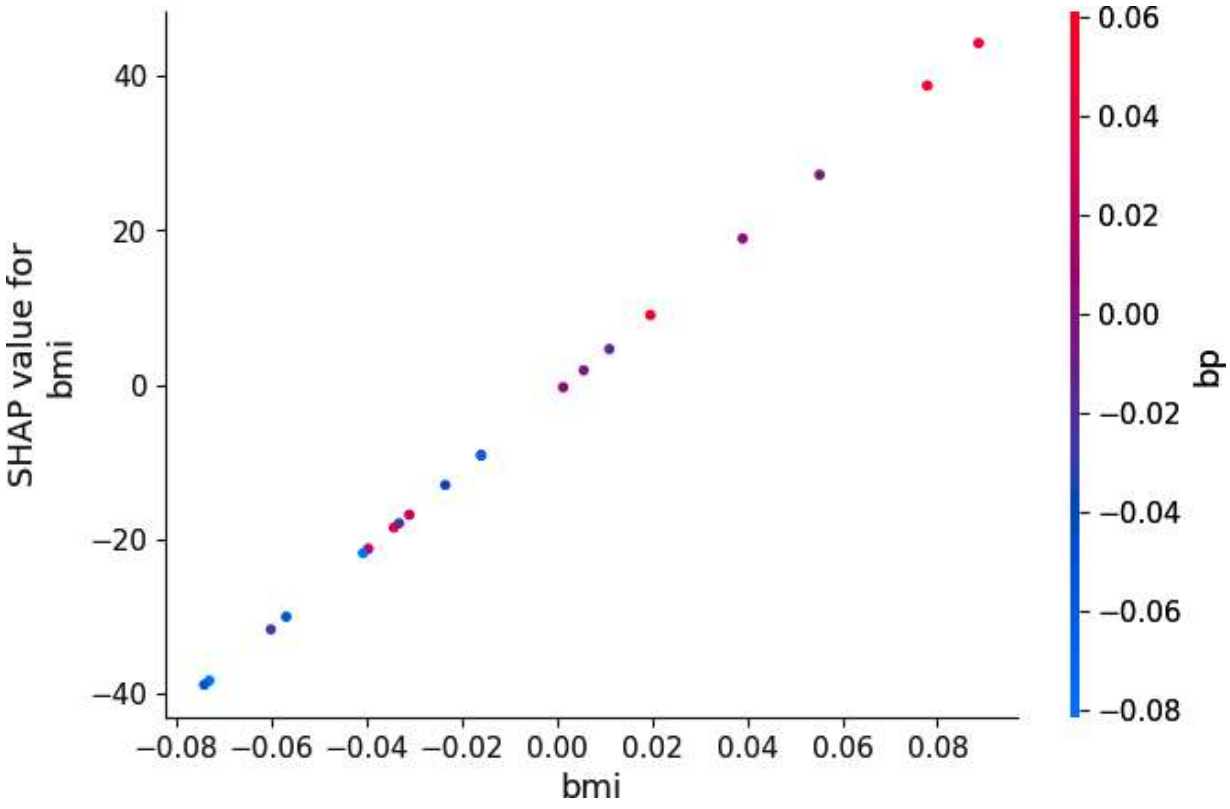


Figure 9-13. A SHAP dependence plot[4]

Testing Concept Activation Vectors

Understanding how deep learning models make decisions can be tricky. Their vast size, intricate workings, and, often hidden, internal processes make them difficult to interpret. Furthermore, systems like image classifiers often focus on minute details rather than broader, more understandable

concepts. To help decipher these complex models, Google researchers developed Concept Activation Vectors (CAVs). CAVs translate a neural network's inner workings into concepts that are easily grasped by humans. A method called Testing CAVs (TCAV) is used to evaluate these interpretations and is a key component of the LIT toolkit, which is detailed in Chapter 8.

We can define broader, more relatable concepts by using sets of example input data that are relevant to the model we're examining. For instance, to define the concept *curly* for an image model, we could use a collection of images depicting curly hairstyles and textures. Note that these examples don't have to be part of the original training data; users can provide new data to define concepts. Using examples like this has proven to be an effective way for for both experts and nonexperts to interact with and understand models.

CAVs allow us to arrange examples, such as images, based on their connection to a specific concept. This visual confirmation helps ensure that the CAVs accurately represent the intended concept. Since a CAV represents the direction of a concept within the model's internal representation, we can calculate the cosine similarity between a set of images and the CAV to sort them accordingly. It's important to note that the images being sorted are not used in training the CAV. Figure 9-14 illustrates this with two concepts—CEO and Model Females—showing how images are sorted based on their similarity to each concept.

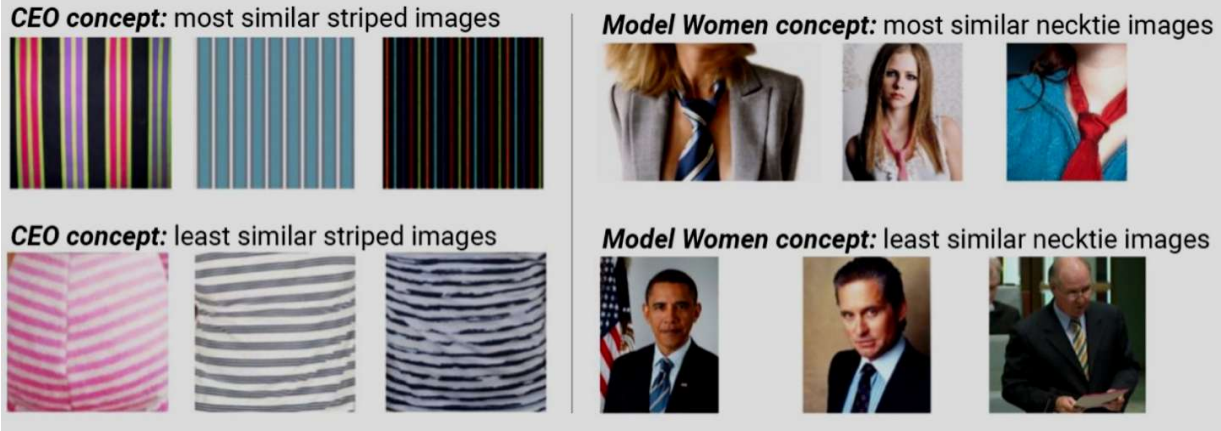


Figure 9-14. Two concepts of interest, and images sorted by similarity to each concept (source: Interpretability Beyond Feature Attribution, Been Kim et al., ICML, 2018)

On the left are sorted images of stripes, with respect to a CAV learned from a more abstract concept, “CEO” (collected from ImageNet). The top three images are the most similar to the CEO concept and look like pinstripes, which may relate to the ties or suits a CEO may wear, which provides confirmation of the idea that CEOs are more likely to wear pinstripes than horizontal stripes.

On the right are sorted images of neckties, with respect to a “Model Females” CAV. The top three images are the most similar to the concept of female models, but the bottom three images show males in neckties. This also suggests that CAVs can be used as a standalone similarity sorter to sort images to reveal any biases in the example images from which the CAV is learned.

AI Explanations

Cloud-based tools and services can also be very valuable for interpreting your model results. Let’s look at one of these now, Google’s AI Explanations service.

AI Explanations integrates feature attributions into Google’s AI Platform Prediction service. AI Explanations helps you understand your model’s outputs for classification and regression tasks. Whenever you request a prediction on AI Platform Prediction, AI Explanations tells you how much each feature in the data contributed to the predicted result. You can then use this information to verify that the model is behaving as expected, identify any bias in your model, and get ideas for ways to improve your model and your training data.

Feature attributions indicate how much each feature contributed to each given prediction. When you request predictions from your model normally using AI Platform Prediction, you only get the predictions. However, when you request explanations, you get both the predictions and the feature attribution information for those predictions. There are also visualizations provided to help you understand the feature attributions.

AI Explanations currently offers three methods of feature attribution. These include sampled Shapley, integrated gradients, and XRAI, but ultimately all of these methods are based on Shapley values. We’ve discussed Shapley

enough that we don’t need to go over it again, but let’s look at the two other methods, integrated gradients and XRAI.

Integrated gradients

Integrated gradients is a different way to generate feature attributions with the same axiomatic properties as Shapley values, based on using gradients, and is orders of magnitude more efficient than the original Shapley method when applied to deep networks. In the integrated gradients method, the gradient of the prediction output is calculated with respect to the features of the input, along an integral path. The gradients are calculated at different intervals, based on a scaling parameter that you can specify. For image data, imagine this scaling parameter as a “slider” that is scaling all pixels of the image to black. By saying the gradients are integrated, it means they are first averaged together, and then the element-wise product of the averaged gradients and the original input is calculated. Integrated gradients is one of the techniques included in LIT (see Chapter 8).

XRAI

The eXplanation with Ranked Area Integrals (XRAI) method is specifically focused on image classification. The XRAI method extends the integrated gradients method with additional steps to determine which regions of the image contribute the most to a given prediction.

XRAI performs pixel-level attribution for the input image, using the integrated gradients method. Independently of pixel-level attribution, XRAI also oversegments the image to create a patchwork of small regions. XRAI aggregates the pixel-level attribution within each segment to determine the attribution density of that segment, and then it ranks each segment, ordering them from most to least positive. This determines which areas of the image are the most salient or contribute most strongly to a given prediction.

Example: Exploring Model Sensitivity with SHAP

Production ML applications require in-depth investigations into the model's sensitivities to avoid any bad surprises for the model's end users. As we discussed in this chapter, SHAP is a great tool for investigating any ML model, regardless of the framework.

SHAP supports models consuming tabular, text, or image data. To get started, you need to `pip install` SHAP as follows:

```
$ pip install shap
```

Once you have installed SHAP, you can use it in a number of ways. Here, we are demonstrating two of the most common use cases.

Regression Models

Let's say you have a regression model that uses tabular input features and predicts a value between 0 and 1. You can investigate the sensitivity with SHAP as follows.

Let's start with an example model. Here, we are training a linear regression model to predict the likelihood of diabetes:

```
import shap
from sklearn import linear_model

# Load the diabetes dataset
X, y = shap.datasets.diabetes(n_points=1000)

# Split the data into training/testing sets
diabetes_X_train = X[:-20]
diabetes_X_test = X[-20:]

# Split the targets into training/testing sets
diabetes_y_train = y[:-20]
diabetes_y_test = y[-20:]

# Create linear regression object
regr = linear_model.LinearRegression()
```

```
# Train the model using the training sets
regr.fit(diabetes_X_train, diabetes_y_train)
```

Once the regression model is trained, we can use SHAP to test the model for its sensitivity. First, let's create a SHAP explainer object. The object unifies the interfaces of the SHAP library and assists in the generation of explanation plots:

```
explainer = shap.Explainer(regr, diabetes_X_trai
```

The `shap_values` can be generated by calling the explainer object. The `shap_values` are the sensitivity representation of a specific dataset, in our case, the test set:

```
shap_values = explainer(diabetes_X_test)
```

We can visualize the generated sensitivity explanations as a waterfall plot by calling `shap.plots.waterfall`:

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

The waterfall plot in Figure 9-15 shows nicely which feature has the highest impact on the sensitivity for a given input example.

The example showed the sensitivity testing for a simple regression model. In the following section, we are expanding the example to check for the importance of specific word tokens in text.

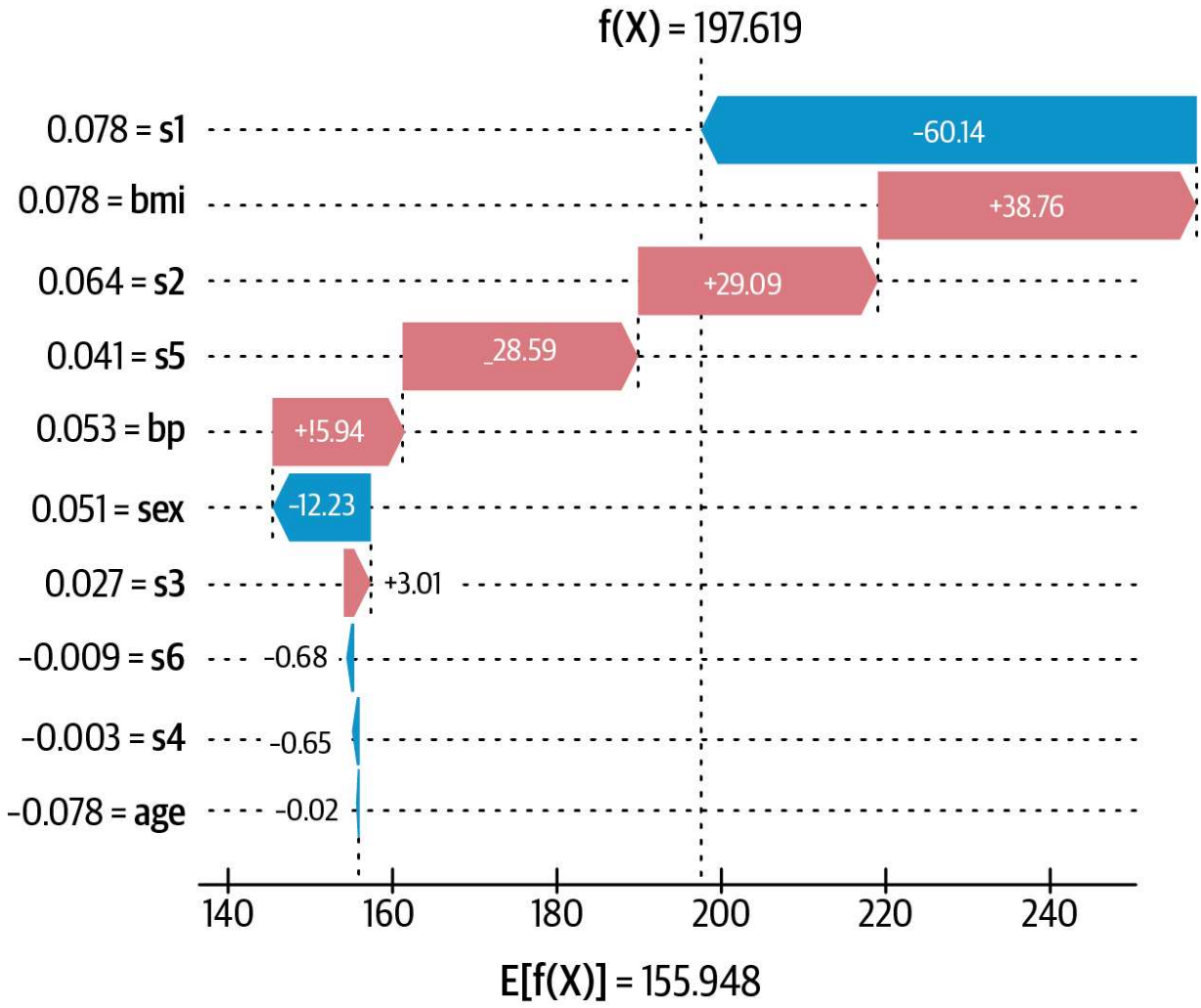


Figure 9-15. SHAP waterfall plot for a single sample and a regression model

Natural Language Processing Models

Measuring the influence of specific words or tokens in text can be done very similarly as shown in the previous example, but we need to tokenize

each text sample.

Like in our previous example, let's define our model, train it, or load a trained model. In our case, we use a pretrained GPT-2 model, but you can use any natural language processing (NLP) model. Load the model and the tokenizer:

```
from transformers import AutoModelForCausalLM, Au

tokenizer = AutoTokenizer.from_pretrained("gpt2",
model = AutoModelForCausalLM.from_pretrained("gpt
```

NOTE

It is important that the loaded tokenizer and the token IDs match the preprocessing setup used during the model training and deployment.

Now let's assemble the SHAP explainer using the model and the tokenizer. The model can also be replaced by a prediction wrapper function, which will produce the model outputs:

```
import shap
explainer = shap.Explainer(model, tokenizer)
```

With the explainer object now created, we can evaluate which token has the biggest influence on the test sentence by plotting the sensitivity using `shap.plots.text`. In our case, it showed that the terms “Machine” and “best” and the character “!” have the biggest influence:

```
shap_values = explainer(["Machine learning is the
shap.plots.text(shap_values)
```

Figure 9-16 shows the result.

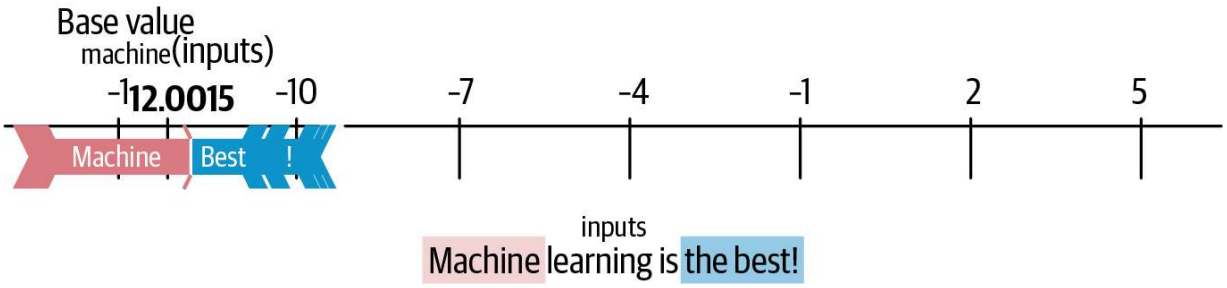


Figure 9-16. SHAP example with a deep learning NLP model

Conclusion

In this chapter, we introduced the importance of model interpretability. We also discussed several techniques for interpreting models to understand how they make predictions and to guide improvements to the models to reduce potential harms. This included a discussion of the differences between intrinsically interpretable model architectures such as tree-based models and lattice models, and other model architectures that must be interpreted