Explainability for Tabular Data

Much of the success of deep learning has focused on unstructured data like images, text, audio, and video; however, the vast majority of machine learning models in production are built with tabular data in mind. Think of all the data contained in relational databases and spreadsheets composed of numeric and categorical feature sets. These are examples of structured data and make up the vast majority of real-world AI use cases. In this chapter, we'll examine explainability techniques that are most often used when working with tabular data, like Shapley values, permutation feature importance, tree interpreters, and various versions of partial dependence plots.

Permutation Feature Importance

Here's what you need to know about permutation feature importance:

- Once a model has been fit to the training data, the permutation importance for a single feature measures the decrease in a model score when that feature value is randomly shuffled.
- By shuffling the values of a given feature, you destroy the model's ability to make meaningful predictions using that feature. If the model predictions suffer and the model score is much worse, then the information provided by that feature must have been important to the model when making predictions. On the other hand, if the change in the model score is negligible then that feature isn't as important.

Pros	Cons
• It's easy to implement. Scikit-learn provides a nice, easy-to-use library for computing permutation feature importance. • The result is intuitive. The method of permutation feature importance is easy to explain and understand. • Permutation-based methods work equally well for mixed modes of tabular data (e.g., numeric and categorical features).	• Results can be misleading when features are highly correlated. This method has an underlying assumption that features are independent. • The results of permutation importances do not reflect the intrinsic predictive value of a feature by itself, but instead reflect how important a feature is for a particular model. • The computation of permutation importances is heavily dependent on the feature shuffling, and different shuffles may produce different results. Multiple runs may be necessary to get a more accurate picture.

Permutation feature importance is a perturbation-based feature attribution technique commonly used for tabular datasets (see Chapter 2 for discussion on feature attributions and perturbation techniques). The common pattern is that the model features are perturbed or modified in some way and then predictions are made on these new examples. Using the model predictions from this collection of new, perturbed examples, you can then determine the impact each feature has on predictions by seeing how the model's predictions vary.

For example, for permutation feature importance, once a model has been fit to the training data, the importance of a feature is determined by measuring the prediction error after permuting the values of a given feature in the validation set. By shuffling the values of a given feature, you destroy the model's ability to make meaningful predictions using that feature.

When you measure the resulting change in the validation error, if the decrease is negligible, then the information provided by that feature wasn't very important or useful in determining the model predictions. That is to say, your model can still do a pretty good job without that feature. If, on the other hand, the model predictions suffer and the validation error is much worse, then the information provided by that feature must have been important to the model when making predictions.

In summary, the permutation feature importance of a model is the decrease in a model score when a single feature value is randomly shuffled. It's particularly helpful for nonlinear or hard-to-interpret models since it only relies on a fitted estimator and the model score could be any evaluation metric that makes sense, such as the mean square error or R^2 for a regression task or accuracy for a classification model. As such, using the taxonomy discussed in Chapter 2, permutation feature importance is a post hoc, global, model-agnostic explainability technique.

Permutation Feature Importance from Scratch

Let's implement permutation feature importance in an example. We'll use the California Housing dataset (*https://oreil.ly/Xdgx3*).[1] This dataset was collected from the 1990 US census and each row represents one census block group (each block group typically has a population of 600 to 3,000 people). Each example contains eight feature attributes like the average number of bedrooms per home or the median income of block residents. The target label for this dataset is `MedianHouseVal`, the median value of houses within each block expressed in hundreds of thousands of dollars. A small sample of this dataset is shown in Table 3-1, and Table 3-2 gives a description of each of the features and the label.

Table 3-1. The California Housing dataset contains feature attributes of houses at different locations around California suburbs.

MedInc	HouseAge	AveRooms	AveBedrms	Population	AveOccup	Latitude	Longitude	MedianHouseVal
8.3252	41	6.984127	1.02381	322	2.555556	37.88	−122.23	4.526
8.3014	21	6.238137	0.97188	2,401	2.109842	37.86	−122.22	3.585
7.2574	52	8.288136	1.073446	496	2.80226	37.85	−122.24	3.521
5.6431	52	5.817352	1.073059	558	2.547945	37.85	−122.25	3.413
3.8462	52	6.281853	1.081081	565	2.181467	37.85	−122.25	3.422

Table 3-2. The label variable `MedianHouseVal` represents the median values of the houses within each block, measured in hundreds of thousands of dollars ($100,000).

Feature name	Feature
MedInc	Median income in block group (in $10,000)
HouseAge	Median house age in block group in years
AveRooms	Average number of rooms per household
AveBedrms	Average number of bedrooms per household
Population	Block group population
AveOccup	Average number of household members
Latitude	Block group latitude
Longitude	Block group longitude
MedianHouseVal	Median house value (in $100,00)—target

1 R. Kelley Pace and Ronald Barry, "Sparse Spatial Autoregressions," *Statistics and Probability Letters* 33 (1997): 291–97.

To start, we'll build and train a simple neural network in TensorFlow (the full code for this example can be found in the GitHub repository (*https://oreil.ly/LIfWN*) for this book):

```
model = tf.keras.Sequential([
   tf.keras.layers.Dense(40, activation=tf.nn.relu,
                         input_shape=[len(X_train[0])]),
                         tf.keras.layers.Dense(20, activation=tf.nn.relu),
                         tf.keras.layers.Dense(1)
])

model.compile(optimizer='adam', loss='mse', metrics=['mse'])
model.fit(X_train, y_train, epochs=30)
```

Once the model is trained, we measure the root mean squared error on the holdout test set. For this model the test error is 0.51, which represents an error of about $51,000. To compute the importance score for a given feature, we'll shuffle those feature values in the holdout set and see how much it affects our model performance. For important features, we'd expect the difference in model error to be large, whereas for less important features, shuffling the feature values won't have as much of an effect.

Suppose we want to measure the importance of median income (`MedInc`). First, we permute the feature values for that feature, then recompute the test error, and compare with the test error 0.51 we measured before (without shuffling `MedInc`):

```
model_rmse = 0.51
df_perturb['MedInc'] = df_perturb['MedInc'].sample(frac=1.0).values
preds = model.predict(df_perturb.values)
feature_rmse = np.sqrt(compute_mse(preds, y_test))
permutation_feature_importance = model_rmse - feature_rmse
```

This indicates that the importance score for `MedInc` (median household income) is about –0.44. Repeating this process for each feature in our dataset, we get a table of importance scores for each feature, as shown in Table 3-3.

It's important to note that the scores computed in Table 3-3 are only relative, meaning they can only be interpreted in relation to one another. On their own, the importance score for a single feature doesn't mean much. Even though the permutation importance score is based on the mean squared error loss, it doesn't directly translate to the home's value. For example, we can't say that on average $44,000 of a home's value is determined by the neighborhood's median income. However, we can use these numeric scores to indicate which feature has more or less of an impact on a model's prediction values in relation to each other.

Table 3-3. The importance score of each feature measures how much the model error is affected when those feature values are shuffled and predictions are made against the validation set.

Feature	Score
Latitude	–1.272
Longitude	–1.162
AveRooms	–0.852
AveBedrms	–0.731
MedInc	–0.443
AveOccup	–0.295
HouseAge	–0.079
Population	–0.036

According to Table 3-3, the latitude and longitude are the most important features for determining house value because they have the largest score in absolute value while the median house age and block population are the least important.

> This method of permutation feature importance relies on independence of the different features. If two features, say `feature_A` and `feature_B`, are highly correlated, then when we shuffle the values of `feature_A`, the model may still be able to do well since `feature_B` essentially contains the same information. Similarly, for shuffling the values of `feature_B` when we compute its feature importance. So, although these features may individually be very important for the model to make accurate predictions, you wouldn't be able to catch that relationship using permutation feature importance because the two features are highly correlated.
>
> In the example with the California Housing dataset, we could (and should!) examine this kind of multicollinearity in our dataset during our routine data exploration, but this can be particularly tricky to catch when you are working with a large number of features.

Permutation Feature Importance in scikit-learn

In practice, you'd want to shuffle and measure more than once since each shuffling introduces some randomness, which could affect the final result. Here, the shuffling is a kind of sampling without replacement; we're truly just reordering the values for a given feature. However, different shuffles may produce different results. By the law of large numbers (LLN), with repeated samplings we'll ultimately approach the true value of the feature importances, so multiple runs are necessary to get a more accurate picture. When building models in scikit-learn, there is a nice implementation of permutation feature importance that can be used for any fitted estimator.

Let's take another look at the California Housing dataset and fit a neural network in scikit-learn. The features of this dataset include numeric and categorical features, so we'll use the `ColumnTransformer` and `Pipeline` in scikit-learn to create and train our model. The notebook on permutation feature importance (*https://oreil.ly/LIfWN*) in the GitHub repository for this book contains the full code for this example.

Once the model is trained, we can use the `permutation_importance` function in scikit-learn to calculate the feature importance by passing the fitted estimator, the data and targets on which the permutation importance will be computed, the number of times to permute features and calculate the scores, and the scoring metric to use. We'll take `n_repeats` to be 30 to get a reasonable sample size, and we'll use R^2 as the evaluation metric when scoring the model. The following code block shows how to implement this in the scikit-learn library:

```
r = permutation_importance(mlp_pipeline, X_test, y_test,
                           n_repeats=30,
                           scoring=['r2'],
                           random_state=0)
```

How Many Repeats Is Enough?

Determining the right number of repeats for feature permutation may take a bit of experimentation. In the scikit-learn implementation, the default for the number of times to permute a feature is `n_repeats=5`. However, you might find that for your dataset, you need more. The larger number of repeats you choose, the more stable the result will be. However, this also increases the computation cost, which can become quite large with larger datasets. It is worth exploring and experimenting a bit with values ranging from 5 to 30 to get an idea of what value works best for your use case and dataset.

In this example, the permutation feature importance is calculated by first evaluating the model baseline using R^2 on the dataset on `X_test`. Then for each feature in the dataset, that feature column is permuted and the R^2 score is computed. The permutation importance is then the difference between the baseline R^2 score and the R^2 score using the permuted features. This process is repeated again and again; in our case, 30 times. The mean and standard deviation of the importance values are returned, so we can inspect the distribution of the importance scores for each feature:

```
for i in r['r2'].importances_mean.argsort()[::-1]:
    print(f'{cal_features[i]:<8}'
          f"{r['r2'].importances_mean[i]:.3f}"
          f" +/- {r['r2'].importances_std[i]:.3f}")
```

which returns:

```
Latitude: 1.842 +/- 0.033
Longitude: 1.758 +/- 0.024
MedInc: 0.673 +/- 0.014
HouseAge: 0.045 +/- 0.003
AveOccup: 0.003 +/- 0.001
AveBedrms: 0.003 +/- 0.001
Population: 0.002 +/- 0.001
AveRooms: 0.002 +/- 0.001
```

Perhaps not surprisingly we (re)learn a basic rule of real estate: house values depend most on location.

Using the Test Set or the Training Set

Permutation feature importances are a global explainability technique and rely on a batch of data to compute feature importances. In the example here, we used the test set `X_test` but we could easily have substituted the training set `X_train` instead. By using the test set, it's possible to pinpoint those features that have the greatest importance when generalizing to new data, which is ultimately what we care about in practice. Using the training set to determine importances might cause you to believe that those features are important for unseen data as well at the time of inference. However, it is possible the model was overfitting and those features aren't actually that important. In fact, any features that are deemed important for the training set but not on the test set are precisely those features that are most likely to cause the model to suffer at generalization.

As noted in the documentation for scikit-learn's `permutation_importance` function, the permutation importance of a feature does not reflect the intrinsic predictive value of a feature by itself but how important this feature is for a particular model. Even though a feature may have low importance for a bad model, that same feature could have high importance for a good model. Thus, you should always evaluate a model using a holdout set *before* computing any feature importances.

Shapley Values

Here's what you need to know about Shapley values:

- Derived from game theory to explain how a feature influences a predicted value
- Commonly implemented explainability technique available in several open source packages as well as cloud-based options

Pros	Cons
• Shapley values can be used for individual predictions, cohorts, and to globally explain the model. • Attributions sum to the total predicted value. • The values provide an intuitive understanding for stakeholders.	• Feature influence is often conflated with causality by practitioners, end users, and stakeholders. • These values are computationally intensive and difficult to use with models with more than ~100 features. • Choosing a good baseline can be difficult.

In Chapter 2, we discussed how Shapley values are computed and, broadly, how they are used in machine learning. In the past several years, Shapley values have become one of the most popular explainability methods for tabular datasets and models. They offer a lucrative way to say "this feature mattered, and this one didn't" with relatively little effort up front given several OSS and cloud-based solutions for computing them. However, this ease of use also hides some of the more difficult implications of incorrectly computing Shapley values.

SHAP (SHapley Additive exPlanations)

The terms *Shapley values* and *SHAP values* are often used interchangeably. However, this is not technically correct. Shapley values represent the theory, and SHAP values are a specific implementation for calculating Shapley values. You may also see reference to *sampled Shapley*, which is an approximation method of an exact Shapley value. Because SHAP values adhere to the four axioms of Shapley values (see the sidebar "The Axioms of Fairness"), it is pragmatically okay to use either term. For continuity, we will continue to use the term Shapley values here to refer to the SHAP values.

The Axioms of Fairness

Shapley values are considered a "favorable and fair" attribution method since they satisfy the four axioms of fairness: efficiency, symmetry, null player, and additivity/linearity:

- The *efficiency* axiom states that the total attribution is distributed in a lossless manner among all the model features. That is to say, the sum of all feature attributions determined when applying Shapley valuesequals the total attribution of the model.
- The *symmetry* axiom says that if two features play equal roles, then their Shapley values must be equal. This means that only the role of a play matters; the labels or specific names are irrelevant. Changing feature names won't change the Shapley value.
- The *null player* axiom says that if the marginal importance of a feature is always zero, then the baseline value of that feature is zero.

- The axiom of *additivity/linearity* ensures consistency among feature attributions with respect to linear combinations of models.

These four axioms provide a unique characterization of the Shapley value. Let's discuss what these axioms state in the context of machine learning explainability. See also the discussion on the axiomatic approach to evaluating explainability techniques in Chapter 6.

SHAP (*https://oreil.ly/5NdmT*) is the most popular OSS implementation of Shapley values. It builds upon the core idea of Shapley values and extends it to support multiple ML frameworks, along with providing a variety of useful visualization methods. SHAP has several implementations for different types of ML models and architectures, as shown in Table 3-4.

Table 3-4. An overview of different implementations within the SHAP library and their uses

SHAP class	ML model /architectures	Notes
`TreeExplainer`	Tree models, XGBoost, scikit-learn	High performance when computing Shapley values
`DeepExplainer`	DNNs, TensorFlow	Based on DeepLIFT, approximate Shapley values, difficult to configure
`GradientExplainer`	Differentiable models, TensorFlow	Slower than DeepExplainers, also approximates Shapley values
`LinearExplainer`	Linear regression	Computes the exact Shapley value; i.e., weights multiplied by feature values
`KernelExplainer`	Model agnostic	More difficult to configure, and slowest SHAP Explainer in terms of computation time

SHAP can also be used for other data modalities, such as images and text. See "Explaining Tree-Based Models" on page 63, which is where you'll find more information on how to use `TreeExplainer`.

Open Source Implementations of Shapley Values

There are other open source implementations of Shapley values that you might find useful. Captum is an OSS framework from Meta for using Explainable AI with PyTorch models. Captum offers a wide range of explainability algorithms, with a flexible approach that makes it easy to swap between different feature attribution techniques by changing only a few lines of code.

Captum supports many feature attribution techniques including sampled Shapley, DeepLIFT, DeepLiftSHAP (SHAP based on DeepLIFT, similar to SHAP's `Deep Explainer`), feature ablation, feature permutation, Integrated Gradients, SmoothGrad, KernelSHAP, and GradientSHAP. For differences between these implementations, and their limitations, see Captum's excellent Algorithm Comparison Matrix (*https://oreil.ly/qs0Gq*).

Captum does not have any built-in capabilities for visualizing feature attributions for structured data. Instead, attributions are returned as a tensor (or multiple tensors if more than one data sample was provided) of feature attribution (or in this case, Shapley values) that maps to the original organization of the inputs for the model (e.g., the first value in the tensor corresponds to the first feature).

Next, we will see how to use the SHAP library to compute Shapley values for a model trained on the California Housing dataset again. The following code block creates and trains an XGBoost model, then creates an `Explainer` object to compute feature attributions using SHAP. Providing the `Explainer` with an individual prediction returns the calculated Shapley values for that example:

```
model = xgboost.XGBRegressor(objective='reg:squarederror',
                             n_estimators=500)
model.fit(X_train, y_train)

explainer = shap.Explainer(model)
shap_values = explainer(X_test)
```

This approach of creating an explainer by calling `shap.Explainer(model)` works regardless of the model architecture or ML framework used. See the SHAP notebook (*https://oreil.ly/Pft5B*) in the GitHub repository for this book for the full code for this example. There is also an example there using a TensorFlow model.

The `shap_values` is an enumerable corresponding to all the data samples given to `explainer` to predict. In the previous code snippet, we're passing all the examples from the test set `X_test` to `explainer`. The following code block takes an individual value (i.e., the first value) in `shap_values` and extracts the model's prediction for that example, the baseline value, and Shapley values for each feature in the dataset:

```
first_row_shap = shap_values[0]
shapley_values = first_row_shap.values

print(f'Predicted Value: {first_row_shap.base_values + sum(shapley_values)}\n')
print(f'Baseline: {first_row_shap.base_values}\n')

print('Shapley values for features:')
for i, shapley_value in enumerate(shapley_values):
    print(f'{shap_values.feature_names[i]}: {shapley_value}')

print('\nMost to least influential features:')
most_influential_ordering = np.argsort(-np.abs(shapley_values))
for i in range(len(shapley_values)):
    print(f'{shap_values.feature_names[most_influential_ordering[i]]}:
          {shapley_values[most_influential_ordering[i]]}')
```

For the California Housing dataset, this code block returns:

```
Predicted Value: 1.889114398509264

Baseline: 2.063443660736084

Shapley values for features:
MedInc: 0.370290607213974
HouseAge: 0.03518122434616089
AveRooms: 0.15876026451587677
AveBedrms: 0.036185089498758316
Population: -0.06795859336853027
AveOccup: -0.10934144258499146
Latitude: -1.0922733545303345
Longitude: 0.49482694268226624

Most to least influential features:
Latitude: -1.0922733545303345
Longitude: 0.49482694268226624
MedInc: 0.370290607213974
AveRooms: 0.15876026451587677
AveOccup: -0.10934144258499146
Population: -0.06795859336853027
AveBedrms: 0.036185089498758316
HouseAge: 0.03518122434616089
```

Let's unpack this a bit and see how we should interpret this output. For this example instance, i.e., the first example of the test set, the feature values are:

```
MedInc: 3.5625
HouseAge: 43.0
AveRooms: 5.64741641337386
AveBedrms: 1.0486322188449848
Population: 1054.0
AveOccup: 3.2036474164133737
Latitude: 34.11
Longitude: -118.01
```

and the model predicts 1.889, or a median house value of $1,889,000. We also see that the baseline prediction for our model is 2.06, or median house value $2,060,000. The baseline prediction is just the expected value of the model output; that is, the average of the model predictions on all values in the test set. One of the fundamental properties of Shapley values is that the SHAP values of all the input features will sum up to the difference between the baseline model output and the model prediction for that example. This is also demonstrated visually using the waterfall plot in the next section. In fact, this is exactly what we see in the preceding example. If we compute `np.mean(xgb_reg.predict(X_test))`, we see that the average predicted value for the model is 2.06. And summing the baseline value with the Shapley values for each feature in this example gives exactly the model prediction 1.889. The rest of the output lists the individual Shapley values for each feature in this example instance and simply arranges them in increasing order.

Visualizing Local Feature Attributions

To visualize these feature attributions, we can use one of SHAP's many plotting methods. The waterfall plot, shown in Figure 3-1, is one of the most popular. Note that the SHAP values of all the input features sum up to the difference between the baseline model output, in this case 2.063, and the model's prediction for that example, here 1.889:

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

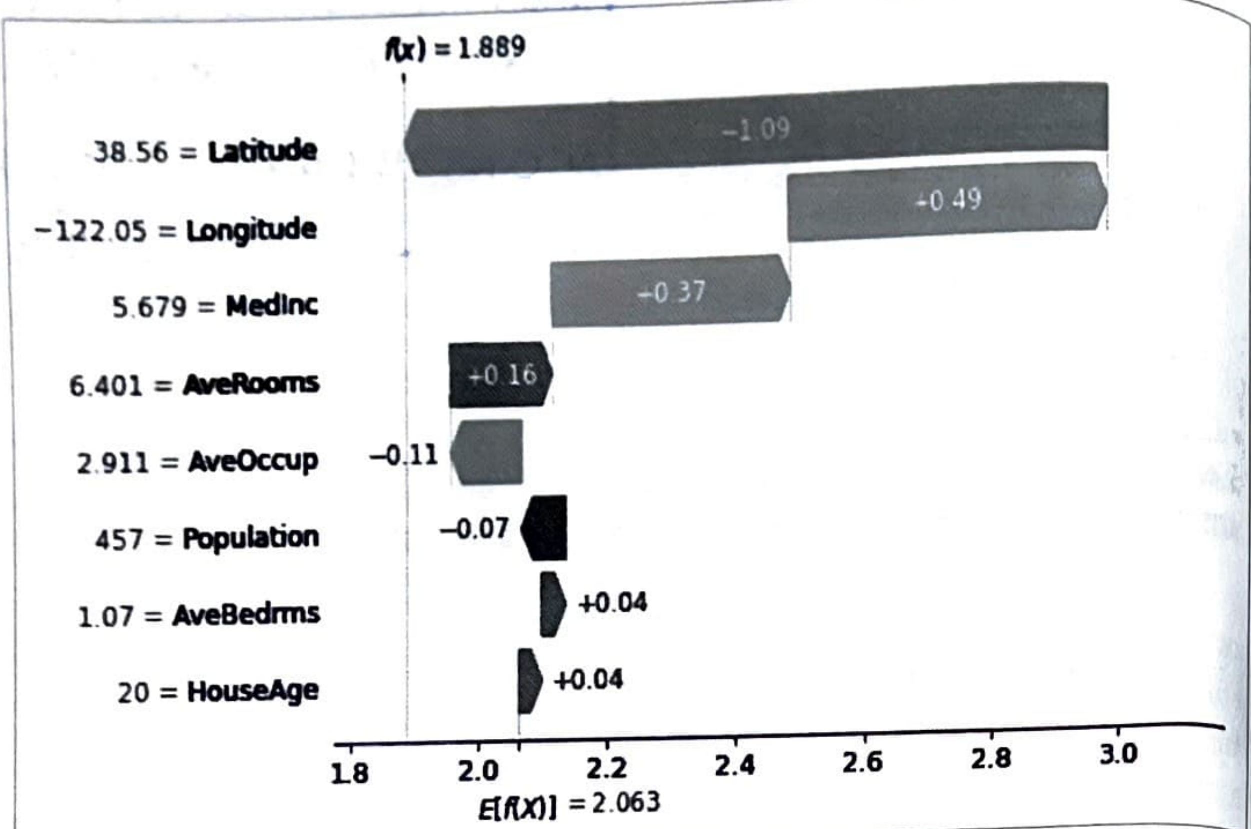

Figure 3-1. Feature attributions for a single prediction in the California Housing dataset using SHAP's waterfall visualization.

The waterfall visualization contains many pieces of information to give context to the feature attributions. The y-axis lists all features, with the feature's value to the left of the feature label (i.e., `AveRooms` = 6.401 for this example). Each feature attribution is represented as a row in the waterfall, color coded with an arrow for whether the feature contributed positively or negatively toward the final prediction value. Red bars pointing right indicate a positive contribution, while blue bars pointing left indicate a negative contribution. The rows in the visualization are ordered from top to bottom by the greatest to smallest contribution to the predicted score.

As you can imagine, if your model has more than eight features this visualization can become quite unwieldy and hard to read, and likely the less important features have very small attribution values. Shapley feature attributions are *additive*, meaning we can sum them together to represent the overall contribution by a group of features. To make the plot more readable, SHAP aggregates the lowest influence features into a single bar (with the label "4 other features," for example).

> Remember, positive and negative feature attributions simply indicate how the feature contributed to the numerical value of the prediction, not more or less. The feature attribution's magnitude of influence is the absolute value of the attribution.

The visualization also helpfully displays the predicted value with a label of `f(x)=...` along with a gray line through all rows. Likewise, the baseline predicted value is shown along the x-axis with a label `E[f(X)] =...` (*E* is used to indicate this is the expected value of a prediction, although SHAP uses baseline scores internally).

There are two ways to read the waterfall chart. Starting at the top will tell you which features had the greatest influence in the model's prediction. Starting at the bottom and reading toward the top will demonstrate how each feature contribution moved from the baseline toward or away from the predicted value.

> Be careful not to infer causality from how Shapley values are displayed, and to caution your stakeholders to avoid making the same mistake. This is much easier to do than it may seem at first, and we even found ourselves making these associations while writing this book.
>
> Most solutions for displaying Shapley values will use bars to indicate the magnitude of the feature's influence, and stack those bars together to make the chart easier to understand. However, in making these visualizations easier to understand, it is also easy to fall prey to constructing a narrative of "Because of Feature A, Feature B's attribution is..." or "Feature A first contributed to the predicted score, then Feature B built on that..." However, Shapley values simply indicate relative influence by a feature, not a causal relation between features. See Chapter 7 for further discussion on displaying and understanding XAI.

A force plot, like the one in Figure 3-2, is similar to a waterfall plot but puts all the feature attributions displayed on a single axis. It's just a different visual representation of the same information but can be used in conjunction with a waterfall plot to illustrate how features push a prediction value toward (or away) from the baseline prediction. The code snippet that follows shows how to produce such a plot:

```
shap.initjs()
shap.plots.force(shap_values[0])
```

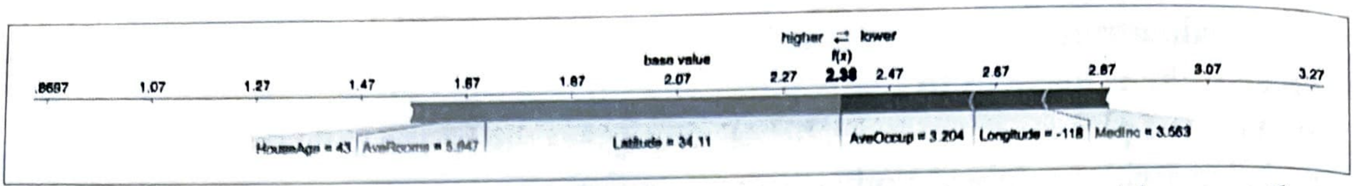

Figure 3-2. The SHAP force plot for an individual prediction showing positive contributions on the left, and negative contributions on the right.

The x-axis of these charts are the range of the values for the feature (with a histogram showing the distribution of values in light gray above the x-axis), while the left y-axis is the feature's Shapley value for each individual prediction of the dataset. SHAP also includes a second measure in these visualizations, which is based on what it determines is the feature with the most cross-interaction with the primary feature you visualized. The values of this secondary feature are shown using a heatmap, coloring each point in the chart according to the value for the secondary feature.

> Red and blue colors are used indiscriminately in SHAP's visualizations, and do not necessarily represent a feature's influence on a prediction. SHAP visualizations for individual predictions use red and blue to indicate a positive or negative Shapley value, but other visualizations in SHAP will use red and blue for other purposes, like to distinguish between two feature values. For example, in Figure 3-4, the red and blue colors are used to represent the value of the `AveOccup` feature across the entire dataset, while in Figure 3-1, the colors represent whether the SHAP value for a particular feature was positive or negative.

Visualizing Global Feature Attributions

There are other ways that SHAP can visualize Shapley values. In our opinion, one of the most useful for explanation analysis is the scatter plots. With the scatter plot, SHAP can also be used to generate visualizations that look at the impact of a feature's contribution as the value of that feature changes in the dataset.

To generate a scatter plot, we provide SHAP with the column of data we'd like to visualize. In the code snippet here, we'll look at the Shapley values for the median income:

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

Figure 3-3 shows the Shapley values for the feature MedInc in the plot on the left. We see an upward trend for the feature influence as the value of MedInc increases. This suggests that census blocks with higher median income are more likely to positively contribute to the prediction of the median house value away from the baseline, while smaller MedInc values influence the model more back toward the baseline prediction score. Compare this with the plot on the right in the figure. The Shapley values for HouseAge don't show such a recognizable trend. At the bottom of the plot is a histogram in light gray showing the distribution of data values.

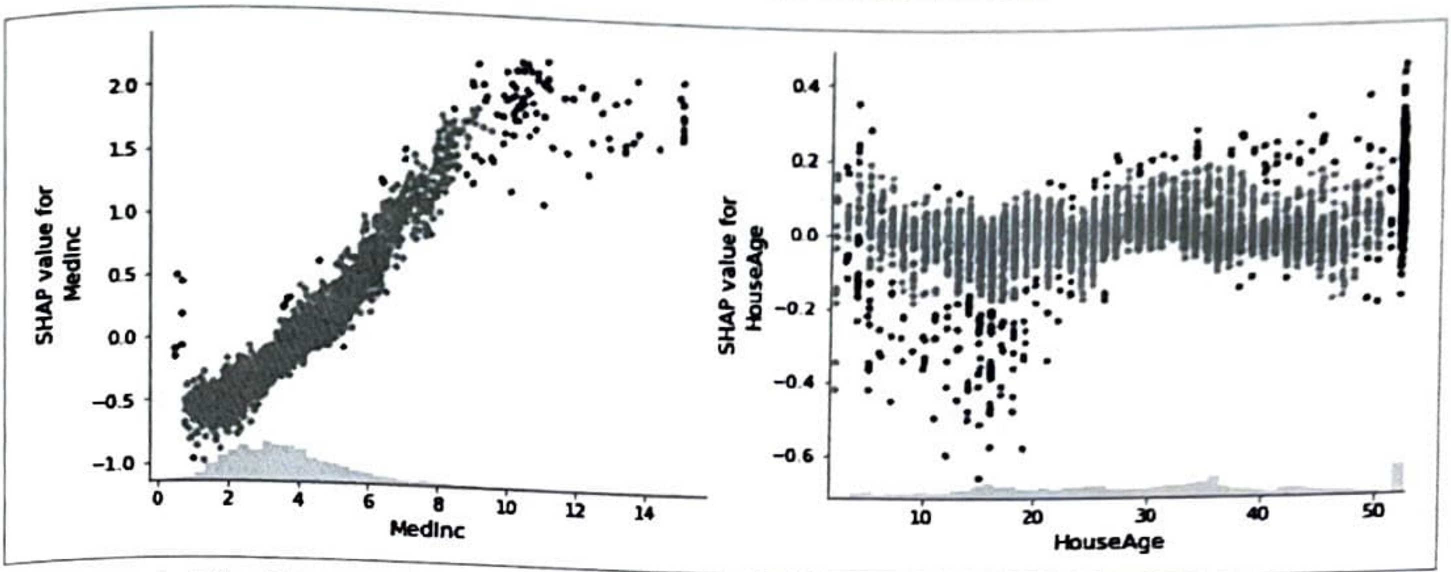

Figure 3-3. The SHAP scatter plot displays all Shapley values against a given feature value. On the left is the feature MedInc *(the median income of families in the census block) while on the right is the scatter plot for* HouseAge *(the median age of houses in the census block).*

Looking at the scatter plot for HouseAge in Figure 3-3, notice that there is quite a large vertical dispersion of points for any given value of the median house age. This indicates that there must be a strong nonlinear effect between HouseAge and the other model features. Otherwise, we'd see a trend more similar to that for MedInc.

By passing the entire Explanation object to the scatter plot, we can show which feature is most driving that interaction effect. This is demonstrated in the following code snippet, and the resulting graphic is shown in Figure 3-4. SHAP picks out the feature that has the strongest interaction with HouseAge, in this case, for this model, it is AveOccup (the average number of households in the census block), and plots the two features together. The points represent HouseAge and the color of the points indicate Shapley values for AveOccup. If there is an interaction between the two features, it should show up visually and may indicate that there is collinearity between these features in our dataset that influence the model:

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

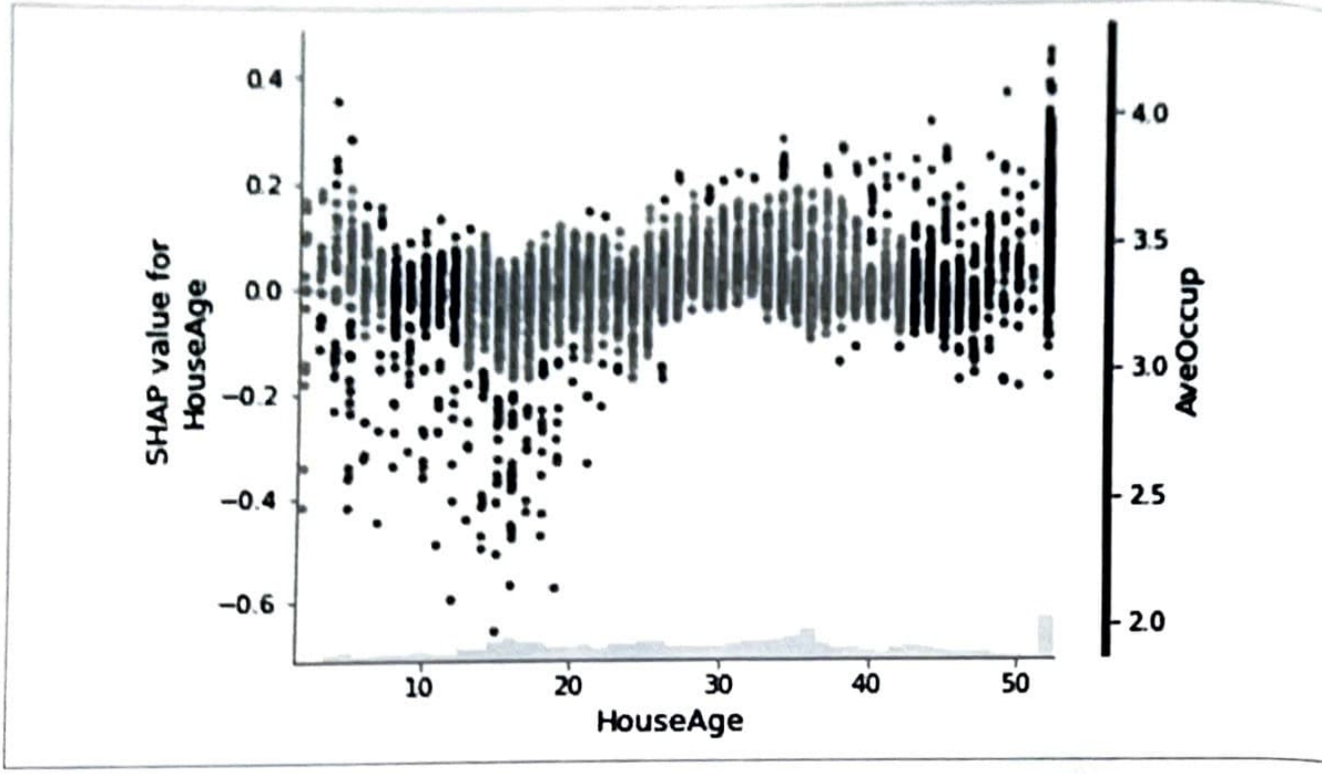

Figure 3-4. SHAP scatter plot for all Shapley values of `HouseAge` and `AveOccup` in the California Housing dataset. (Print readers can see the color image at https://oreil.ly/xai-fig-3-4.)

The scatter plot shown in Figure 3-5 is for the block population feature, denoted by `Population` in the dataset. For the most part, the Shapley values for this feature hover between 0.2 and negative 0.2.

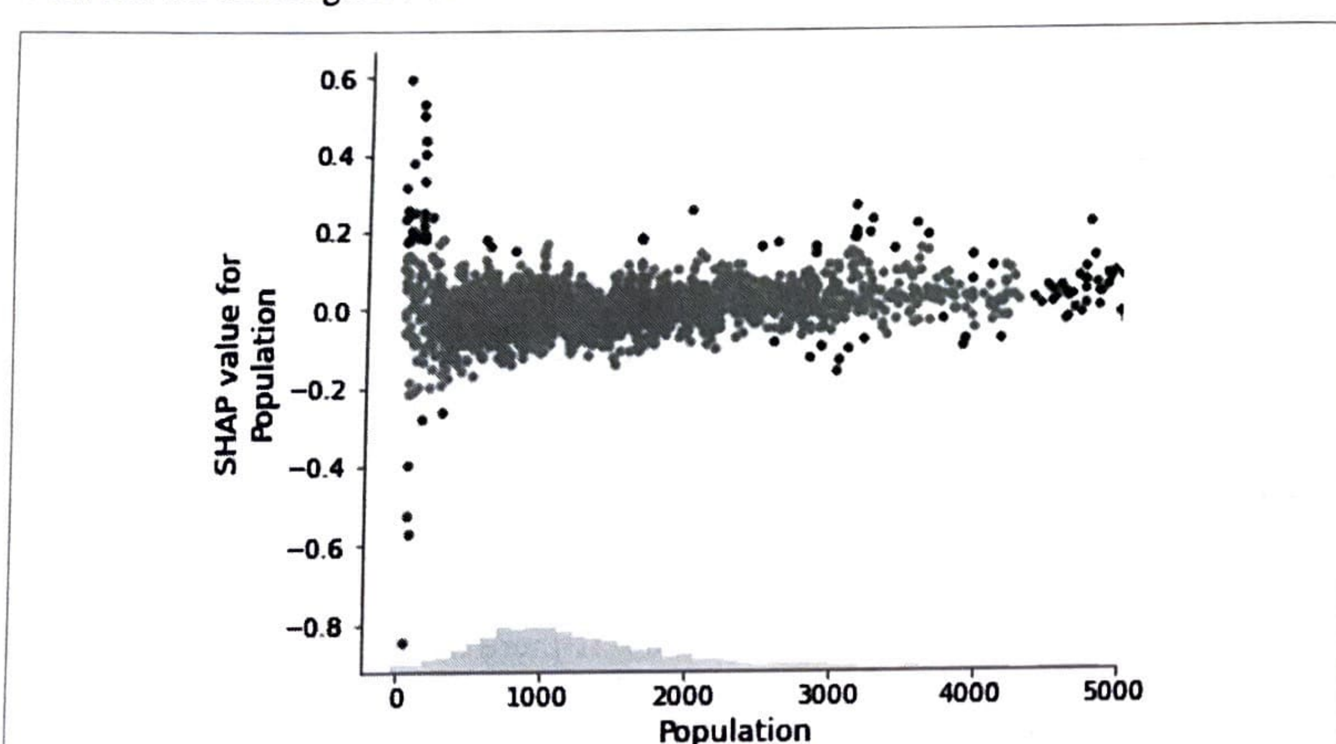

Figure 3-5. SHAP scatter plot for all Shapley values of the `Population` feature in the California Housing dataset.

However, there are two interesting aspects of the model that are revealed on closer examination:

- `Population` values less than approximately 250 can have a much larger influence on the model's prediction.
- There is a large spread in the Shapley values for a `Population` value near zero.

For the first observation, we could hypothesize that the model has learned an edge case where a very low population within a census block is very informative to the prediction.

Interpreting SHAP Scatter Plots

Recall that Shapley values are additive, so we need to view them in the context of Shapley values for other features to understand their relative contribution to the overall prediction. For example, in the scatter plot in Figure 3-5, it could simply be that the overall predicted value is much larger, or even so large that the relative influence of `Population` at the lower values is even less!

Our second observation reveals an area where the `Population` feature is likely not reliably contributing to the model, or it could be that the overall predicted score varies quite a bit due to other features. A useful technique is to normalize the Shapley values for all features in a prediction to the range of –1 to 1 (relative to the prediction score), which allows one to reliably understand an individual feature's influence without having to constantly reference other features. An important caveat for normalization is that your Shapley values may not cleanly add up to the overall predicted score due to the use of sampled Shapley values, so be sure any code written to handle these normalized values does not assume a strict range or summation to zero. As of writing this chapter, SHAP does not support the ability to normalize Shapley values across predictions.

SHAP can also display global feature attributions. When given all predictions to the `force()` plot function (i.e., `shap.plots.force(shap_values)`), SHAP will render an interactive chart (shown in Figure 3-6) of all attributes across all predictions. This can appear quite intimidating at first.

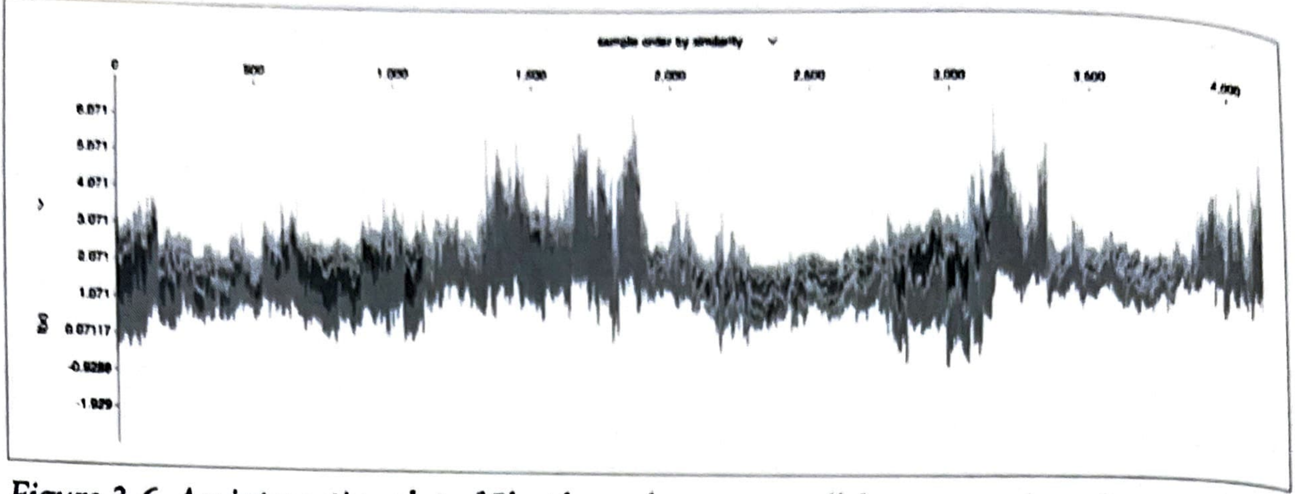

Figure 3-6. An interactive plot of Shapley values across all features and predictions for the test dataset. (Print readers can see the color image at https://oreil.ly/xai-fig-3-6.)

However, the real power in this visualization is to explore the global feature attribution values for an individual feature (as in Figure 3-7), or how two features compare (as shown in Figure 3-8). For Figure 3-7, we chose `MedInc` as the sample selector on the top and specify `MedInc` effects for the y-axis. This shows how various values of the median income push the model predictions toward (or away) from the baseline.

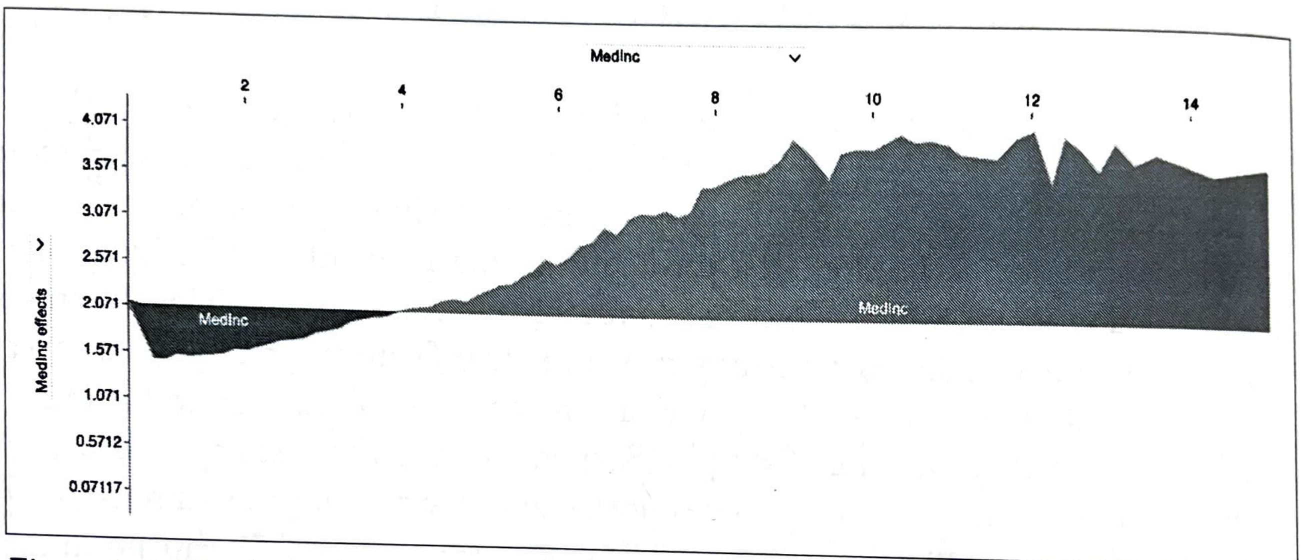

Figure 3-7. Shapley values for the `MedInc` feature across the entire test dataset show how the feature influences the model's prediction as the value of the feature changes.

In Figure 3-8, we swap the output `MedInc` on the left with `Population` to see how the two features compare.

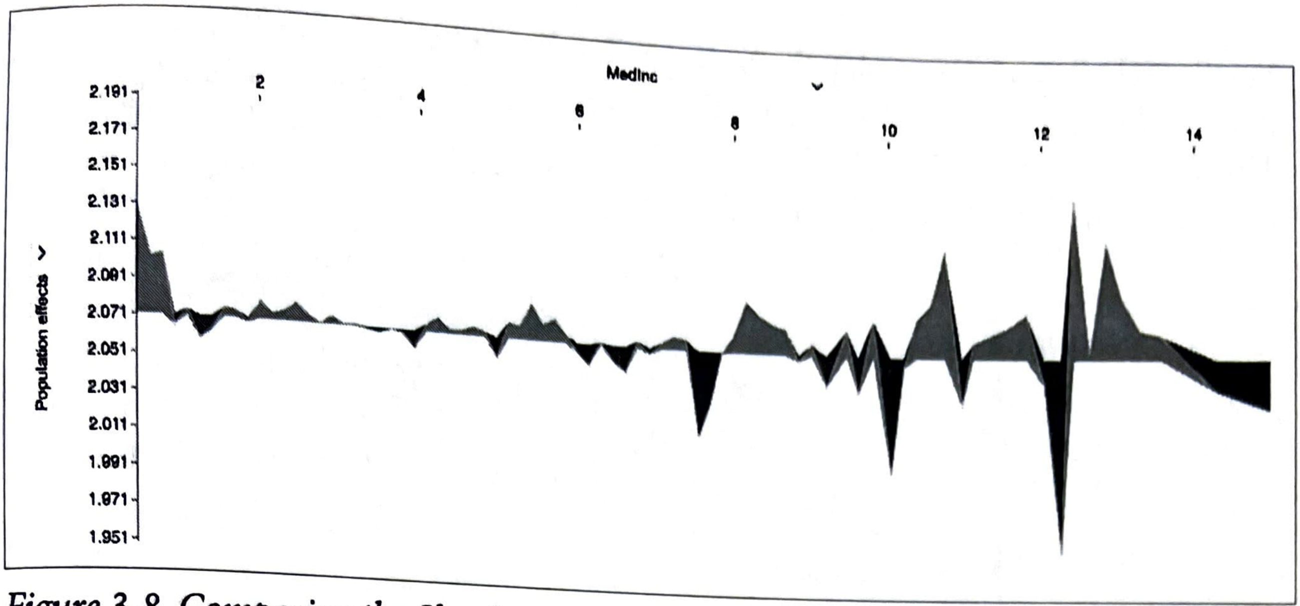

Figure 3-8. Comparing the Shapley value of `MedInc` to the `Population` feature shows that the influence on the model by the median income does not change in a meaningful way as the population changes.

For visualizing the global attributions of features, it may be more palatable to use SHAP's beeswarm plot (shown in Figure 3-9). These features are split into individual rows, with each row plotting all of the individual Shapley values along the x-axis, for all values of that feature. Each value of a feature is rendered as a point, so larger clusters of points (the beeswarm) show where many feature values had a similar Shapley value. SHAP also colors each point with a normalized heat mapping from the feature's lowest to highest values:

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

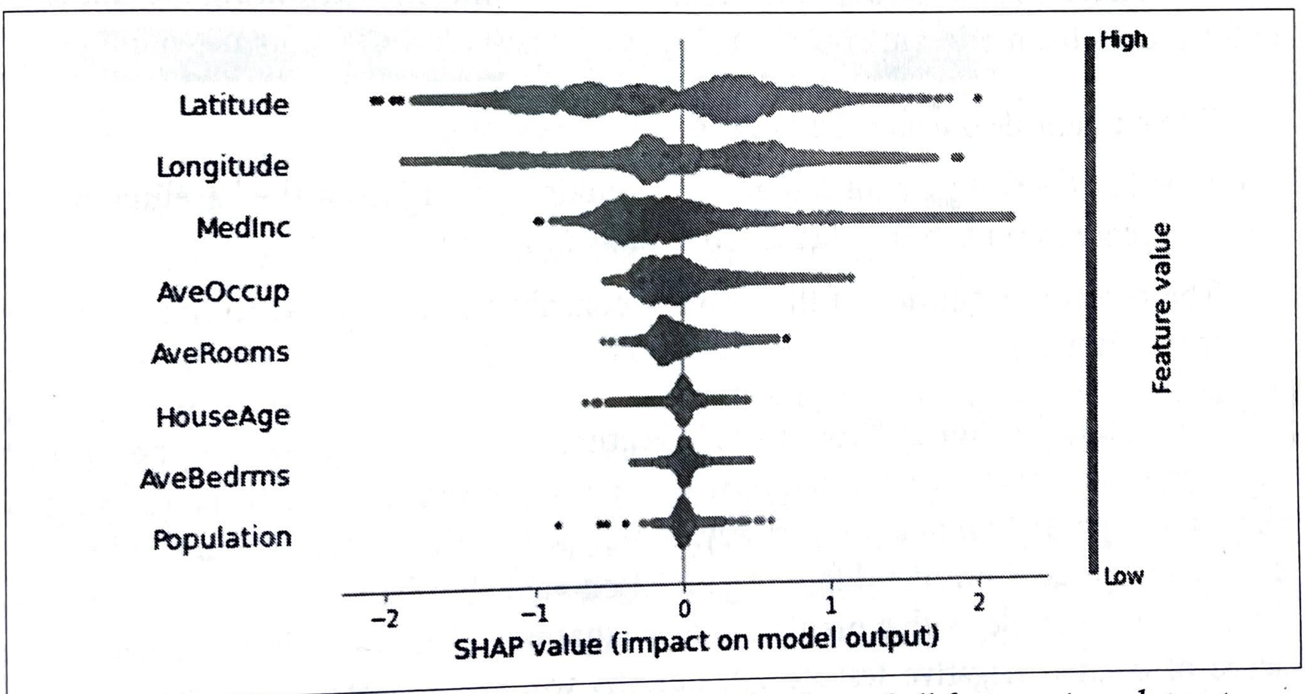

Figure 3-9. Beeswarm plot of Shapley values for all values of all features in a dataset. Note that the x-axis is the Shapley value while the heatmap color of points is related to the values in the dataset. (Print readers can see the color image at https://oreil.ly/xai-fig-3-9.)

Finally, SHAP can also display the average Shapley value for each feature with a bar plot, as shown in Figure 3-10, using the command `shap.plots.bar(shap_values)`. This is often referred to as global feature attributions because it gives insight into the most influential features in the model, regardless of the specific input values.

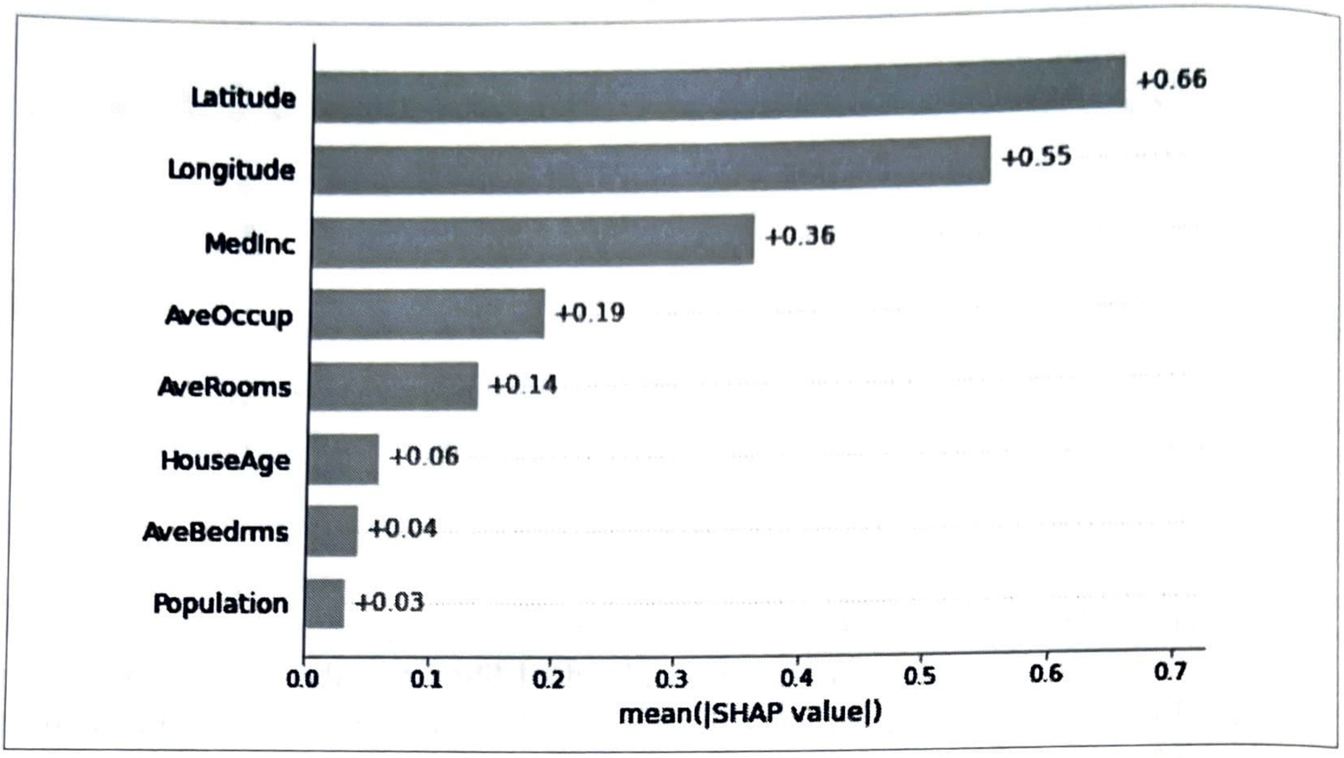

Figure 3-10. Bar plot of the average Shapley value for all features of the model.

Interpreting Feature Attributions from Shapley Values

Once you have Shapley values, what do they actually mean? Attributing the influence of features in the model with Shapley value gives three pieces of information:

- The numerical contribution to the prediction's score
- Whether the feature contributed in the model moving from the baseline prediction score toward the predicted score, or away from it
- The relative magnitude of the feature's contribution compared to other features in the dataset

What is meant by "toward" or "away"? Feature attribution values may be positive or negative, but that does not necessarily mean a positive attribution contributed to the model's prediction in a positive way. It may be that this feature was actually influencing the model toward a different prediction value but had less impact than other features. For a model with a prediction score that was –100 with a baseline prediction score of 0, then negative feature attributions would be contributing "toward" the final predicted score, and positive attributions away from it. Likewise, for a baseline prediction score that is greater than the predicted value, a negative attribution value would be contributing toward the final predicted value.

Managed Shapley Values

Many of the largest ML platforms offer a service for generating Shapley values to understand feature influence, and there are a number of open source libraries as well. Broadly, managed Shapley offerings fall into two groups: those that offer a hosted version of SHAP, and those that have built their own Shapley value library. Given our focus on the SHAP library so far, you may ask what the advantage is of going with a different implementation. While SHAP has become broadly popular, it is only one way of calculating Shapley values, and may not be the most accurate. For the purposes of demonstrating how to work with a managed Explainable AI service, we'll focus on using Google Cloud's Vertex AI Python SDK since that is what we are most familiar with.

Google Cloud Platform (GCP)—Explainable AI

Google Cloud's Explainable AI product supports calculating feature attributions based on Shapley values. While the implementation is proprietary, Google claims it provides faster sampled Shapley value calculations than other implementations and is more robust than many other tabular feature attribution techniques. The framework is model-agnostic and has native support for TensorFlow, PyTorch, XGBoost, and scikit-learn, and the ability to explain any model through the use of Vertex Custom Containers.

There are several ways to use Google Cloud's Explainable AI, as shown in Table 3-5.

Table 3-5. How to access GCP's Explainable AI

Access	Useful for
CLI (gCloud SDK)	Easy testing from the command line
REST API	Incorporating into an existing workflow
Vertex AI Python SDK	Generating cloud-based explanations from a notebook
Vertex AI Notebooks and Workbench	Computing explanations locally within the notebook's VM

If you already (or plan to) use other products in GCP's Vertex AI platform, an added advantage is that their Explainable AI product is built into several other products, such as AutoML, online and batch predictions, and Model Monitoring. Since all of these products rely on the same Explainable AI technology, this gives the flexibility of having portability in your explanations. For example, an AutoML model's global feature attribution values will be valid for understanding any feature attribution values calculated as part of an online prediction request when your model is served in production.

The Explainable AI framework is designed to offer flexibility with your model and your explanation technique. For most DNN models, it is easiest to use the Vertex AI SDK, which will try to infer the input and outputs of your model and generate a metadata configuration for you that we can augment to specify the technique we want (in this case, sampled Shapley) and the number of sample paths to compute. For the purposes of this example, we'll assume you already have a trained TensorFlow 2 model uploaded to Vertex AI (although Explainable AI works with TensorFlow and other frameworks):

```
from google.cloud import aiplatform
from google.cloud.aiplatform.explain.metadata.tf.v2 import
    saved_model_metadata_builder

explainable_ai_builder = saved_model_metadata_builder.SavedModelMetadataBuilder
        (path_to_my_model)
explainable_ai_metadata = explainable_ai_builder.get_metadata()

explanations_parameters = aiplatform.explain.ExplanationParameters({
        "sampled_shapley_attribution": {"path_count": 20}})

endpoint = model.deploy(machine_type="n1-standard-2",
            explanation_metadata=explainable_ai_metadata,
            explanation_parameters=explanations_parameters)
```

When a model is deployed to a prediction endpoint, we also (optionally) pass along the information needed to both tell Explainable AI how to understand our model's inputs and outputs (the metadata) and the parameters regarding what type of explainability we want.

To get an explanation, we then make a normal prediction call as shown in the following code. If we wanted to, we could also override some of our Explainable AI configuration for a particular prediction to change the number of paths, use a different baseline, or return the top K features by Shapley value:

```
instances = [{'dense_31_input': test_data.iloc[0].values.tolist()}]
explanation_spec_override = {"parameters":{"sampled_shapley_attribution":
        {"path_count": 5}}
endpoint.explain({"instances": instances, "explanation_spec_override":
        explanation_spec_override})
```

As part of the prediction response, Vertex will also return a payload with the feature attribution values (Shapley values in this case) and an approximation error, which represents how accurate the values were. We can then visualize these feature attributions with the included visualization widget using the following code snippet:

```
m = explainable_ai_sdk.load_model_from_vertex(my_project, 'us-central1',
        model_endpoint_id)
explanations = m.explain(instances)
explanations[0].visualize_attributions()
```

See "Sampled Shapley technique" on page 33 for a discussion of the trade-off between the number of sampled Shapley paths and the attribution error.

Microsoft Azure and AWS SageMaker

Of course, there are other cloud providers that provide model explainability, most notably Microsoft Azure and AWS SageMaker, both of which also use a version of SHAP. SageMaker's Clarity tool uses SHAP's KernelSHAP to generate feature attributions (as shown in this guide (*https://oreil.ly/45aWp*)), allowing the number of sampled paths, how to aggregate Shapley values, and baselines to be configured. Baselines can either be manually provided as a list or automatically sampled and calculated from a provided dataset.

Azure offers an Explainability service (*https://oreil.ly/T0m2X*) with support for `TreeExplainer`, `DeepExplainer`, `LinearExplainer`, and `KernelExplainer`. The service also has a `TabularExplainer` that is intended to be a high-performance improvement on SHAP, choosing the correct type of Explainer based on the model. `TabularExplainer` can also generate synthetic explanations based on extracting summary statistics from the training dataset. Likewise, `TabularExplainer` can sample explanations from the validation dataset. The explanation results can be visualized in Azure's ML Studio.

Explaining Tree-Based Models

Here's what you need to know about explanations for tree-based models:

- Decision trees are intrinsically explainable since each prediction can be described as a series of decision points for the model features, ultimately leading to the final prediction.
- For ensemble tree methods, like random forest, the explanation of the prediction is simply the average of the bias terms plus the average of the contributions of each feature within each tree in the forest.

Tree-based models can be anything from simple decision trees to gradient-boosted trees and random forests. Here are some general pros and cons to consider when thinking about explainability for these types of models.

Pros	Cons
• Explainability for decision trees is intuitive and easy to communicate to nontechnical audiences. • `treeinterpreter` is an easy-to-implement library that can be used to determine feature contributions for many of scikit-learn's tree-based models.	• Although inherently explainable, simple decision trees often do not perform as well as more complicated tree models, like random forest or gradient-boosted trees. • At this time, the `treeinterpeter` library does not support multilabel classification.

Decision trees are a popular tool for regression and classification problems and provide the epitome of interpretability, provided they don't go too deep. A decision tree consists of a series of nodes, each providing a split, or decision point, for various features that ultimately leads to the model output. The features and decision boundary points are determined by the training data in a way to minimize the mixing of class labels in the final leaves of the tree. This mixing is typically measured using entropy in the form of information gain, the Gini index, or the weighted mean square error in the case of a regression task.

> The Gini index is a number between 0 and 1, which represents the purity of the classification split. It's computed by subtracting the sum of the square probabilities of each class from one. A Gini index of 0 represents absolute purity of the classification; that is, each class is perfectly separated. A Gini index of 1 indicates a random distribution among the classes; that is, the classes are evenly split across subsets. Ideally, you want to split the examples in your dataset according to a feature that yields a low Gini index. Thus, when training a decision tree, the features and cutoff values to use at each split are chosen so that the Gini index of each split is minimized.

The process starts at the root node of the tree and each subsequent nonleaf node splits the data so that when you reach the leaf node, you have a predicted outcome. In this way, it's possible to follow the precise decision splits through the tree and explain how the features ultimately contributed to a certain prediction. Furthermore, because of this structure, decision trees easily allow for counterfactual analysis in that a user is able to ask (and answer!) what-if questions regarding how the model makes its decisions.

Looking again at the California Housing dataset and training a simple decision tree with a maximum depth of three nodes, we can visualize the final model, as shown in Figure 3-11. The full code can be found in the GitHub repository for this book. Notice that a feature might be used for more than one split, as in the two nodes that use the `MedInc` feature, or a feature might not be used at all, like the `Population`, `HouseAge`, or `AveBedrms` features:

```
dt_reg = DecisionTreeRegressor(max_depth=3)
dt_reg.fit(X_train, y_train)

dot_data = export_graphviz(dt_reg, out_file="ca_housing.dot",
                           feature_names=cal_features,
                           filled=True, rounded=True,
                           special_characters=True,
                           leaves_parallel=False)
graph = pydotplus.graphviz.graph_from_dot_file("CA_housing.dot")
Image(graph.create_png())
```

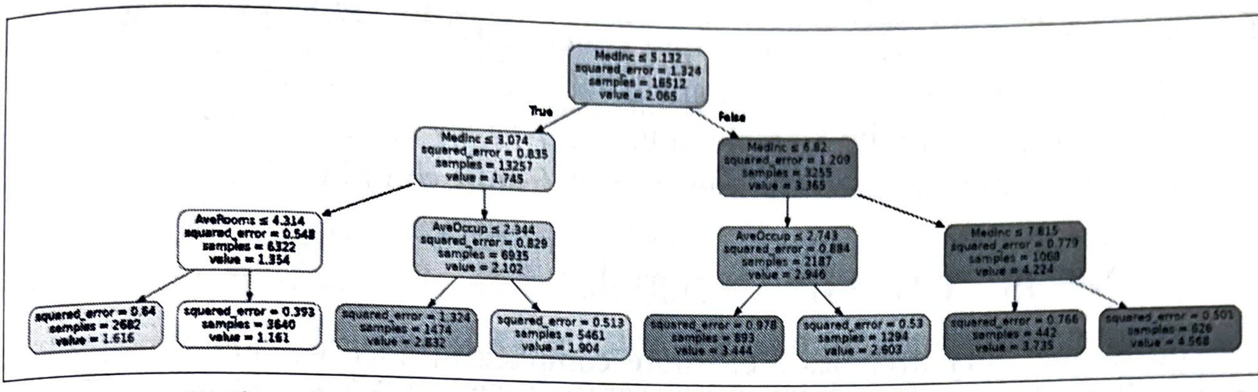

Figure 3-11. Each node of the decision tree indicates a decision cutoff for the features of the given instance.

With a maximum depth of only three nodes, the final model isn't difficult to visualize. From Figure 3-11, we see that the root node for this decision tree has `MedInc` ≤ 5.132 and each subsequent node is determined by a similar feature cutoff. If the condition of a node is met, the decision path moves to the left; otherwise, the decision path goes to the right. Figure 3-12 traces the decision path for a given instance in the test set. For this particular instance, following through each step of the decision tree, since `MedInc` ≤ 5.132, `MedInc` ≥ 3.074, and `AveOccup` ≥ 2.344, the predicted median house value is $190,400.

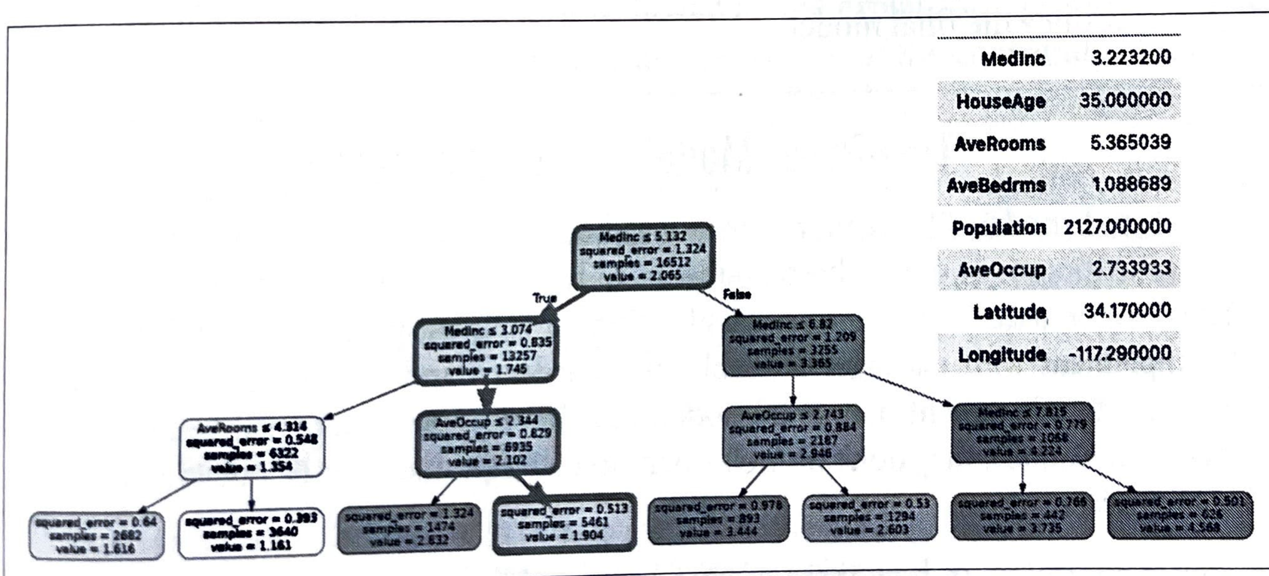

Figure 3-12. The decision path for a given instance can be traced through the decision tree according to the feature values for the instance and the learned decision cutoff of the tree.

The value of the root node is the mean of the labels taken from the training data. This is the initial bias of the model. To explain the prediction for this or any given instance, we start at the root node and add or subtract the feature contributions for each node in the decision path. Looking again at Figure 3-12 and the example instance there, we see that the prediction is given by 1.904 = 2.065 (the bias) - 0.32 (because `MedInc` ≤ 5.132) + 0.357 (because `MedInc` ≥ 3.074) - 0.198 (because `AveOccup` ≥ 2.344).

Each decision split is determined by a feature, sometimes a feature may be used more than once (as `MedInc` is), and each split either adds or subtracts from the current value until the final prediction outcome is returned. This gives the contributions of each decision split as in the example in Figure 3-12. By adding the contributions for each feature, we can see exactly how much each feature contributed to the prediction.

From Decision Trees to Tree Ensembles

Of course, the story becomes a bit more complicated once we move from simple decision trees to ensemble techniques like random forest and gradient-boosted trees. Ensemble methods are meta-algorithms that combine several machine learning models as a technique to decrease the bias and/or variance to improve model performance. By building several models, with different inductive biases, and aggregating their outputs, we hope to get a model with better performance.

Random forests use a bagging ensemble technique where many decision trees are trained in parallel and then their predictions are aggregated to create the final model prediction. Gradient-boosted trees and adaptive boosting algorithms build individual decision trees sequentially. The idea behind gradient boosting is to iteratively build an ensemble of models where each successive model focuses on learning the examples the previous model got wrong, ultimately taking a weighted average of those predictions to produce the final model.

Tree-Based Models and Extrapolation

Tree-based models like random forest and boosted trees are widely popular among ML practitioners and have become the go-to model, particularly for structured data. They can be used in regression, classification, or ranking problems, and they're easy to implement with packages available in scikit-learn and TensorFlow. However, as with any technique, this family of models does have its drawbacks that you should be aware of. Namely, they don't do well when predicting values that lie outside the range of the training data.

This is an artifact of how these models split up the input space of a given problem. Whether it's a random forest of decision trees, or an ensemble of decision trees with gradient-boosting framework, these models are trained to find partitions in the feature sets of inputs sometimes many levels deep to bucket each instance into its corresponding label value. More formally, a decision tree with L leaves divides the features space into L regions and determines a set of rules from the training data so that it can output a value to leaves based on averages. These models are inherently noncontinuous functions and thus they can struggle to extrapolate to unseen or uncommon labels.

For example, since the California Housing dataset only has examples with the median values of homes between $15,000 and $500,000, our model will likely not do well when presented for predicting an instance that has the label $1,000,000. Naturally, this is less of a problem for classification tasks as the label is either 0 or 1, but for regression tasks it is something to keep in mind.

Ensemble tree models are a favorite among machine learning practitioners and data scientists in industry because they are easy to train and yield very powerful models. In fact, for structured data, boosted tree algorithms are often considered the go-to model and you'll often see it show up on the leaderboard for many Kaggle competitions. However, due to their ensembled nature, these models often trade explainability for performance. In fact, even a single tree of depth 10 can already have thousands of nodes, meaning that using it as an explanatory model is almost impossible.

The main idea behind explainability of decision trees is that any prediction outcome can be traced from the root node, through the various node decision points, and ultimately to the leaf that determines the prediction. To explain why a certain prediction was made, you can trace through the decision path starting at the root node, either adding or subtracting the contribution at each node depending on the feature value and the learned decision cutoff. For random forests, since the prediction is the average of the predictions of all the trees in the forest, the explanation of the prediction is simply the average of the bias terms plus the average of the contributions of each feature within each tree.

The `treeinterpeter` package (*https://oreil.ly/sxXlm*) has a nice Python implementation and can be used on scikit-learn's `DecisionTreeRegressor`, `DecisionTreeClassifier`, `RandomForestRegressor`, `RandomForestClassifier`, `ExtraTreeRegressor`, `ExtraTreeClassifier`, `ExtraTreesRegressor`, and `ExtraTreesClassifier`.

For a given element of the test set, calling *predict* with the tree interpreter returns the trained models:

```
from treeinterpreter import treeinterpreter as ti
prediction, bias, contributions = ti.predict(rf_reg,
                                             X_test.iloc[[0]].values)
```

Recall, the `bias` is the average of the bias for each of the trees in the model. The `contributions` is a vector of size equal to the number of features in the dataset; in this case, 2.07. We can parse these values to get the following:

```
Bias term (training set mean): [2.06854048]
Feature contributions:
MedInc -0.96
Population 0.61
AveBedrms 0.12
AveRooms -0.11
AveOccup -0.06
Longitude -0.05
Latitude 0.02
HouseAge -0.02
```

This is a post hoc, local interpretation of the trained random forest for the given instance.

> So far we have discussed how to use the `treeinterpeter` library for regression and single label classification tasks. But what about multilabel classification tasks? Unlike normal classification tasks where the label consists of mutually exclusive labels over a collection of two or more potential classes, multilabel classification tasks have a label that consists of more than one nonmutually exclusive class label. For example, an image may contain both a dog and a cat in it, or a toxic comment might be both insulting and threatening, or a movie might be both action and adventure, or a patient might be at risk for heart disease as well as stroke. At the time of writing, the `treeinterpeter` library does not support multilabel classification.

SHAP's TreeExplainer

`TreeSHAP` is an algorithm that can also be used to compute Shapley values for tree-based models. The `TreeExplainer` library is built on the same principles for computing Shapley values as described in the previous section and optimized for tree-based models like XGBoost, LightGBM, CatBoost, PySpark, and most other tree-based scikit-learn models.

To use the `TreeExplainer`, we'll first train a simple XGBoost model using the same California Housing dataset as before. The full code for this example can be found in the treeinterpreter notebook in the GitHub repository (*https://oreil.ly/sdr6a*) accompanying this book:

```
import xgboost as xgb
xgb_reg = xgb.XGBClassifier(max_depth=3,
                            n_estimators=300,
                            learning_rate=0.05)
xgb_reg.fit(X_train, y_train)
```

Just as before, we'll create an `Explainer` class, passing it to our trained XGBoost model, only this time instead of using `Explainer`, we'll call `TreeExplainer`:

```
explainer = shap.TreeExplainer(xgb_reg)
```

Using this `explainer` object, we can then get explanations for an individual prediction, just as we did before. The following code block indicates how to create a "force" plot using the SHAP library; the resulting plot is shown in Figure 3-13:

```
shap_values = explainer.shap_values(X_train)
shap.force_plot(explainer.expected_value[1],
                shap_values[1][0,:],
                X_train.iloc[0,:])
```

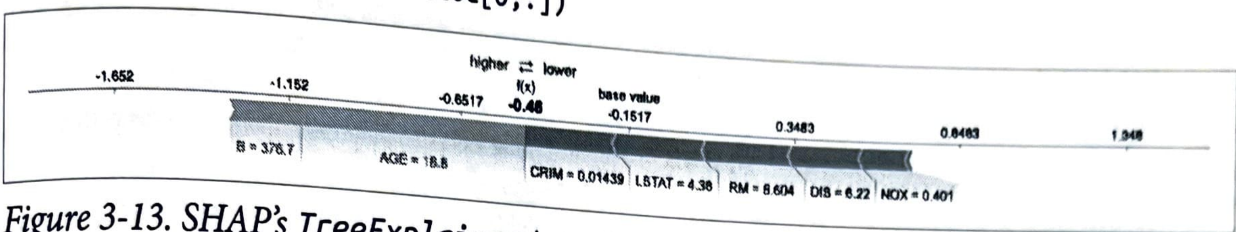

Figure 3-13. SHAP's `TreeExplainer` is optimized for tree-based models. This graph shows the result of an explanation from an individual prediction from an XGBoost model.

Partial Dependence Plots and Related Plots

Partial dependence plots (PDPs), individual conditional expectation (ICE) plots, and accumulated local effects (ALE) plots are a closely related family of explainability tools that allow for visualizing the causal interaction between features and model predictions. These methods are related in that ICE and ALE plots are variations of PDPs and meant to address some of the shortcomings that PDPs may exhibit. We'll outline how each of these techniques works and highlight some of the pitfalls you should be aware of when using these techniques, particularly how ICE and ALE aim to be improvements over PDPs.

As you'll see in the sections to follow, this family of techniques is applied to a model after it has been fully trained and used to visualize the interaction between one or two features of the entire training dataset of the model and the output label. Using the taxonomy outlined in Chapter 2, these techniques are considered post hoc, model-agnostic explainability methods. As we'll see, partial dependence plots are global, whereas individual conditional expectations and accumulated local effects plots are local. In fact, since these methods aim to provide some insight into the causal relationship between feature and the predicted label, they are often thought of as interpretability methods. We'll start by discussing partial dependence plots since they are a bit more easily understood and a simpler entry point to these methods.

Partial Dependence Plots (PDPs)

Here's what you need to know about partial dependence plots:

- PDPs are a useful technique to visualize the marginal effect a specific feature has on a model's predictions.

Pros	Cons
• PDPs are easy to implement and have simple-to-understand interpretation. • When a feature is not correlated to other features, it is possible to infer a causal relationship between that feature and the model predictions.	• There is an underlying assumption of independence of features. • PDPs naively substitute feature values to measure feature dependence. When two features are correlated, the fake data points that are created are not a good representative of the true data distribution. • Sparsity of feature values in the distribution for a given feature (i.e., areas where a feature value lacks good representation) cause the partial dependence plot to be less reliable. • A PDP is really only useful for visualizing at most two features at a time.

PDPs are a useful tool to show how individual input features of a model contribute to the model's predicted outcome variable. This is done by measuring the marginal effect that a specific feature has on the label and plotting the result. By examining the resulting plot, one can easily visualize the causal relationship between the input feature and the label.

When we say we measure the marginal effect, this just means that we measure how the expected value of the model output changes with respect to the feature value. That is, given a trained model f, for any feature, we compute the corresponding partial dependence function as a function over the input feature values by taking the expected value, or average value, over the entire dataset. For example, suppose there are k features in a dataset consisting of n examples. Then we can write each example x_i as $\left(x_i^1, x_i^2, \cdots, x_i^k\right)$. For a regression model, the partial dependence function of the first feature x^1 is given by:

$$f_{pdp}^1(x) = \frac{1}{n} \sum_{i=1}^{n} f\left(x, x_i^2, \cdots, x_i^k\right)$$

where the values of x_i^2 through x_i^k are all the remaining feature values taken from all the n examples in the dataset. Similarly, the partial dependence function for the second feature x^2 is:

$$f_{pdp}^2(x) = \frac{1}{n} \sum_{i=1}^{n} f\left(x_i^1, x, x_i^3 \cdots, x_i^k\right)$$

and for the third feature x^3 is:

$$f^3_{pdp}(x) = \frac{1}{n}\sum_{i=1}^{n} f\left(x_i^1, x_i^2, x, x_i^4, \cdots, x_i^k\right)$$

and so on.

In this way, for any feature we can create and plot the function f^v_{pdp} for any feature v. This plot describes precisely how the predicted output variable changes on average with respect to the given feature. If there is a monotonic relationship between the feature and model predictions, it is immediately apparent.

Of course, in practice you'll use a library to implement this technique, and scikit-learn has a nice implementation called `PartialDependenceDisplay` in the class of inspection tools. As an example, let's look again at the California Housing dataset (*https://oreil.ly/zMb25*). This is a regression problem where the task is to develop a model that predicts the median house price `MedianHouseVal` using as input eight features of the census block. We'll build a model in scikit-learn using a neural network; see the following code block (the full code for this example is available in the notebook for partial dependence plots (*https://oreil.ly/9lnFi*) in the GitHub repository for this book):

```
mlp_reg = MLPRegressor(hidden_layer_sizes=[30, 20, 10],
                       max_iter=500)

# Create pipeline
transformer = ColumnTransformer([
    ('numerical', MinMaxScaler(feature_range=(-1,1)), cal_features),
])

mlp_pipeline = Pipeline(steps=[
    ('transform', transformer),
    ('model', mlp_reg)
])
```

Once the model is trained, to create the partial dependence plots, we'll use the `from_estimator` method of `PartialDependenceDisplay` specifying the model pipeline, the dataset, and which feature we want to visualize:

```
PartialDependenceDisplay.from_estimator(
    mlp_pipeline, X_train, features=['MedInc']
)
```

For the feature `MedInc`, the median income for the families within the census block, the partial dependence plot tells a very clear story. As shown in Figure 3-14, there appears to be a positive correlation between `MedInc` and the target value, the median house value: as median income increases, the median home value increases as well. Also note the similar trend we see represented for `MedInc` in Figure 3-3 as well. This makes sense with what we'd expect our model to learn.

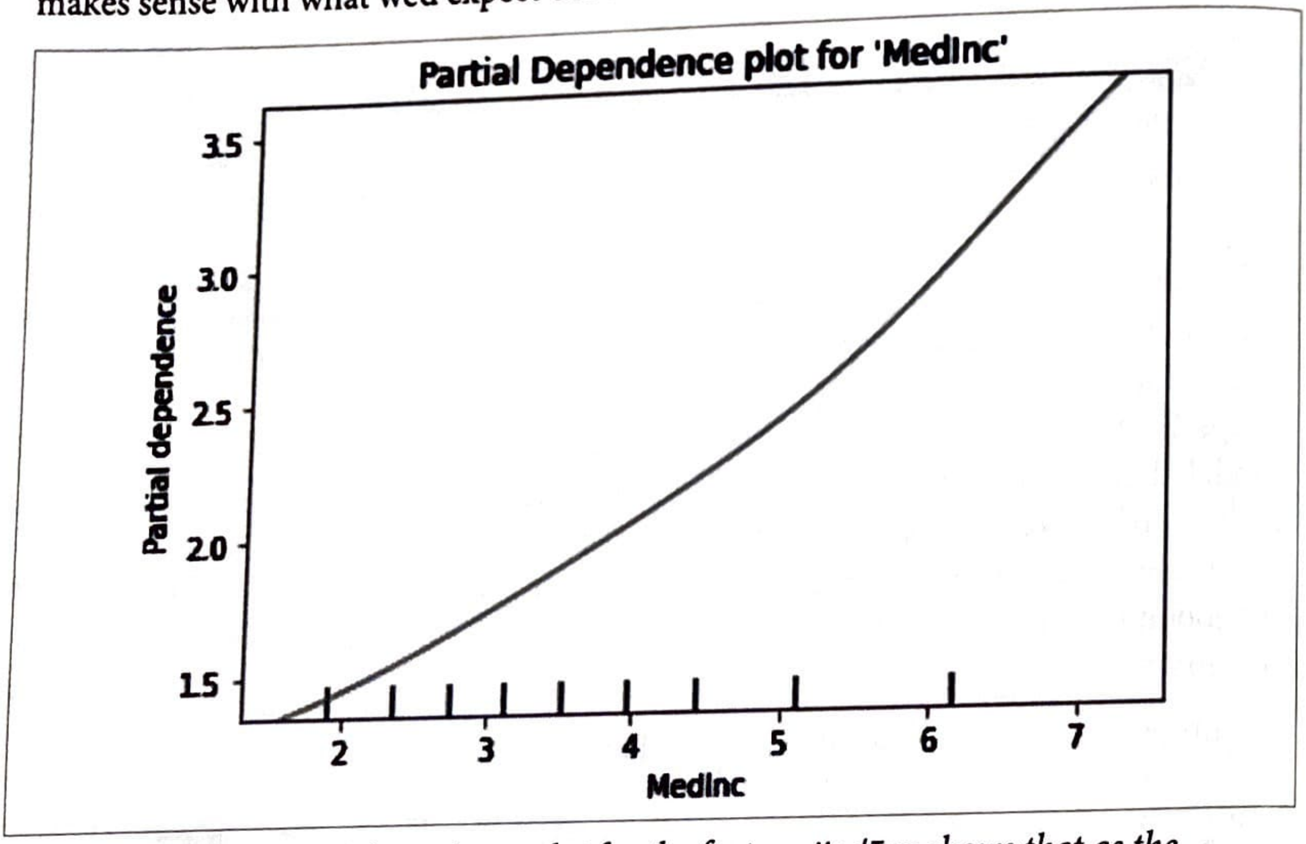

Figure 3-14. The partial dependence plot for the feature `MedInc` shows that as the median family income increases, so does the median home value. The tick marks on the x-axis represent the deciles of the `MedInc` features values.

In fact, for most features in this dataset, the relationship appears fairly straightforward. However, the feature representing the median house age `HouseAge` is more interesting. Figure 3-15 shows the partial dependence plot for the `HouseAge` feature. For houses less than 40 years old, the relationship is nearly constant, with a slight downward trend. However, for very old homes the median house value starts to increase dramatically. Furthermore, these examples only account for about 10% of the training data.

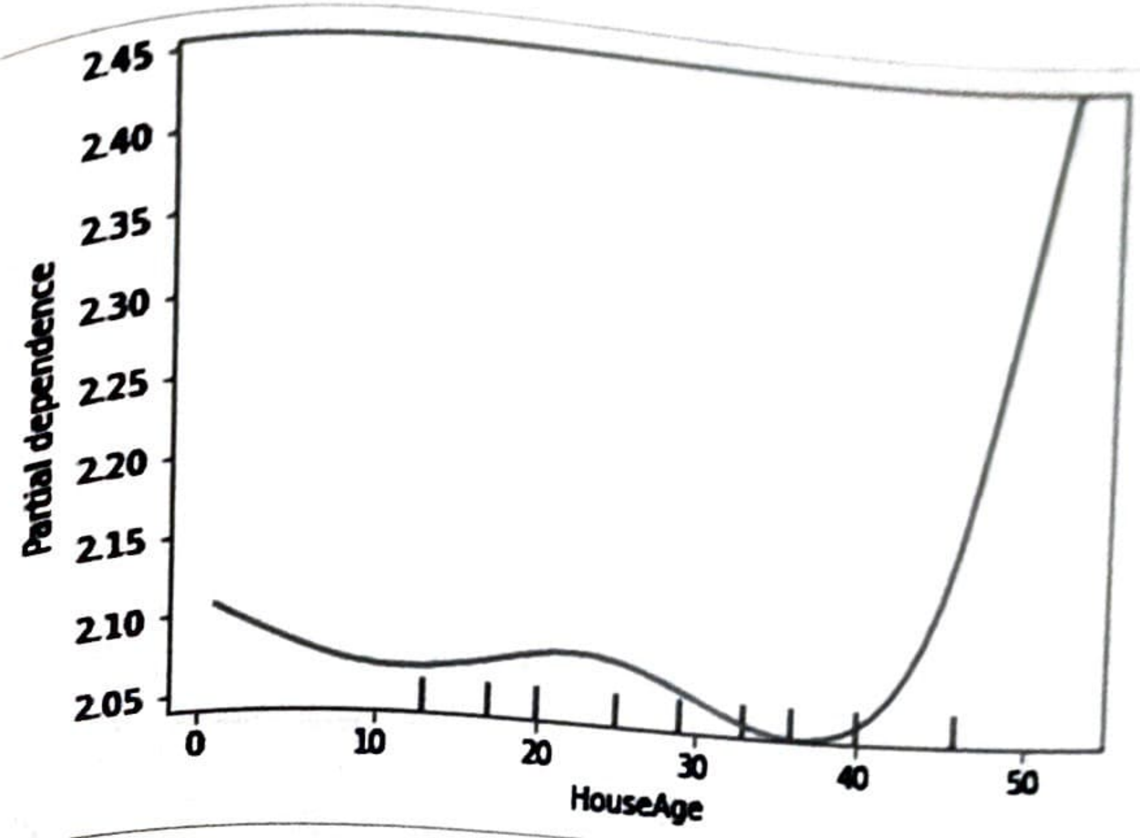

Figure 3-15. The HouseAge feature appears to have a quadratic relationship with the model predictions.

Working with classification models

For classification models, the partial dependence function is defined in exactly the same way. In the case of binary classification, the model predicts a probability that a given example belongs to the positive class. So, the partial dependence function returns the marginal effect a feature has on the predicted probabilities.

What about multiclass classification? So far, we've described how to apply partial dependence plots for regression tasks and binary classification tasks; but what if our model is multiclass? In this case, you can plot the partial dependence for a feature against each of the possible output labels. Take, for example, the Wine Quality dataset (*https://oreil.ly/mQMND*) from the UCI ML Repository. This dataset can be viewed as a classification task to predict the quality of the wine on a scale from 0 to 10 using physicochemical properties like acidity, citric acid, chlorides, and pH as features.

When building a multiclass model like this, we'll use the `OnevsRestClassifier` in scikit-learn as shown in the following code block (the full code for this example is contained in this book's GitHub repository (*https://oreil.ly/9lnFi*)):

```
from sklearn.multiclass import OneVsRestClassifier
multi_clf = OneVsRestClassifier(
    MLPClassifier(
        hidden_layer_sizes=[256, 128, 64, 32],
        max_iter=500)
    ).fit(X, y)
```

To visualize the partial dependence plots, we must create a plot for each target class. In this dataset, the target values are 3, 4, 5, 6, 7, and 8; the corresponding partial dependence plots are shown in Figure 3-16.

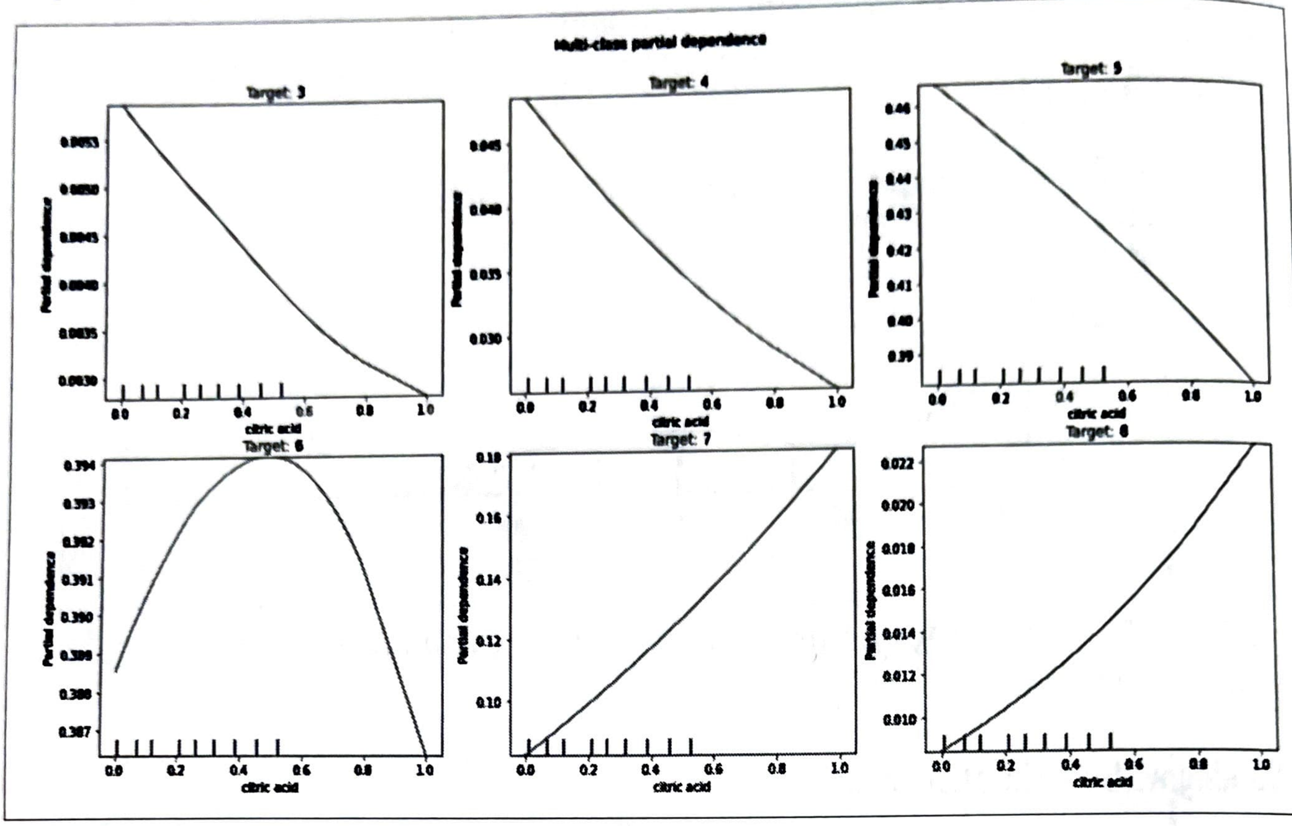

Figure 3-16. For multiclass classification, create a partial dependence plot for each target class.

We have to take extra care in interpreting the causal relationship between different target labels. For the wine quality dataset, we see that the "citric acid" feature is negatively correlated with the labels 3, 4, and 5, positively correlated with target labels 7 and 8 and somewhere in between for target label 6.

But what does this mean? One interpretation is that there is some tipping point for citric acid amounts when assessing the quality of red wines. However, as with all explainability techniques, this information should be viewed in relation to the other features. More likely, the concentration of citric acid interacts with other physicochemical properties of the wine that together influence the target quality label.

Assumption of independence

Partial dependence plots are a simple and intuitive way to explain how model features influence model predictions. When a feature is not correlated to other features, the partial dependence plot provides a direct causal connection by showing how model values change on average as the feature value changes. However, as with all explainability methods, the visualizations provided by partial dependence must be taken in context, especially because it's highly likely that some input features are at least somewhat correlated.

Look at the partial dependence plots for AveRooms and AveBedrms from the California Housing dataset (*https://oreil.ly/sxq3w*) side by side in Figure 3-17. In this dataset, AveRooms represents the average number of rooms per census block and AveBedrms indicates the average number of bedrooms per census block. Not surprisingly, their partial dependence plots are nearly identical.

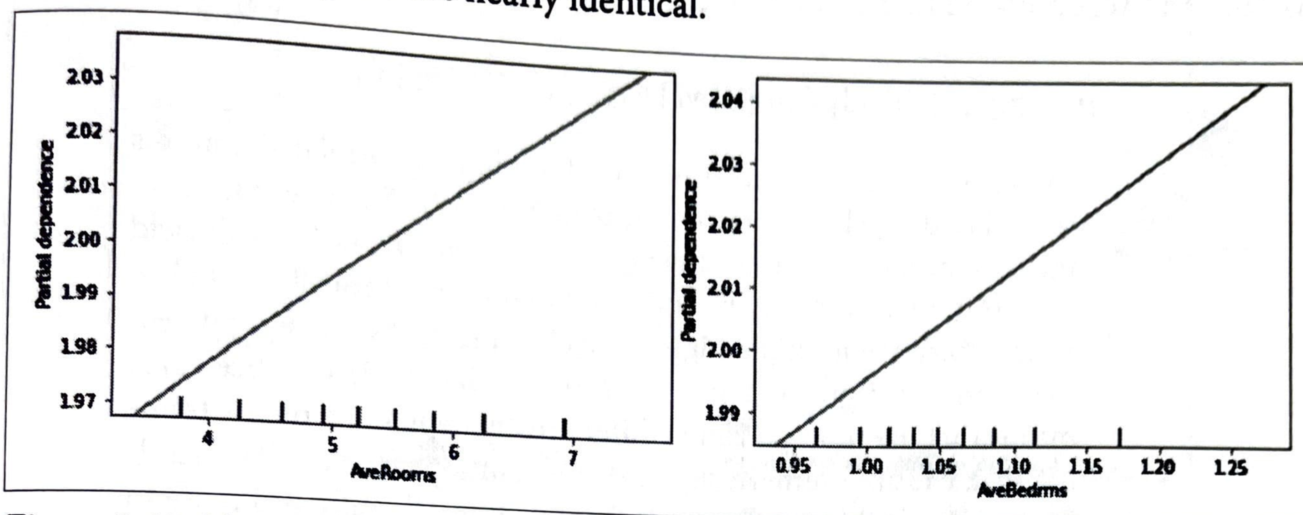

Figure 3-17. The partial dependence plots for AveRooms and AveBedrms are both positively correlated with expected median house value. However, these features are highly correlated.

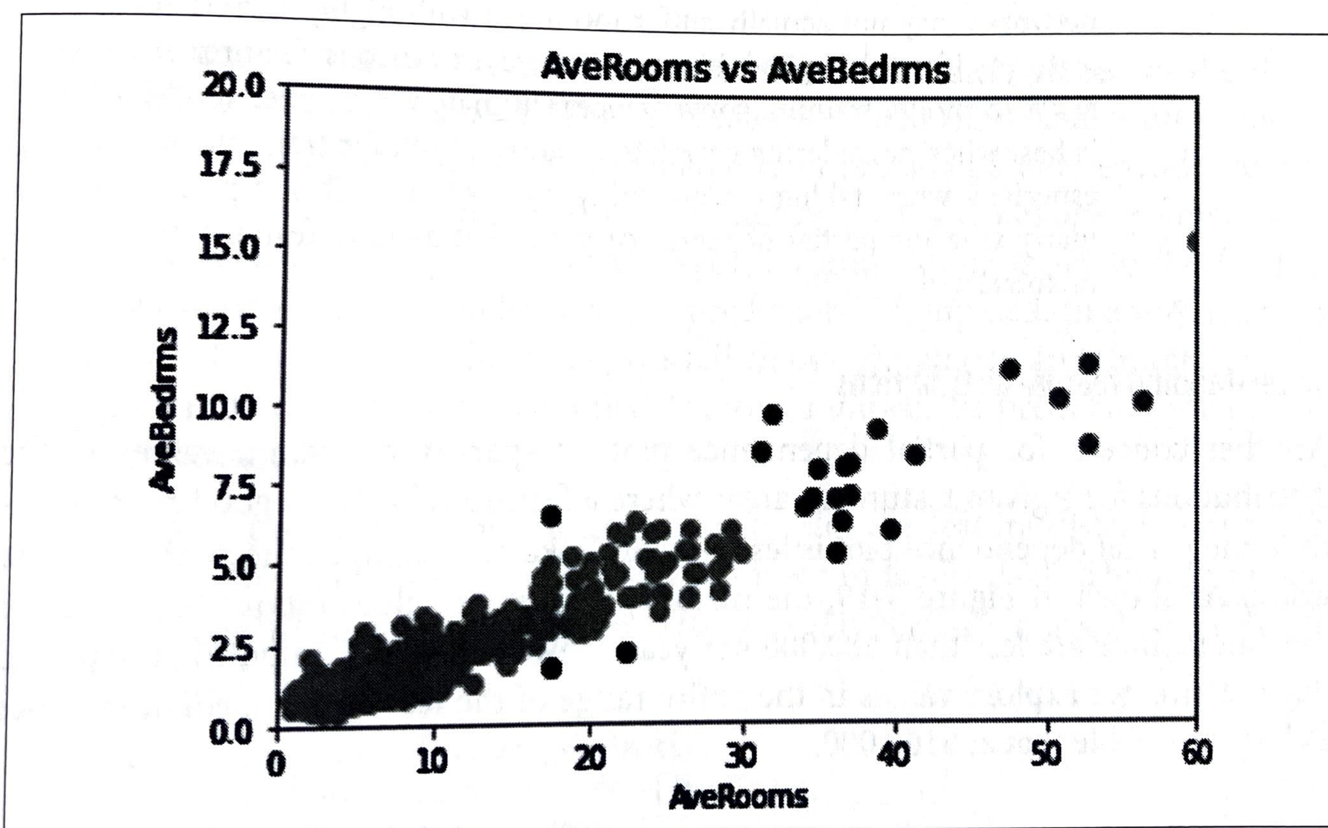

Figure 3-18. The features AveRooms and AveBedrms are very highly correlated.

However, these two features are positively correlated, as shown in Figure 3-18, with a Pearson correlation coefficient of 0.847. So, when it comes to explaining our model and interpreting how these features influence the model's predictions, it's unclear how much the AveRooms feature alone is affecting median house values or if AveBedrms is really a confounding variable between `AveRooms` and the target variable. The lack of independence distorts the causal interpretation of the partial dependence plot when attempting to explain our model predictions. One way to address this problem is to use a conditional distribution instead of the marginal distribution. We'll see how this is done in "Accumulated Local Effects (ALE)" on page 81.

> **Dealing with Highly Correlated Features in ML Models**
>
> In general, it is best practice to exclude highly correlated features when building classic machine learning models. For linear models, this is a must since multicolinearity (*https://oreil.ly/YstUb*) can yield solutions that vary wildly and are possibly numerically unstable. When there are multiple highly correlated features, the weight vector from the regression normal equation has high variance. This means that the model weights differ greatly depending on the training data causing numerically unstable solutions that don't generalize well. In addition, extraneous features typically only add noise to models during training and can lead to longer convergence times.
>
> More generally, while correlated features might not harm your model, they certainly won't improve it either. So, while deep neural networks may not actually suffer too much with highly correlated features, it's still a good idea to remove extraneous features to assist in model training convergence. Ultimately, a simpler model is best when considering correlated features, both for training and especially when taking explainability into account. This is particularly true for partial dependence plots that assume features are uncorrelated.

Understanding feature distributions

Another concern for partial dependence plots is sparsity of feature values in the distributions for a given feature. In areas where a feature value lacks good representation, the partial dependence plot is less reliable. Take, for example, the `MedInc` feature again. As shown in Figure 3-19, the majority of feature values (about 97%) for the median income are less than $80,000 per year. However, when computing the partial dependence, we explore values in the entire range of the feature for median incomes as high as double that at $160,000.

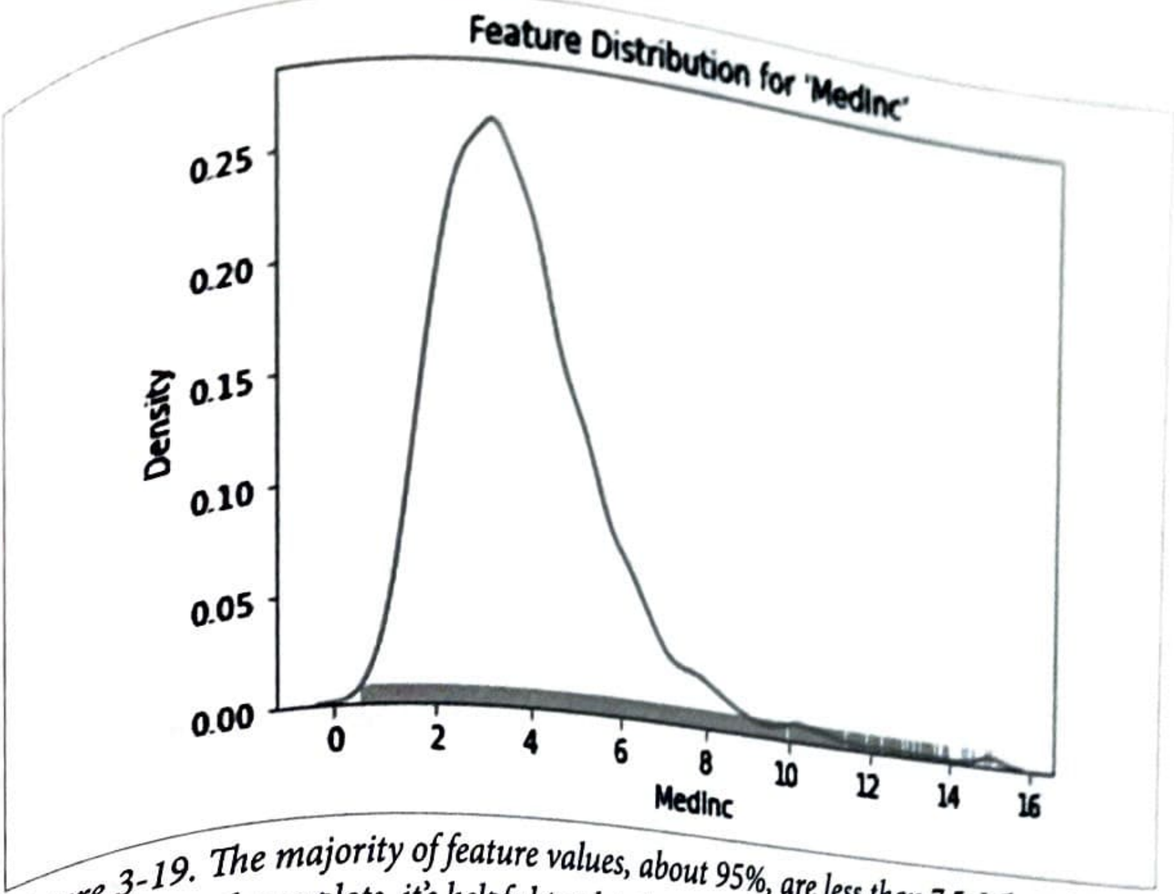

Figure 3-19. The majority of feature values, about 95%, are less than 7.5. When plotting partial dependence plots, it's helpful to plot the feature distributions as well.

These extreme values for the median income don't accurately represent the distribution of our training dataset. This can cause our model to make unreliable or unrealistic predictions that lead to unreliable or unrealistic partial dependence plots for those values. Remember, when computing the partial dependence function for a given feature value, say $m = 0.8$, the `MedInc` feature value is set to 0.8 for each training example and the model's average prediction is computed. In a way, it doesn't make sense to assign an outlier to apply such an outlier value to an arbitrary training example in the dataset. Our model may return an unrealistic prediction for such an unrealistic example.

One way this problem can be ameliorated is to plot the feature distribution on the same axes as the partial dependence plot, as shown in Figure 3-20. This can be done using the `rugplot` function in seaborn (*https://oreil.ly/3iHGB*), a well-known Python data visualization library:

```
PartialDependenceDisplay.from_estimator(
    mlp_pipeline, X_train, features=['MedInc'])
sns.rugplot(data=X_train, x='MedInc')
```

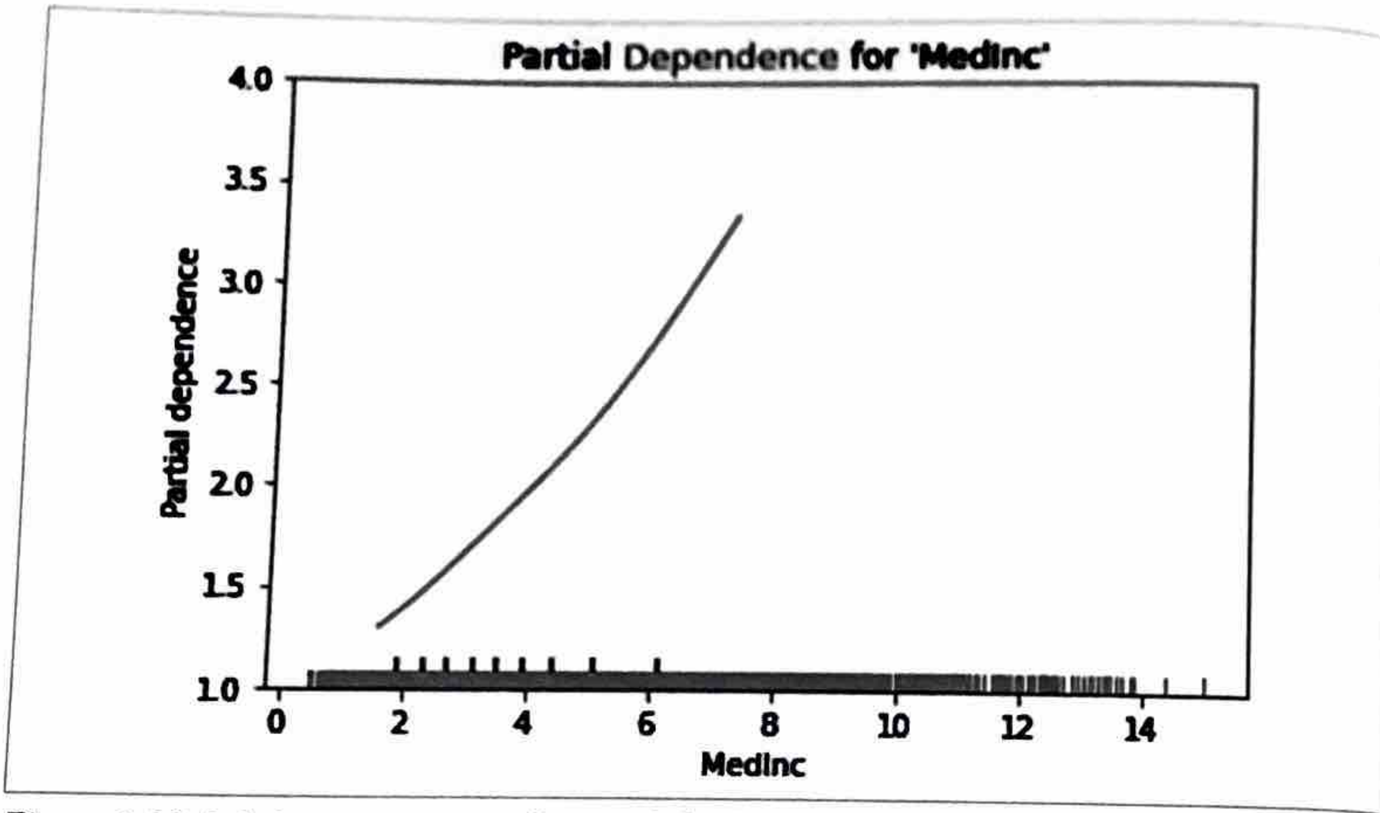

Figure 3-20. It is important to understand the feature value distribution when interpreting the partial dependence plot.

Looking at Figure 3-20, you're likely more inclined to trust the portions of the partial dependence plot where the feature distribution is denser as opposed to those areas where there are fewer feature values, e.g., above $80,000 per year.

Partial dependence plots provide a nice intuitive visualization that helps to explain how a feature is related to the model's target predictions. However, a partial dependence plot is really only useful for one feature at a time. It's not uncommon for some ML models to have tens or hundreds of input features, and each partial dependence feature plot must be examined individually. This quickly becomes intractable.

It is possible to plot two features at once, as in Figure 3-21, but visualizing three or more features in a meaningful way is not possible. To plot two features together, we simply specify a feature as an ordered pair, as in the following code block:

```
PartialDependenceDisplay.from_estimator(
    mlp_pipeline, X_train, features=[('HouseAge', 'MedInc')])
```

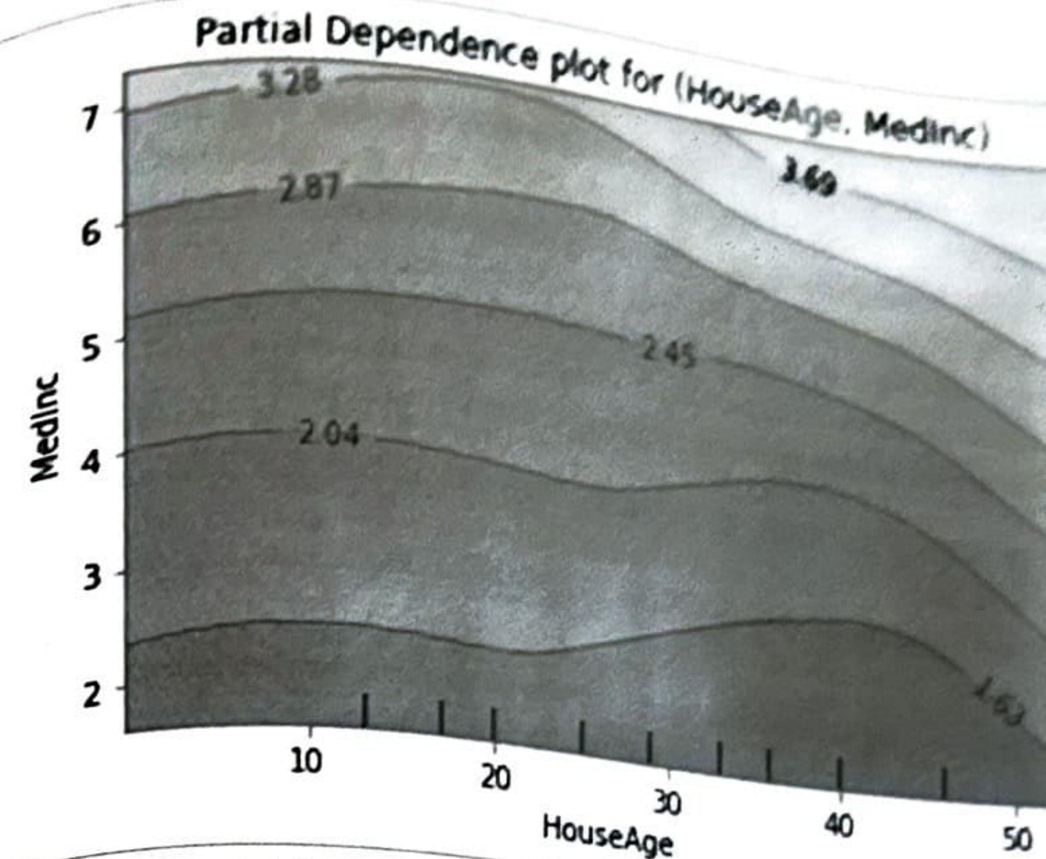

Figure 3-21. The partial dependence plot for two features in the California Housing dataset, HouseAge and MedInc. As HouseAge and MedInc increase together, the median house value also increases gradually. Decreasing MedInc and keeping HouseAge constant decreases median house value; however, the median house value is more stable with changes in HouseAge alone.

Individual Conditional Expectation Plots (ICEs)

Here's what you need to know about individual conditional expectation plots:

- ICE plots extend partial dependence plots by visualizing the dependence on a feature for each instance in the dataset.

Pros	Cons
• ICE plots address some of the shortcomings of partial dependence plots. • They illustrate the dependence of features for each example, giving a more holistic view in lieu of an aggregate. • They allow the user to see heterogeneity in a relationship between feature and model prediction, which is lost when averaging.	• ICE plots have many of the same issues as partial dependence plots, namely an assumption of feature independence. • Graphs quickly become overcrowded and noisy; it's only feasible to plot (at most) one feature at a time.

ICE plots can be thought of as an extension of partial dependence plots. While partial dependence plots graph the overall average of a specific feature value across the entire dataset, individual conditional expectations instead visualize the dependence on a feature for each individual instance. In this way, partial dependence plots are a *global* explainability technique while ICE plots are considered a *local* explainability technique. In short, for any feature, the partial dependence plot is an average of the model prediction values obtained from the ICE plot. One line in the ICE plot represents the predictions for a single instance in our dataset mapped as a function of varying feature value for the feature in question.

Partial dependence plots give aggregated information, but ICE plots allow you to visualize feature contributions on the individual example level. This can be really helpful, because aggregations can lose vital information. Take, for example, the `HouseAge` feature in the California Housing dataset (*https://oreil.ly/38qFE*), which represents the median age of the houses in the census block. Looking at the partial dependence plot, note the left plot in Figure 3-22: when the median house age is greater than 40 years, there is a strong positive correlation with the predicted target. This may be somewhat justifiable since older, historic homes could increase in value. However, that isn't always the case and it's important to not deduce a causal relationship that might not exist. Certainly, there are older housing blocks that do not have high median value.

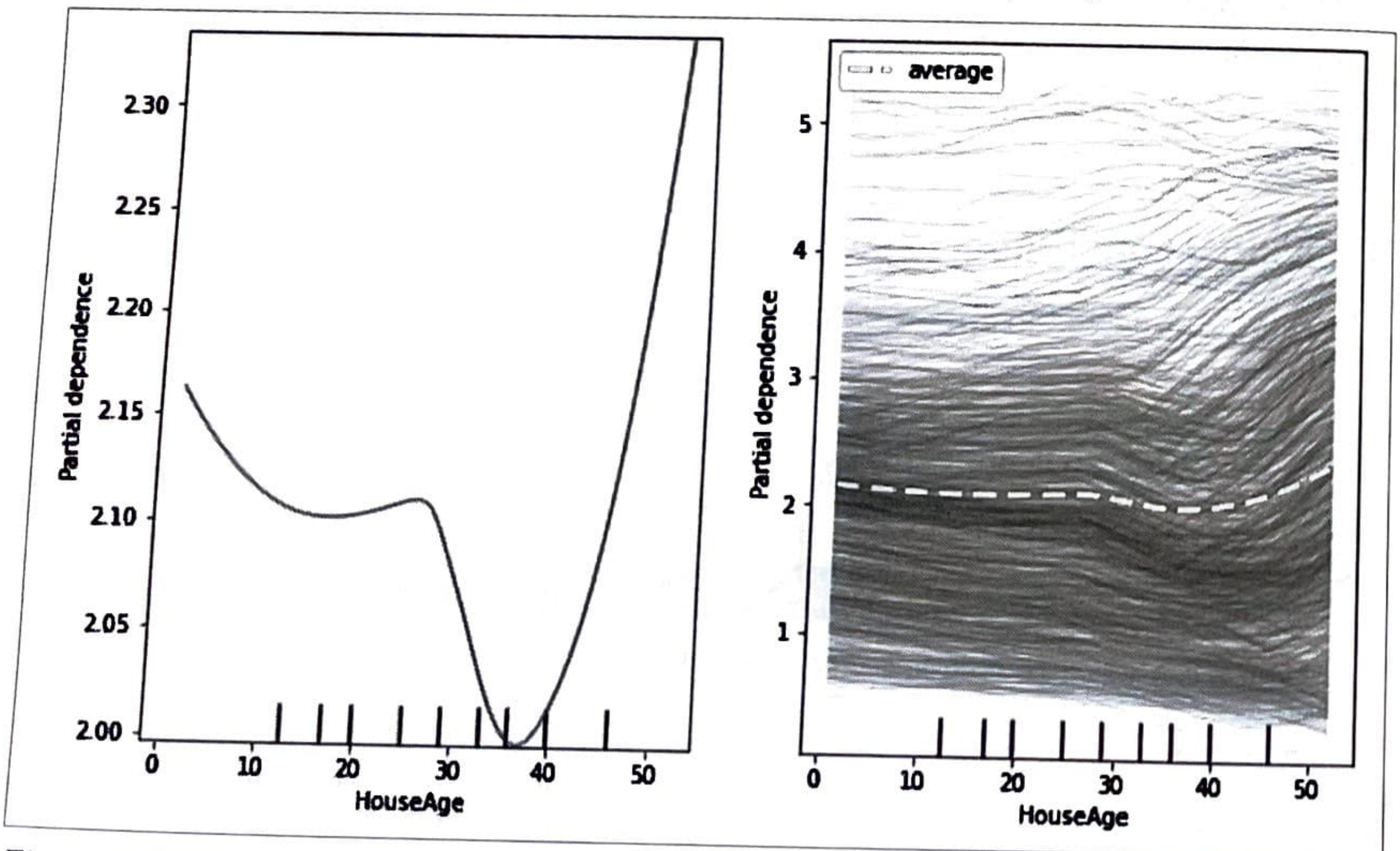

Figure 3-22. For the `HouseAge` feature, there are a few examples where an increase in the median age of houses after 40 years indicates a decrease in the median house value for that block. Note the range of the difference in the range of the y-axis in the two plots.

Using a partial dependence plot could hide the heterogeneous relationship between the houses' age and the median house value. However, looking at the ICE lines, we can see the overall trend and also note that there are some exceptions. We can see this in the individual conditional expectation plot on the right in Figure 3-22. Although the majority of lines do curve up (in fact, in the overwhelming majority), once HouseAge is greater than 40, there are a few examples where the median house value is decreasing.

Accumulated Local Effects (ALE)

Here's what you need to know about accumulated local effects plots:

- ALE plots improve upon partial dependence plots by taking into account the conditional dependence of correlated features and by computing the marginal effect by taking differences instead of averages.

Pros	Cons
• There are nice open source libraries that can be used to implement ALE plots. • ALE plots can still yield useful results when features are correlated.	• The implementation of ALE plots is much less intuitive than PDPs or ICE plots and they can be difficult to explain to nontechnical audiences. • Although ALE plots can handle correlated features, you should still be careful when interpreting results in this setting because strongly correlated features will vary together.

Similar to partial dependence plots, ALE plots visualize the relationship between the features of a model and the model's predictions. However, ALE plots improve upon partial dependence plots in two important ways:

- They take into account the conditional dependence of correlated features.
- They measure the marginal effect of a feature value by computing differences instead of averages.

These improvements allow ACE plots to represent a feature's influence more accurately without the confounding effects of correlated features. Take, for example, the first point. One of the biggest faults of partial dependence plots is that they handle correlated features in a very naive way. Recall that the algorithm for producing a partial dependence plot for a feature simply varies the value of that feature over a range of values in the feature space and measures the average model prediction value. When two features are correlated, this approach becomes problematic because the fake data points that are created when sampling the feature space are not a true representative of the data distribution.

Suppose you have a dataset that indicates the presence of heart disease in a patient from various patient features including sex, age, smoking history, height, and weight, among others. There is a positive correlation between a person's height and weight, so the partial dependence plot for either of these features would be very misleading. When computing the partial dependence plot for height, an entire range of height values are naively substituted into the dataset and their predictions are averaged to get a single value. However, for a sample feature value height of 5 ft., when you naively replace the height of all training examples to be 5 ft. then there could be some examples with height of 5 ft. and weight of 300 lb. Generally speaking, this would be considered an outlier for the dataset and not an accurate representation of the training dataset. As a result, the model that was trained on the training distribution, without examples like this, will have a very skewed prediction on this outlier and have a negative effect on the resulting PDP.

One way to address this issue of correlated features is to take into account the conditional distribution of one feature with respect to another so that these unlikely examples, like the patient that is 5 ft. tall and weighs 300 lb., are weighted less when computing the average of model predictions. This allows you to de-emphasize those examples that wouldn't normally occur in the dataset that would ultimately throw off the model predictions and ultimately the resulting partial dependence plot.

However, this approach still suffers from combining the effects of correlated features. Accumulated local effects plots go a step further, as described in the second point at the beginning of this section. That is, ALE plots mitigate this issue by taking the differences of model predictions in place of averages. By measuring the differences instead of the averages or conditional averages, the ACE plot is better able to gauge the effect of the given feature value on the model prediction.

So, for our height feature, to measure the local effect at a height of 5 ft., we'll compute the local effect by taking in the model's predictions for each example, first setting the height to be 4.9 ft. and then taking the height to be 5.1 ft. and computing the difference. In this way, the ALE technique separates out the effect of the height being 5 ft. without being influenced by the weight, or other correlated features. These local effects are then accumulated and plotted to produce the ALE plot for height.

To create ALE plots for our dataset and model, we'll use an open source Python library called Alibi (*https://oreil.ly/NvZm1*). This library has implementations of various local and global explainability methods and its interface is similar to that of scikit-learn, in that there are distinct initialize, fit, and explain steps. We'll look again at the California Housing dataset (*https://oreil.ly/DNVcC*).

To get some intuition about how ALE plots work, we'll start by looking at a linear regression model. By definition, linear models are intrinsically explainable, so it will be a nice toy model to verify the behavior we see in the plots. There are eight features in the dataset including the latitude, longitude, the average number of rooms, and average number of bedrooms per house in the block, as well as others. If we were to train a linear model on this dataset, it would have the form

$\hat{y}(x) = w_0 + w_1 x_1 + \cdots + w_8 x_8$, and since the model is linear, there are no interactions between the various features. In fact, the effect of any feature would simply be the learned coefficient of that input feature value. So, if x_1 represents `MedInc`, the median income of a housing block in tens of thousands of US dollars, then the sign and magnitude of the coefficient w_1 indicates the positive or negative effect that the median income has on the predicted median house value for a given block.

We'll train a simple linear regression and examine that model's predictions with respect to the `MedInc` variable. Not surprisingly, there is a strong linear relationship: as median income increases, so does the median value of homes. For each example in the training dataset, Figure 3-23 plots the median income of each example on the x-axis and the model's prediction for that example on the y-axis. This is done using the following code block (see the ALE notebook (*https://oreil.ly/dez47*) in the GitHub repository for this book for the full code):

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

feature_names = data.feature_names
index = feature_names.index('MedInc')

fig, ax = plt.subplots()
ax.scatter(X_train[:, index], lr_reg.predict(X_train))
```

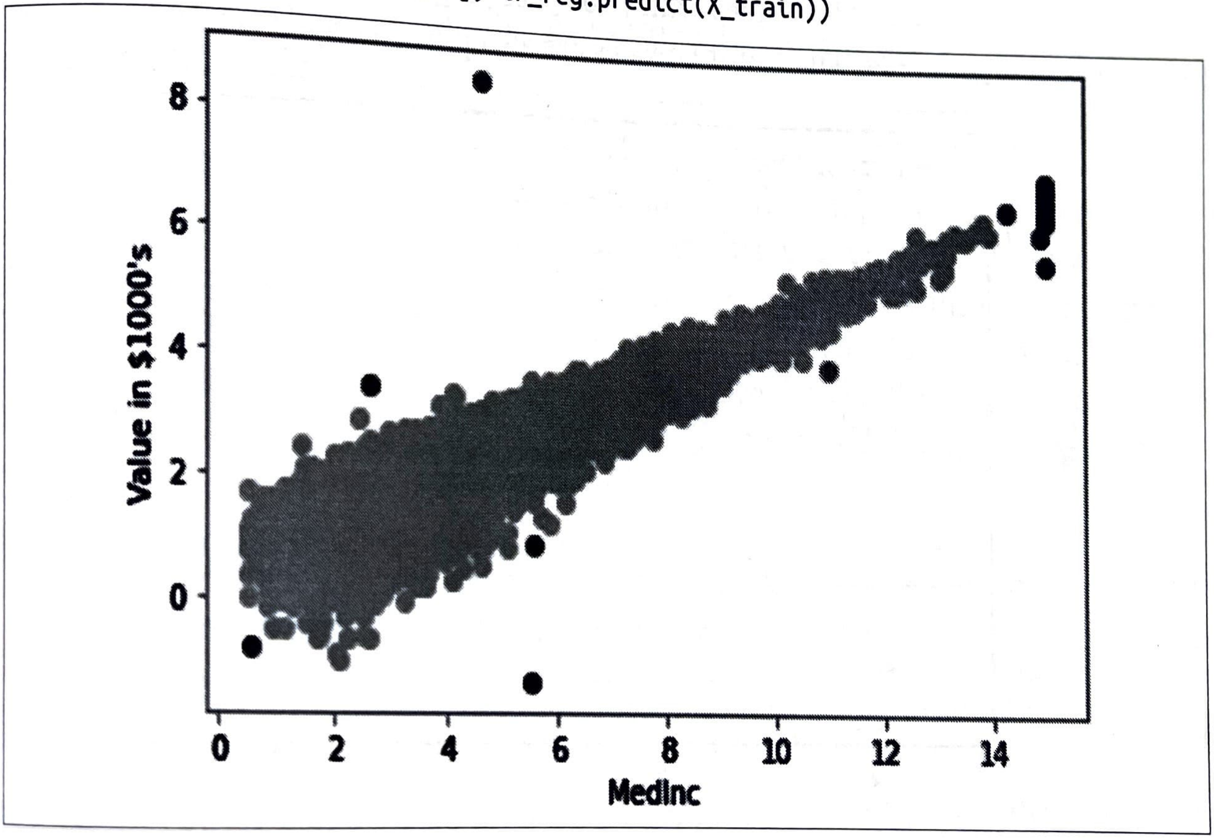

Figure 3-23. There is a strong positive correlation between the `MedInc` feature and the model predictions. As the median home income increases, so does the median value of homes.

We can print the learned coefficients from the linear model and find that the weight for the MedInc feature is about 0.439. This is approximately the slope of the line fit to the points in Figure 3-23. So, for each unit increase in the median income, the model's prediction for the median home value increases by a factor of 0.439. Of course, this doesn't take into account any effects with other, possibly correlated, features in the dataset. Using this simple linear regression as a model, we can see a bit more how ALE works "under the hood."

To start, we initialize the ALE object by passing it a predictor function from our trained linear regression model, a list of the feature names of the dataset, and the target variable name. We can then call the `explain` method, which returns an `Explanation` object that we can inspect and use for displaying the ALE plots. Since ALE plots are a global explainability technique, the `explain` method takes as an argument a batch of data on which to compute the ALE values. We'll use the entire training dataset `X_train`:

```
lr_ale = ALE(lr_reg.predict,
             feature_names=feature_names,
             target_names=['MedianHouseVal'])
lr_exp = lr_ale.explain(X_train)
```

Using the `lr_exp` `Explanation` object, we can visualize the effect of the `MedInc` feature by plotting the ALE values with `plot_ale(lr_exp, features=['MedInc'])`, as shown in Figure 3-24. The feature deciles are also plotted on the x-axis.

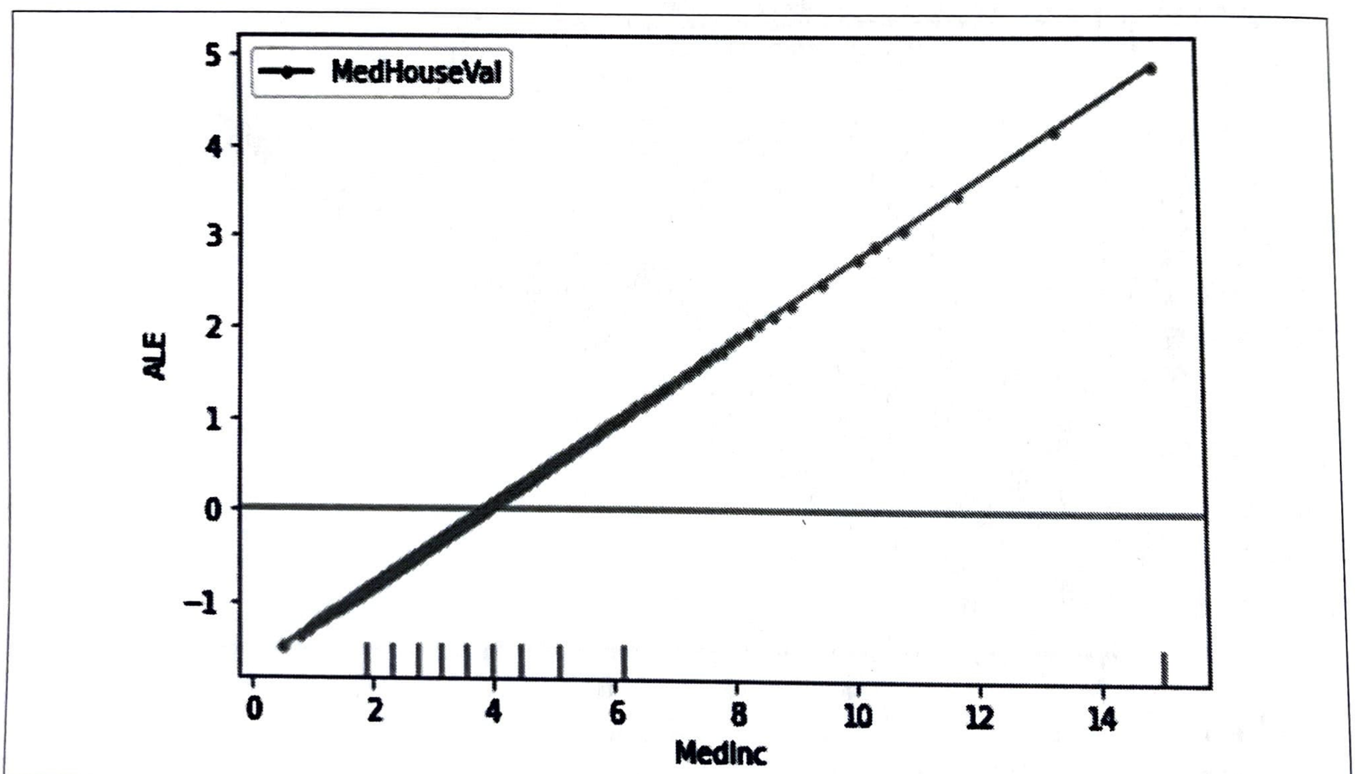

Figure 3-24. The ALE plot for `MedInc` shows the effect of median income on the linear regression's predicted value for median house value across the training dataset. When `MedInc` is less than 4, the effect is negative; i.e., lower median income causes the predicted value of down with respect to the average prediction.

The plot in Figure 3-24 is a line of slope approximately 0.43, which we would expect because our model is a linear regression and the learned weight coefficient for MedInc is 0.43. In fact, `lr_exp` contains a numpy array of ALE values for each feature in the dataset, so we can use `plot_ale` to plot any feature we request with the `features` argument. Calling the `plot_ale` function without arguments will plot the effects of every feature. Doing so, you'll see that the slopes of the ALE plots are precisely the coefficients of the linear regression model.

Remember how the algorithm for ALE works: first the feature `MedInc` is partitioned into a set of intervals covering the range of those feature values. For each interval, the feature value for `MedInc` for the data points within the interval are replaced with the upper and lower interval endpoints, and the difference in the model predictions are averaged across the examples. Since we have a linear model, all of the features (other than `MedInc`) nicely cancel each out in the difference and the only term remaining is the learned weight value taken across the interval.

Take, for example, the case when `MedInc=12` in Figure 3-24. The y-axis represents the local effect of the median income; in this case, the ALE value is about 3.47, corresponding to an increase of about $3,500 for the predicted median house value solely due to the `MedInc` feature. The interpretation is that for neighborhoods where the median household income is about $120,000 per year, the model predicts an uplift of about $3,500 due to this median income feature when compared against the average model prediction.

We can explore this a bit deeper in the data by taking the examples in `X_train` that have `MedInc` values close to 12 and measuring the total expected uplift for median house value for neighborhoods in this range. In this example, the feature values are partitioned so that this corresponds to all training examples with `MedInc` between 11.78 and 13.39. We compute the average of the model predictions on this subset and compare against the average prediction on the entire dataset; this is the zeroth order effect of the linear model. See the notebook (*https://oreil.ly/dez47*) in the GitHub repository for the full code for this example:

```
subset = X_train[(X_train[:, index] > 11.78)
                 & (X_train[:, index] < 13.39)]
lr_reg.predict(subset).mean() - lr_reg.predict(X_train).mean()
```

We get that the difference is about 3.7. This is the total expected uplift for housing blocks with median income close to $120,000.

Of course, for nonlinear models the story is a bit more complicated in that the model will learn more complicated relationships between different features and, as a result, the ALE plots are no longer linear. For example, when training a random forest model, the corresponding ALE plot for `MedInc` is nonlinear and nonmonotonic, as shown in Figure 3-25.

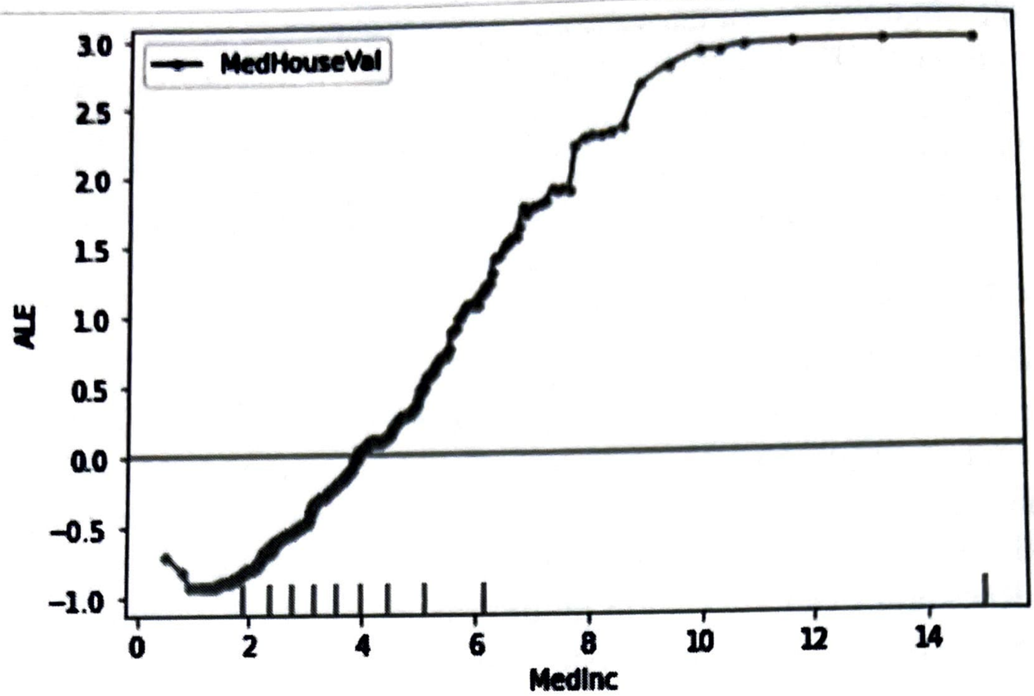

Figure 3-25. Since a random forest model is not linear, the ALE plots should not be expected to be linear either.

We can interpret the plot in Figure 3-25 in the same way as before. Namely, the ALE value at a point is the relative feature effect with respect to the mean feature effect of the random forest model. For example, in the plot, the ALE value for `MedInc=8` is about 2.0. This means that for neighborhoods where the median household income is about $80,000 per year, the model predicts an uplift of about $2,000 for the median house value due to the median income feature with respect to the average model prediction.

Visualizing the ALE plots for each of the features together, as in Figure 3-26, we see that the features that have the largest influence on median house value are `MedInc` (the median income for households within a neighborhood, measured in hundreds of thousands of US dollars), `AveOccup` (average number of household members), and the `Latitude` and `Longitude`.

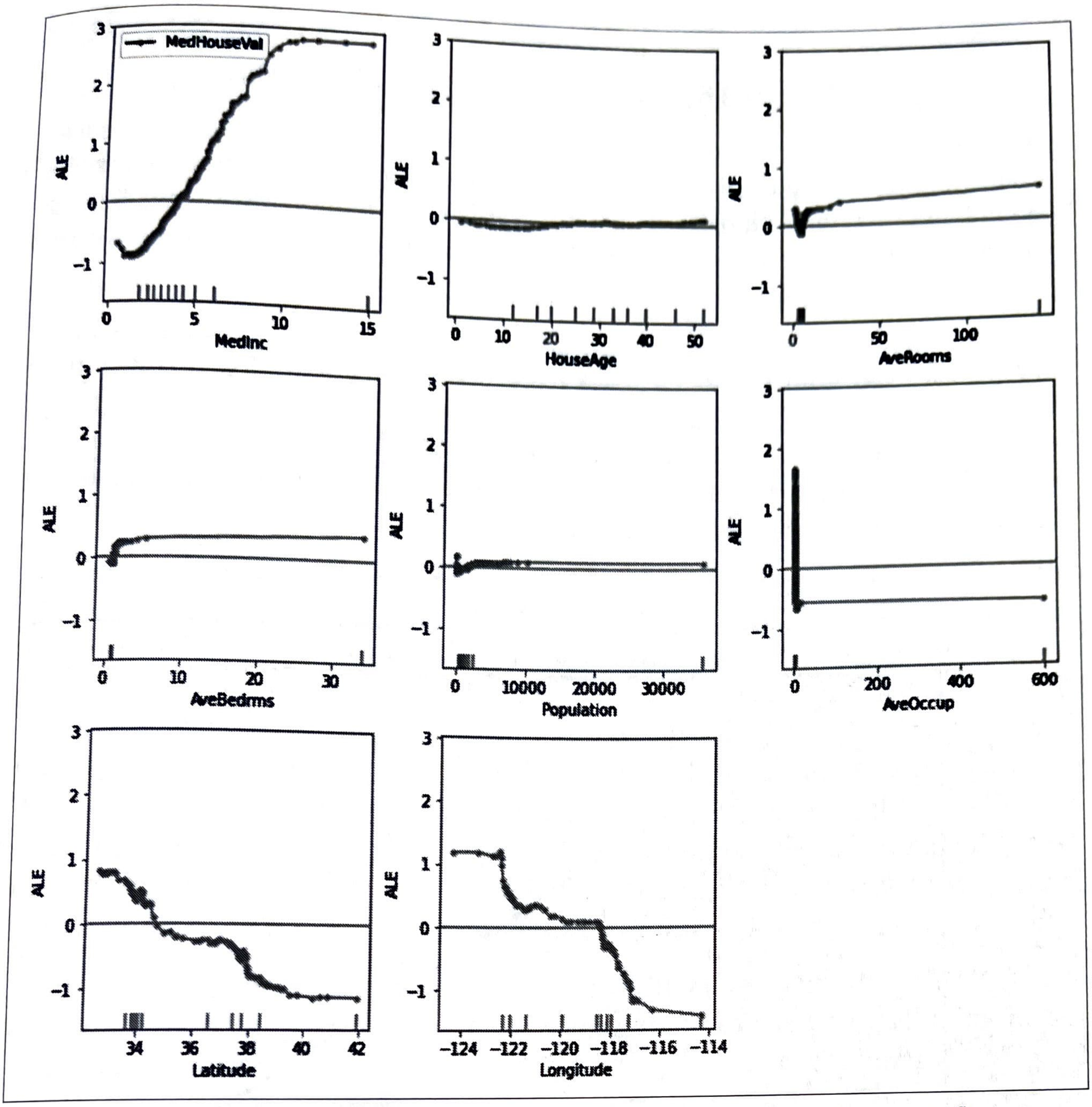

Figure 3-26. For the random forest model, the features that have the largest influence on median house value are the median income, the average number of household members, and the location.

Comparing the Effect of Features Across Different Models

We've seen how the behavior of ALE plots differs for linear versus nonlinear models. Namely, for linear models the ALE plots are linear, whereas for nonlinear models like a random forest, the ALE plots are not necessarily linear or even monotonic. But the differences don't stop there. For example, if we compare the effect of the `AveRooms` feature, we see that the models treat this feature quite differently.

From Figure 3-27, we can see that the random forest model puts a large emphasis on the `AveRooms` feature while the linear regression model doesn't. It's also interesting to note that the feature effects for the linear regression model are strongly negatively correlated while for the random forest there is a slight positive correlation. Also, not only is the feature effect for the random forest nonmonotonic, the effect becomes slightly negative for low values of `AveRooms`.

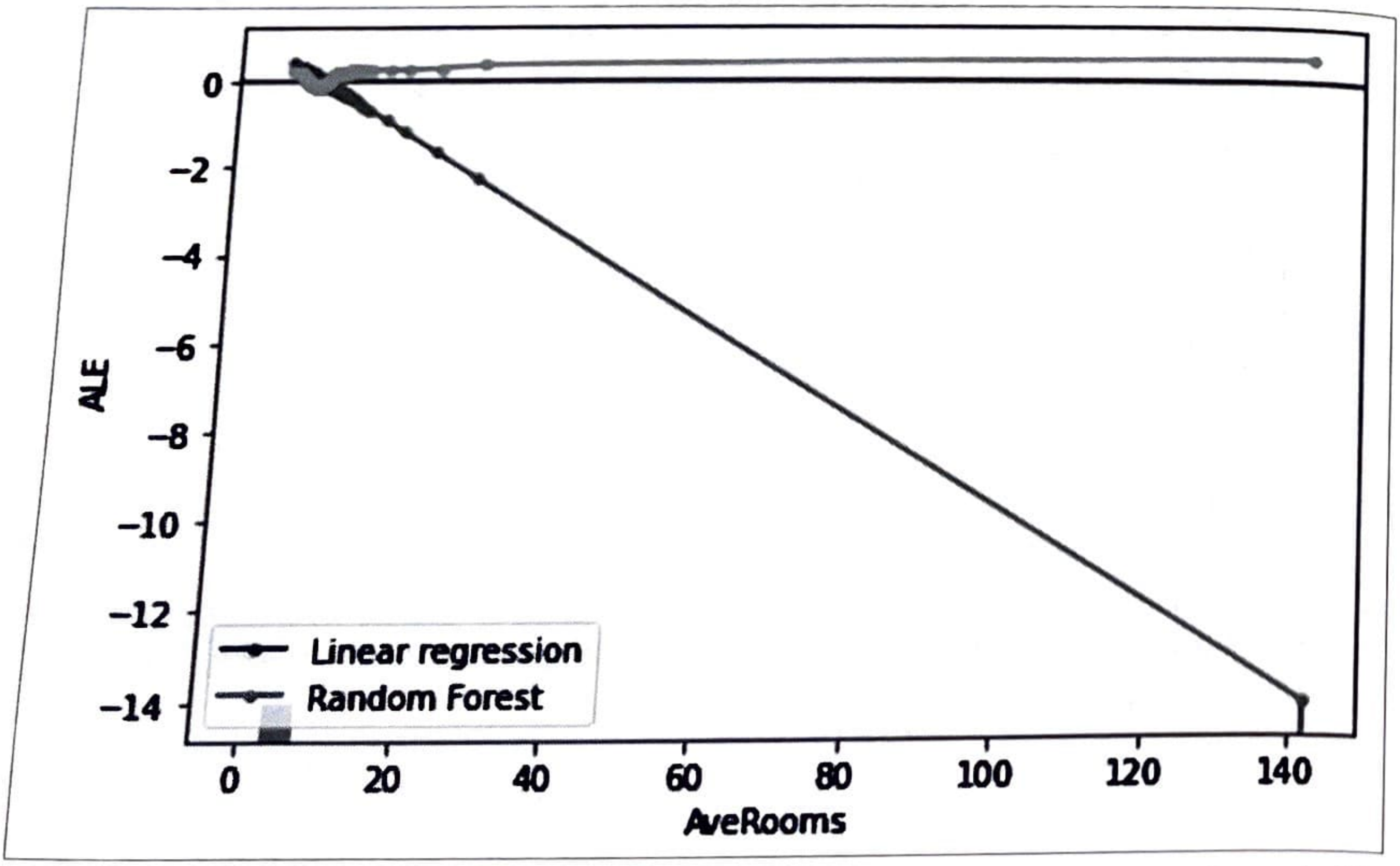

Figure 3-27. The effect of the `AveRooms` feature, average number of rooms per household, differs for the linear regression and the random forest model.

Summary

In this chapter, we saw a collection of explainability techniques that can be used for models trained on tabular datasets. We started with a discussion of feature importances for machine learning models and how they can be measured using permutation feature importance. As the name suggests, this is a perturbation-based explainability technique where the importance of a feature is determined by measuring the change in the model score after permuting the values of a given feature in the dataset. We then dove into Shapley values, starting first with SHapley Additive exPlanations (SHAP), an optimized way to compute Shapley values, and the importance of baselines. The main idea behind baselines is that if one can find a neutral value for a feature, that value will not influence the prediction and therefore not contribute to the Shapley value. The baseline then acts as a kind of placeholder in our model input so we can calculate the Shapley value across different coalitions.

Next, we looked at explainability methods for tree-based models, like decision trees and random forest. Although decision trees lend themselves well to interpretation and are intrinsically explainable for random forest and other more complex tree-based models, we saw how the `treeinterpreter` library could be used to shed light on explaining model predictions. Lastly, we looked at a family of closely related techniques including partial dependence plots (PDPs), individual conditional expectation (ICE) plots, and accumulated local effects (ALE) plots. Each of these techniques visualizes how certain model features contribute to the model predictions. ALE plots address many of the problems associated with PDPs and ICE plots. Namely, they are unbiased and still work well when features are correlated.

In the next chapter, we'll shift our focus to explainability techniques for image use cases. Some techniques we've covered in this chapter, like SHAP, are useful for images with slight modification, but we'll focus on a collection of other techniques designed with computer vision in mind.