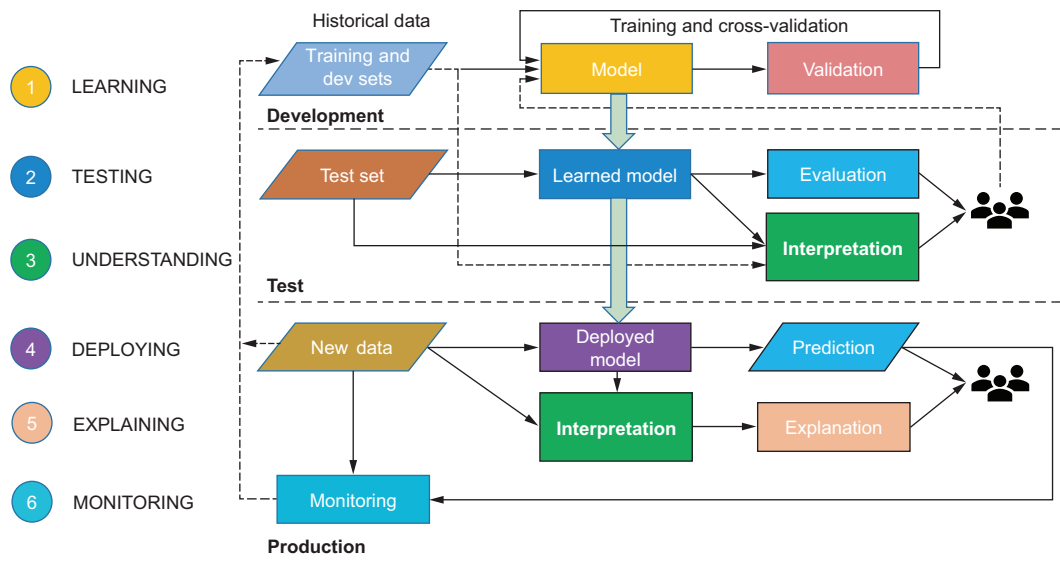


Building explainable machine learning systems

Interpretable AI

Ajay Thampi





The process of building a robust AI system

Interpretable AI

Interpretable AI

BUILDING EXPLAINABLE MACHINE LEARNING SYSTEMS

AJAY THAMPI



MANNING
SHELTER ISLAND

For online information and ordering of this and other Manning books, please visit www.manning.com. The publisher offers discounts on this book when ordered in quantity. For more information, please contact

Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964
Email: orders@manning.com


©2022 by Manning Publications Co. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in the book, and Manning Publications was aware of a trademark claim, the designations have been printed in initial caps or all caps.

- © Recognizing the importance of preserving what has been written, it is Manning's policy to have the books we publish printed on acid-free paper, and we exert our best efforts to that end. Recognizing also our responsibility to conserve the resources of our planet, Manning books are printed on paper that is at least 15 percent recycled and processed without the use of elemental chlorine.

The author and publisher have made every effort to ensure that the information in this book was correct at press time. The author and publisher do not assume and hereby disclaim any liability to any party for any loss, damage, or disruption caused by errors or omissions, whether such errors or omissions result from negligence, accident, or any other cause, or from any usage of the information herein.

 Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964

Development editor: Lesley Trites
Technical development editor: Kostas Passadis
Review editor: Mihaela Batinić
Production editor: Deirdre Hiam
Copy editor: Pamela Hunt
Proofreader: Melody Dolab
Technical proofreader: Vishwesh Ravi Shrimali
Typesetter: Gordan Salinovic
Cover designer: Marija Tudor

ISBN 9781617297649
Printed in the United States of America

White-box models



This chapter covers

- Characteristics that make white-box models inherently transparent and interpretable
- How to interpret simple white-box models such as linear regression and decision trees
- What generalized additive models (GAMs) are and their properties that give them high predictive power and high interpretability
- How to implement and interpret GAMs
- What black-box models are and their characteristics that make them inherently opaque

To build an interpretable AI system, we must understand the different types of models that we can use to drive the AI system and techniques that we can apply to interpret them. In this chapter, I cover three key white-box models—linear regression, decision trees, and generalized additive models (GAMs)—that are inherently transparent. You will learn how they can be implemented, when they can be applied, and how they can be interpreted. I also briefly introduce black-box models. You will learn when they can be applied and their characteristics that make

them hard to interpret. This chapter focuses on interpreting white-box models, and the rest of the book will be dedicated to interpreting complex black-box models.

In chapter 1, you learned how to build a robust, interpretable AI system. The process is shown again in figure 2.1. The main focus of chapter 2 and the rest of the book will be on implementing interpretability techniques to gain a much better understanding of machine learning models that cover both white-box and black-box models. The relevant blocks are highlighted in figure 2.1. We will apply these interpretability techniques during model development and testing. We will also learn about model training and testing, especially the implementation aspects. Because the model learning, testing, and understanding stages are quite iterative, it is important to cover all three stages together. Readers who are already familiar with model training and testing are free to skip those sections and jump straight into interpretability.

When applying interpretability techniques in production, we also need to consider building an explanation-producing system to generate a human-readable explanation for the end users of your system. Explainability is, however, beyond the scope of this book, and the focus will be exclusively on interpretability during model development and testing.

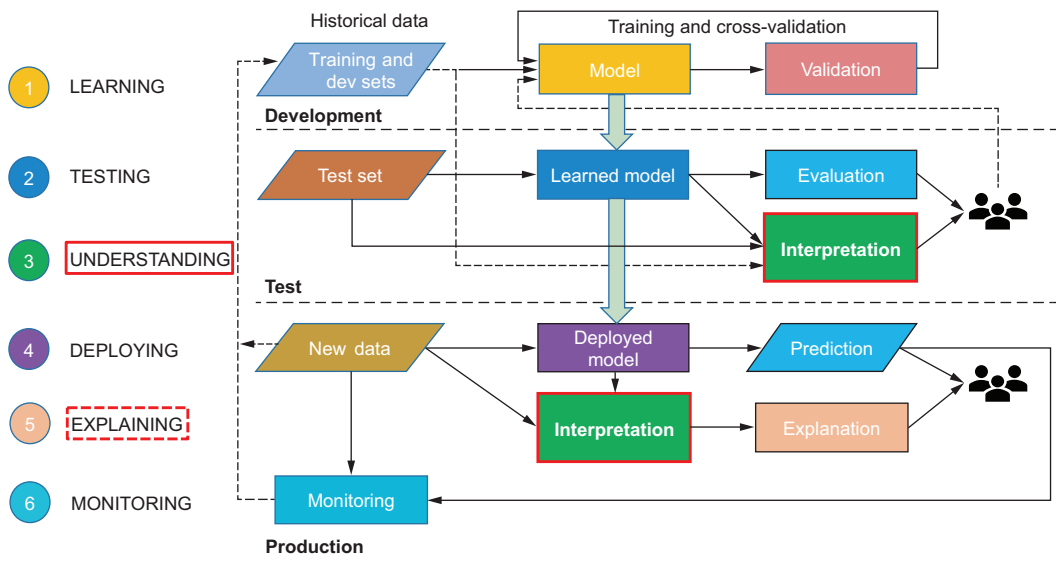


Figure 2.1 The process to build a robust AI system, focusing mainly on interpretation

2.1 *White-box models*

White-box models are inherently transparent, and the characteristics that make them transparent are

- The algorithm used for machine learning is straightforward to understand, and we can clearly interpret how the input features are transformed into the output or target variable.

- We can identify the most important features to predict the target variable, and those features are understandable.

Examples of white-box models include linear regression, logistic regression, decision trees, and generalized additive models (GAMs). Table 2.1 shows the machine learning tasks to which these models can be applied.

Table 2.1 Mapping of a white-box model to a machine learning task

White-box model	Machine learning task(s)
Linear regression	Regression
Logistic regression	Classification
Decision trees	Regression and classification
GAMs	Regression and classification

In this chapter, we focus on linear regression, decision trees, and GAMs. In figure 2.2, I have plotted these techniques on a 2-D plane with interpretability on the x -axis and predictive power on the y -axis. As you go from left to right on this plane, the models go from the low interpretability regime to the high interpretability regime. As you go from bottom to top on this plane, the models go from the low predictive power regime to the high predictive power regime. Linear regression and decision trees are highly interpretable but have low to medium predictive power. GAMs, on the other hand, have high predictive power and are highly interpretable as well. The figure also shows black-box models in gray and italic. We cover those in section 2.6.

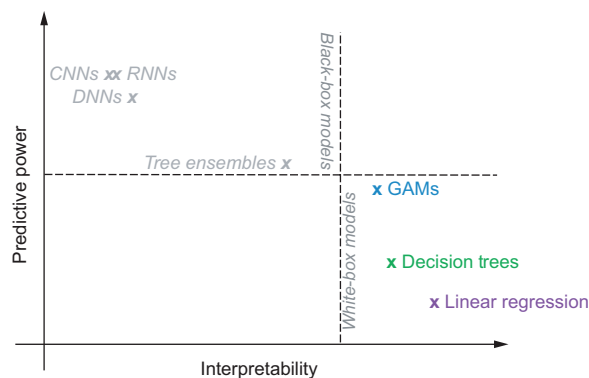


Figure 2.2 White-box models on the interpretability versus predictive power plane

We start off with interpreting the simpler linear regression and decision tree models and then go deep into the world of GAMs. For each of these white-box models, we learn how the algorithm works and the characteristics that make them inherently interpretable. For white-box models, it is important to understand the details of the algorithm because it will help us interpret how the input features are transformed

into the final model output or prediction. It will also help us quantify the importance of each input feature. You'll learn how to train and evaluate all of the models in this book in Python first, before we dive into interpretability. As mentioned earlier, because the model learning, testing, and understanding stages are iterative, it is important to cover all three stages together.

2.2 *Diagnostics+—diabetes progression*

Let's look at white-box models in the context of a concrete example. Recall the Diagnostics+ AI example from chapter 1. The Diagnostics+ center would now like to determine the progression of diabetes in their patients one year after a baseline measurement is taken, as shown in figure 2.3. The center has tasked you, as a newly minted data scientist, to build a model for Diagnostics+ AI to predict diabetes progression one year out. This prediction will be used by doctors to determine a proper treatment plan for their patients. To gain the doctors' confidence in the model, it is important not just to provide an accurate prediction but also to be able to show how the model arrived at that prediction. How would you begin this task?

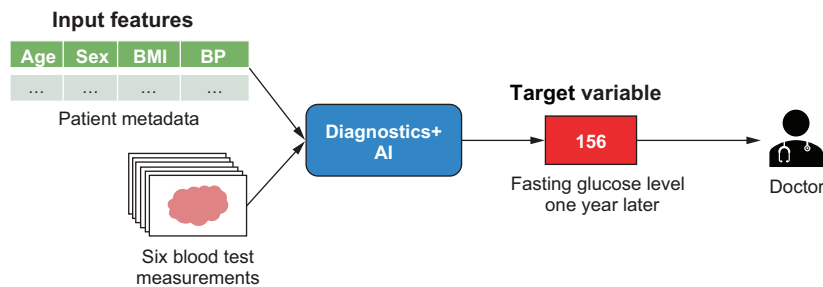


Figure 2.3 Diagnostics+ AI for diabetes

First, let's look at what data is available. The Diagnostics+ center has collected from around 440 patients data that consists of patient metadata such as age, sex, body mass index (BMI), and blood pressure (BP). Blood tests were also performed on these patients, and the following six measurements were collected:

- LDL (bad cholesterol)
- HDL (good cholesterol)
- Total cholesterol
- Thyroid-stimulating hormone
- Low-tension glaucoma
- Fasting blood glucose

The data also contains the fasting glucose levels for all patients one year after the baseline measurement was taken. This is the target for the model. How would you formulate this as a machine learning problem? Because labeled data is available, where you are given 10 input features and one target variable that you have to predict, you can

formulate this problem as a supervised learning problem. The target variable is real valued or continuous, so it is a regression task. The objective is to learn a function f that will help predict the target variable y given the input features x .

Let's now load the data in Python and explore how correlated the input features are with each other and the target variable. If the input features are highly correlated with the target variable, then we can use them to train a model to make the prediction. If, however, they are not correlated with the target variable, then we will need to explore further to determine whether there is some noise in the data. The data can be loaded in Python as follows:

```

> from sklearn.datasets import load_diabetes
  diabetes = load_diabetes()
  X, y = diabetes['data'], diabetes['target']

```

Imports the scikit-learn function to load the open diabetes dataset

Loads the diabetes dataset

Extracts the features and the target variable

We will now create a Pandas DataFrame, which is a two-dimensional data structure that contains all the features and the target variable. The diabetes dataset provided by Scikit-Learn comes with feature names that are not easy to understand. The six blood sample measurements are named s1, s2, s3, s4, s5, and s6, which makes it hard for us to understand what each feature is measuring. The documentation provides this mapping, however, and we use that to rename the columns to something that is more understandable, as shown here:

```

feature_rename = {'age': 'Age',
                  'sex': 'Sex',
                  'bmi': 'BMI',
                  'bp': 'BP',
                  's1': 'Total Cholesterol',
                  's2': 'LDL',
                  's3': 'HDL',
                  's4': 'Thyroid',
                  's5': 'Glaucoma',
                  's6': 'Glucose'}

```

Loads all the features (x) into a DataFrame

Mapping the feature names provided by Scikit-Learn to a more readable form

```

> df_data = pd.DataFrame(X,
                        columns=diabetes['feature_names'])
  df_data.rename(columns=feature_rename, inplace=True)
> df_data['target'] = y

```

Includes the target variable (y) as a separate column

Uses the Scikit-Learn feature names as column names

Renames the Scikit-Learn feature names to a more readable form

Now let's compute the pairwise correlation of columns so that we can determine how correlated each of the input features is with each other and the target variable. This can be done easily in Pandas as follows:

```
corr = df_data.corr()
```

By default, the `corr()` function in pandas computes the Pearson or standard correlation coefficient. This coefficient measures the linear correlation between two variables and has a value between +1 and -1. If the magnitude of the coefficient is above 0.7, that means it's a really high correlation. If the magnitude of the coefficient is between 0.5 and 0.7, that indicates a moderately high correlation. If the magnitude of the coefficient is between 0.3 and 0.5, that means a low correlation, and a magnitude less than 0.3 means there is little to no correlation. We can now plot the correlation matrix in Python as follows:

<pre>import matplotlib.pyplot as plt import seaborn as sns sns.set(style='whitegrid') sns.set_palette('bright') f, ax = plt.subplots(figsize=(10, 10)) sns.heatmap(corr, vmin=-1, vmax=1, center=0, cmap="PiYG", square=True, ax=ax) ax.set_xticklabels(ax.get_xticklabels(), rotation=90, horizontalalignment='right');</pre>	<div style="border-left: 1px solid black; padding-left: 10px;"> <p>Imports Matplotlib and Seaborn to plot the correlation matrix</p> </div> <div style="border-left: 1px solid black; padding-left: 10px; margin-top: 10px;"> <p>← Initializes a Matplotlib plot with a predefined size</p> </div> <div style="border-left: 1px solid black; padding-left: 10px; margin-top: 10px;"> <p>Uses Seaborn to plot a heatmap of the correlation coefficients</p> </div> <div style="border-left: 1px solid black; padding-left: 10px; margin-top: 10px;"> <p>Rotates the labels on the x-axis by 90 degrees</p> </div>
---	--

The resulting plot is shown in figure 2.4. Let's first focus on either the last row or the last column in the figure. This shows us the correlation of each of the inputs with the target variable. We can see that seven features—BMI, BP, Total Cholesterol, HDL, Thyroid, Glaucoma, and Glucose—have moderately high to high correlation with the target variable. We can also observe that the good cholesterol (HDL) also has a negative correlation with the progression of diabetes. This means that the higher the HDL value, the lower the fasting glucose level for the patient one year out. The features seem to have pretty good signal in being able to predict the disease progression, and we can go ahead and train a model using them. As an exercise, observe how each of the features is correlated with each other. Total cholesterol, for instance, seems very highly correlated with the bad cholesterol, LDL. We will come back to this when we start to interpret the linear regression model in section 2.3.1.

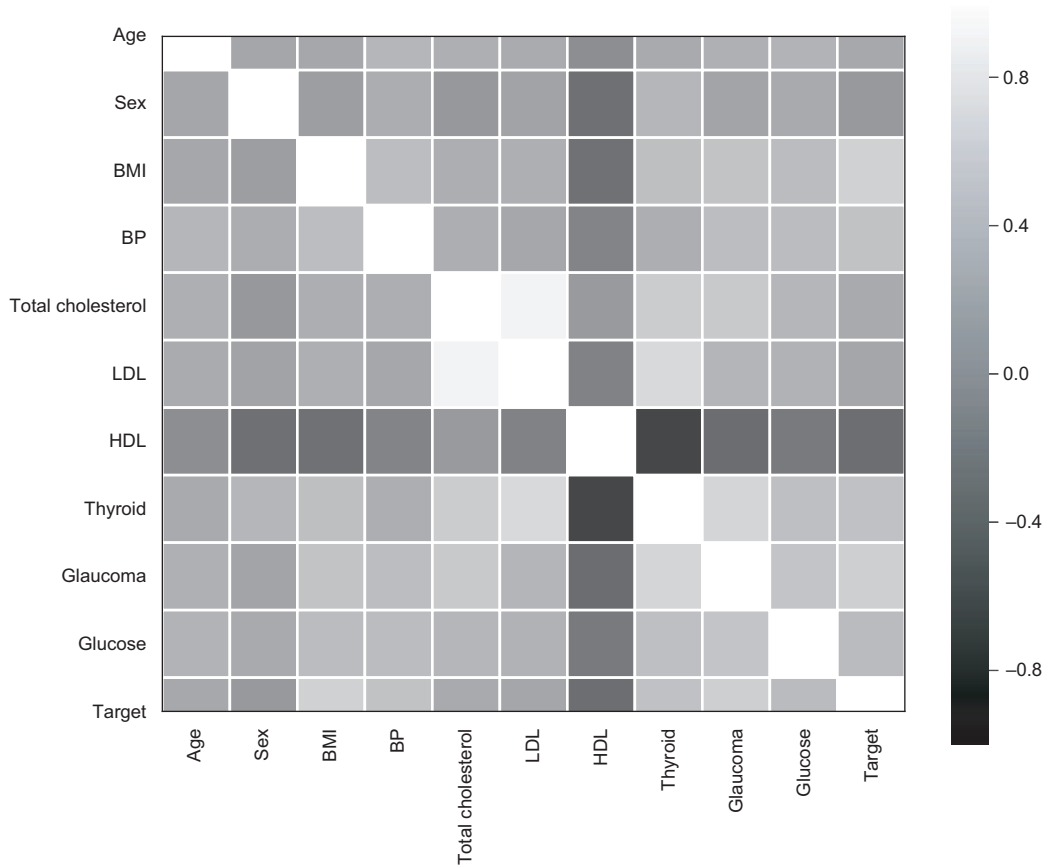


Figure 2.4 Correlation plot of the features and the target variable for the diabetes dataset

2.3 Linear regression

Linear regression is one of the simplest models you can train for regression tasks. In linear regression, the function f is represented as a linear combination of all the input features, as depicted in figure 2.5. The known variables are shown in gray, and the idea is to represent the target variable as a linear combination of the inputs. The unknown variables are the weights that must be learned by the learning algorithm.

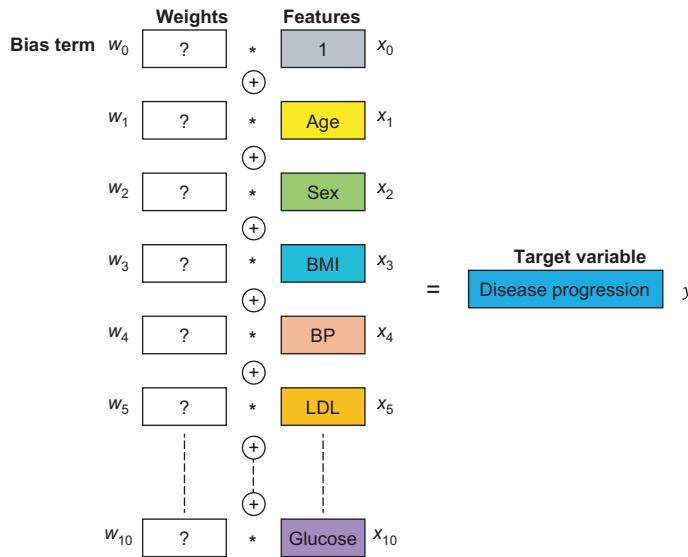


Figure 2.5 Disease progression represented as a linear combination of inputs

In general, the function f for linear regression is shown mathematically as follows, where n is the total number of features:

$$y = w_0 + w_1x_1 + w_2x_2 + \dots + w_nx_n$$

$$= w_0 + \sum_{i=1}^n w_ix_i$$

The objective of the linear regression learning algorithm is to determine the weights that accurately predict the target variable for all patients in the training set. We can apply the following techniques here:

- Gradient descent
- Closed-form solution (e.g., the Newton equation)

Gradient descent is commonly applied because it scales well to a large number of features and training examples. The general idea is to update the weights such that the squared error of the predicted target variable with respect to the actual target variable is minimized.

The objective of the gradient descent algorithm is to minimize the squared error or squared difference between the predicted target variable and the actual target variable across all the examples in the training set. This algorithm is guaranteed to find the optimum set of weights, and because the algorithm minimizes the squared error, it is said to be based on least squares. A linear regression model can be easily trained using the Scikit-Learn package in Python. The code to train the model is shown next. Note that the open diabetes dataset provided by Scikit-Learn is used here, and this dataset has been standardized, having zero mean and unit variance for all the input

features. Feature standardization is a widely used form of preprocessing done on data-sets used in many machine learning models like linear regression, logistic regression, and more complex models based on neural networks. It allows the learning algorithms that drive these models to converge faster to an optimum solution:

Imports numpy to evaluate the performance of model

Imports the scikit-learn class for linear regression

Imports the scikit-learn function to split the data into training and test sets

```

> from sklearn.model_selection import train_test_split
> from sklearn.linear_model import LinearRegression
> import numpy as np

```

```

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

```

Splits the data into training and test sets, where 80% of the data is used for training and 20% of the data for testing, and ensures that the seed for the random-number generator is set using the `random_state` parameter to ensure consistent train-test splits

```
lr_model = LinearRegression()
```

Initializes the linear regression model, which is based on least squares

```
lr_model.fit(X_train, y_train)
```

Learns the weights for the model by fitting on the training set

```
y_pred = lr_model.predict(X_test)
```

```
> mae = np.mean(np.abs(y_test - y_pred))
```

Uses the learned weights to predict the disease progression for patients in the test set

Evaluates the model performance using the mean absolute error (MAE) metric

The performance of the trained linear regression model can be quantified by comparing the predictions with the actual values on the test set. We can use multiple metrics, such as root mean squared error (RMSE), mean absolute error (MAE), and mean absolute percentage error (MAPE). Each of these metrics offers pros and cons, and it helps to quantify the performance using multiple metrics to measure the goodness of a model. Both MAE and RMSE are in the same units as the target variable and are easy to understand in that regard. The magnitude of the error, however, cannot be easily understood using these two metrics. For example, an error of 10 may seem small at first, but if the actual value you are comparing with is, say, 100, then that error is not small in relation to that. This is where MAPE is useful for understanding these relative differences because the error is expressed in terms of percentage (%) error. The topic of measuring model goodness is important but is beyond the scope of this book. You can find a lot of resources online. I have written a comprehensive two-part blog post (<http://mng.bz/ZzNP>) to cover this topic.

The previous trained linear regression model was evaluated using the MAE metric, and the performance was determined to be 42.8. But is this performance good? To check whether the performance of a model is good, we need to compare it with a baseline. For Diagnostics+, the doctors have been using a baseline model that predicts the median diabetes progression across all patients. The MAE of this baseline model

was determined to be 62.2. If we now compare this baseline with the linear regression model, we notice a drop in MAE by 19.4, which is a pretty good improvement. We have now trained a decent model, but it doesn't tell us how the model arrived at the prediction and which input features are most important. I cover this in the following section.

2.3.1 *Interpreting linear regression*

In the earlier section, we trained a linear regression model during model development and then evaluated the model performance during testing using the MAE metric. As a data scientist building Diagnostics+ AI, you now share these results with the doctors, and they are reasonably happy with the performance. But there is something missing. The doctors don't have a clear understanding of how the model arrived at the final prediction. Explaining the gradient descent algorithm does not help with this understanding because you are dealing with a pretty large feature space in this example—10 input features in total. It is impossible to visualize how the algorithm converges to the final prediction in a 10-dimensional space. In general, the ability to describe and explain a machine learning algorithm does not guarantee interpretability. So, what is the best way of interpreting a model?

For linear regression, because the final prediction is just a weighted sum of the input features, all we have to look at are the learned weights. This is what makes linear regression a white-box model. What do the weights tell us? If the weight of a feature is positive, a positive change in that input will result in a proportional positive change in the output, and a negative change in the input will result in a proportional negative change in the output. Similarly, if the weight is negative, a positive change in the input will result in a proportional negative change in the output, and a negative change in the input will result in a proportional positive change in the output. Such a learned function, shown in figure 2.6, is called a linear, monotonic function.

We can also look at the impact or importance of a feature in predicting the target variable by looking at the absolute value of the corresponding weight. The larger the

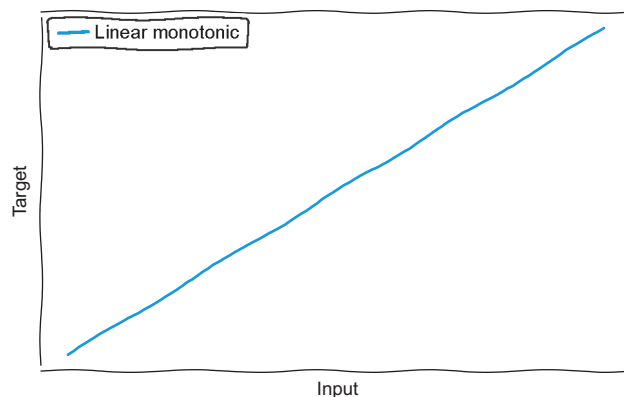


Figure 2.6 A representation of a linear, monotonic function

absolute value of the weight, the greater the importance. The weights for each of the 10 features are shown in descending order of importance in figure 2.7.

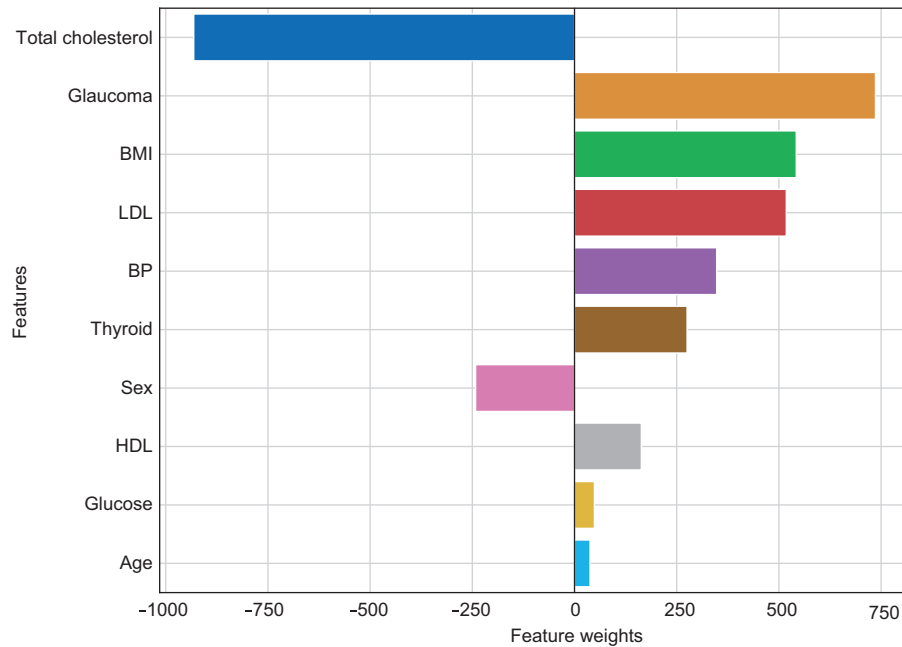


Figure 2.7 Feature importance for the diabetes linear regression model

The most important feature is the Total Cholesterol measurement. It has a large negative value for the weight. This means that a positive change in the cholesterol level has a large negative influence on predicting diabetes progression. This could be because Total Cholesterol also accounts for the good kind of cholesterol.

If we now look at the bad cholesterol, or LDL, feature, it has a large positive weight, and it is also the fourth most important feature in predicting the progression of diabetes. This means that a positive change in LDL cholesterol level results in a large positive influence in predicting diabetes one year out. The good cholesterol, or HDL, feature has a small positive weight and is the third least important feature. Why is that? Recall the exploratory analysis that we did in section 2.2 where we plotted the correlation matrix in figure 2.4. If we observe the correlation among total cholesterol, LDL, and HDL, we see a very high correlation between total cholesterol and LDL and moderately high correlation between total cholesterol and HDL. Because of this correlation, the HDL feature is deemed redundant by the model.

It also looks like the baseline Glucose measurement for the patient has a very small impact on predicting the progression of diabetes a year out. If we again go back to the correlation plot shown in figure 2.4, we can see that Glucose measurement is very highly correlated with the baseline Glaucoma measurement (the second most

important feature for the model) and highly correlated with Total Cholesterol (the most important feature for the model). The model, therefore, treats Glucose as a redundant feature because a lot of the signal is obtained from the Total Cholesterol and Glaucoma features.

If an input feature is highly correlated with one or more other features, they are said to be multicollinear. *Multicollinearity* could be detrimental to the performance of a linear regression model based on least squares. Let's suppose we use two features, x_1 and x_2 , to predict the target variable y . In a linear regression model, we are essentially estimating weights for each of the features that will help predict the target variable such that the squared error is minimized. Using least squares, the weight for feature x_1 , or the effect of x_1 on the target variable y , is estimated by holding x_2 constant. Similarly, the weight for x_2 is estimated by holding x_1 constant. If x_1 and x_2 are collinear, then they vary together, and it becomes very difficult to accurately estimate their effects on the target variable. One of the features becomes completely redundant for the model. We saw the effects of collinearity on our diabetes model earlier where features such as HDL and Glucose that were pretty highly correlated with the target variable had very low importance in the final model. The problem of multicollinearity can be overcome by removing the redundant features for the model. As an exercise, I highly recommend doing that to see if you can improve the performance of the linear regression model.

In the process of training a machine learning model, it is important to explore the data first and determine how correlated features are with each other and with the target variable. The problem of multicollinearity must be uncovered early in the process, before training the model, but if it has been overlooked, interpreting the model will help expose such issues. The plot in figure 2.7 can be generated in Python using the following code snippet:

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
sns.set(style='whitegrid')
sns.set_palette('bright')

weights = lr_model.coef_

feature_importance_idx = np.argsort(np.abs(weights))[:,::-1]
feature_importance = [feature_names[idx].upper() for idx in
    feature_importance_idx]
feature_importance_values = [weights[idx] for idx in
    feature_importance_idx]

f, ax = plt.subplots(figsize=(10, 8))
sns.barplot(x=feature_importance_values, y=feature_importance, ax=ax)
ax.grid(True)
ax.set_xlabel('Feature Weights')
ax.set_ylabel('Features')
```

Imports numpy to perform operation on vectors in an optimized way

Imports matplotlib and seaborn to plot the feature importance

Obtains the weights from the linear regression model trained earlier using the coef_ parameter

Sorts the weights in descending order of importance and gets their indices

Uses the ordered indices to get the feature names and the corresponding weight values

Generates the plot shown in figure 2.7

2.3.2 Limitations of linear regression

In the previous section, we saw how easy it is to interpret a linear regression model. It is highly transparent and easy to understand. However, it has poor predictive power, especially in cases where the relationship between the input features and target is nonlinear. Consider the example shown in figure 2.8.

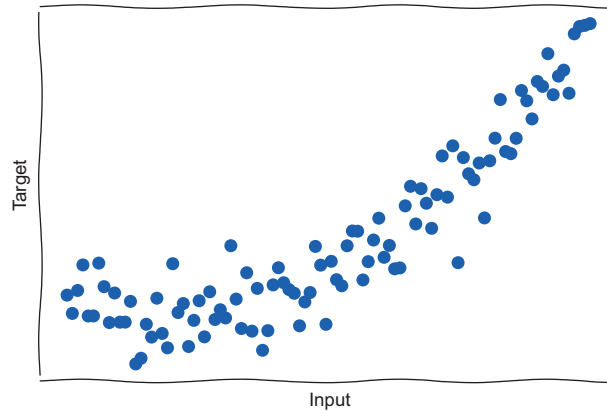


Figure 2.8 Illustration of a nonlinear dataset

If we were to fit a linear regression model to this dataset, we would get a straight-line linear fit, as shown in figure 2.9. As you can see, the model does not properly fit the data and does not capture the nonlinear relationship. This limitation of linear regression is called *underfitting*, and the model is said to have *high bias*. In the following sections, we will see how this problem can be overcome by using more complex models with higher predictive power.

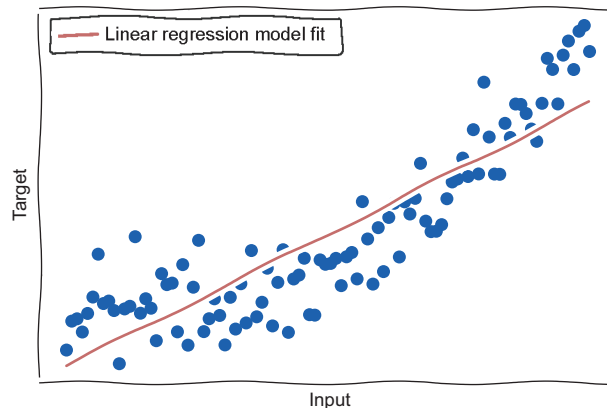


Figure 2.9 The problem of underfitting (high bias)

2.4 Decision trees

A decision tree is a great machine learning algorithm that can be used to model complex nonlinear relationships. It can be applied to both regression and classification tasks. It has relatively higher predictive power than linear regression and is highly

interpretable, too. The basic idea behind a decision tree is to find optimum splits in the data that best predict the output or target variable. In figure 2.10, I have illustrated this by considering only two features, BMI and Age. The decision tree splits the dataset into five groups in total, three age groups and two BMI groups.

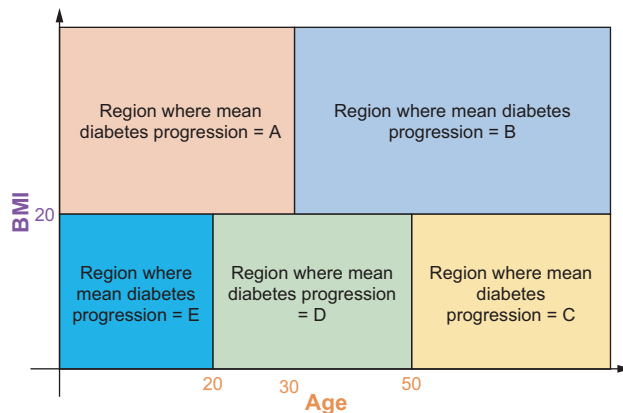


Figure 2.10 Decision tree splitting strategy

The algorithm that is commonly applied in determining the optimum splits is the classification and regression tree (CART) algorithm. This algorithm first chooses a feature and a threshold for that feature. Based on that feature and threshold, the algorithm splits the dataset into the following two subsets:

- Subset 1, where the value of the feature is less than or equal to the threshold
- Subset 2, where the value of the feature is greater than the threshold

The algorithm picks the feature and threshold that minimizes a cost function or criterion. For regression tasks, this criterion is typically the mean squared error (MSE), and for classification tasks, it is typically either Gini impurity or entropy. The algorithm then continues to recursively split the data until the criterion is reduced further or until a maximum depth is reached. The splitting strategy in figure 2.10 is shown as a binary tree in figure 2.11.

A decision tree model can be trained in Python using the Scikit-Learn package as follows. The code to learn the open diabetes dataset and to split it into the training and test sets is the same as the one used for linear regression in section 2.3, so, this code is not repeated here:

Trains the decision tree model ↳	<pre>from sklearn.tree import DecisionTreeRegressor</pre>	Imports the scikit-learn class for the decision tree regressor	Initializes the decision tree regressor. It is important to set the <code>random_state</code> to ensure that consistent, reproducible results can be obtained.
	<pre>dt_model = DecisionTreeRegressor(max_depth=None, random_state=42)</pre>	Uses the trained decision tree model to predict the disease progression for patients in the test set	
	<pre>dt_model.fit(X_train, y_train)</pre>	Evaluates the model performance using the mean absolute error (MAE) metric	
	<pre>y_pred = dt_model.predict(X_test)</pre>		
	<pre>mae = np.mean(np.abs(y_test - y_pred))</pre>		

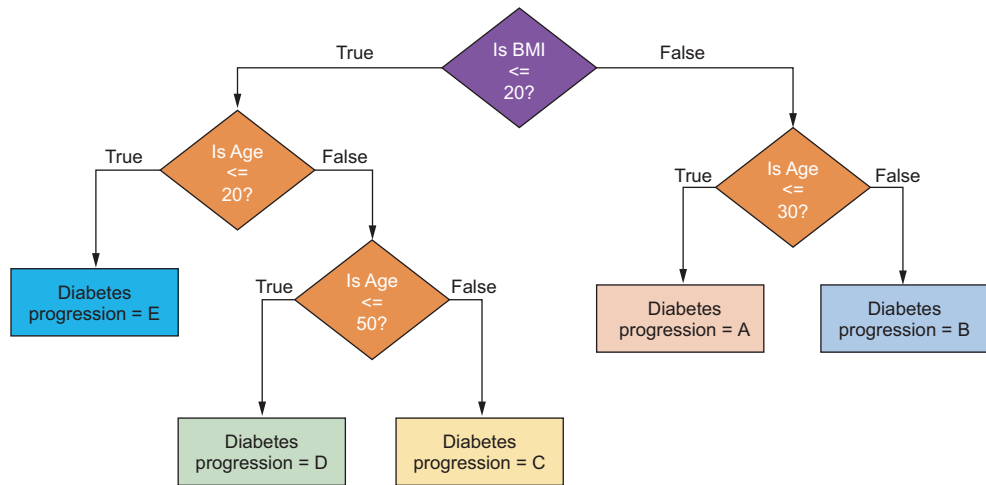


Figure 2.11 Decision tree data splitting visualized as a binary tree

The decision tree model trained here was evaluated using the MAE metric, and the performance was determined to be 54.7. If we tune the *max_depth* hyperparameter and set it to 3, we can improve the MAE performance further to 48.6. This performance, however, is poorer than the regression model trained in section 2.2. I will discuss the reasons for this in section 2.4.2, but first, let's look at how to interpret a decision tree in the following section.

Decision tree for classification tasks

As mentioned in this section, decision trees can also be used for classification tasks. In the CART algorithm, Gini impurity or entropy is used as the cost function. In Scikit-Learn, you can easily train a decision tree classifier as follows:

```

from sklearn.tree import DecisionTreeClassifier
dt_model = DecisionTreeClassifier(criterion='gini', max_depth=None)
dt_model.fit(X_train, y_train)

```

The *criterion* parameter in the *DecisionTreeClassifier* can be used to specify the cost function for the CART algorithm. By default, it is set to *gini*, but it can be changed to *entropy*.

2.4.1 Interpreting decision trees

Decision trees are great at modeling nonlinear relationships between the input and the output. By finding splits in the data across features, the model tends to learn a function that is nonlinear in nature. The function could be monotonic, where a change in the input results in a change in the output in the same direction, or non-

monotonic, where a change in the input could result in a change in the output in any direction and at a varying rate. This is illustrated in figure 2.12.

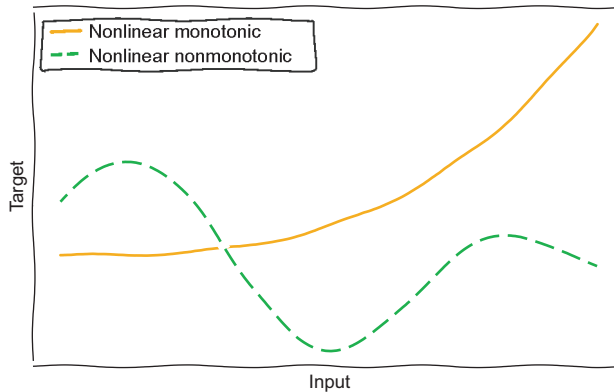


Figure 2.12 Representation of nonlinear, monotonic, and nonmonotonic functions

How do we interpret such a learned nonlinear function? As seen in the previous section, a decision tree can be visualized as a bunch of if-else conditions strung together, where each condition splits the data in two. Such a model can be easily visualized as a binary tree, as illustrated in figure 2.11. For the decision tree model trained for diabetes, the visualization of the binary tree is shown in figure 2.13. The tree can be interpreted as follows.

Starting at the root of the tree, check if the normalized BMI is ≤ 0 . If true, go to the left part of the tree. If false, go to the right part of the tree. Because we are starting at the root of the tree, this node accounts for 100% of the data. This is why *samples* is equal to 100%. Also, if we were to set the *max_depth* to 0 and predict the disease progression, then we would use the average value of all the samples in the data, which is 153.7, represented as *value* in the tree. By predicting 153.7, we would get an MSE of 6076.4.

If the normalized BMI is ≤ 0 , then we go to the left part of the tree and check if the normalized Glaucoma is ≤ 0 . If BMI is ≤ 0 , we would account for approximately 59% of the data, and the MSE would reduce from 6076.4 for the parent node to 3612.7. We can repeat this process until we have reached the leaf nodes in the tree. If we look at, say, the right-most leaf node, this corresponds to the following condition: if BMI > 0 and BMI > 0.1 and LDL > 0 , then predict 225.8 for 2.3% of the data, resulting in an MSE of 2757.9.

Please note that the *max_depth* for the decision tree in figure 2.13 was set to 3. The complexity of this tree will increase as *max_depth* increases or as the number of input features increases.

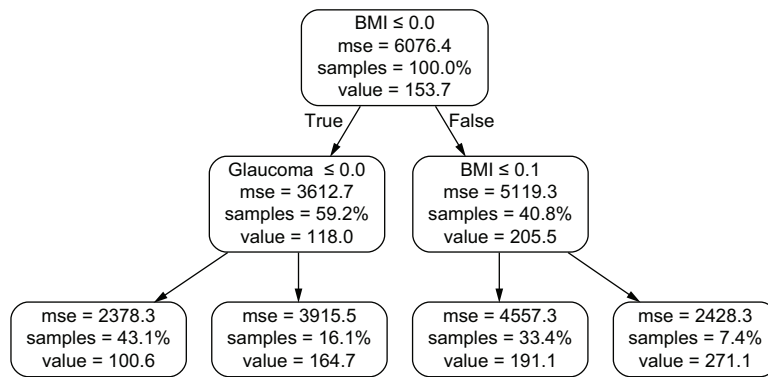


Figure 2.13 Visualization of the diabetes decision tree model

The visualization in figure 2.13 can be generated in Python using the following code snippet:

```

from sklearn.externals.six import StringIO
from IPython.display import Image
from sklearn.tree import export_graphviz
import pydotplus

diabetes_dt_dot_data = StringIO()
export_graphviz(dt_model,
                out_file=diabetes_dt_dot_data,
                filled=False, rounded=True,
                feature_names=feature_names,
                proportion=True,
                precision=1,
                special_characters=True)
dt_graph = pydotplus.graph_from_dot_data(diabetes_dt_dot_data.getvalue())
Image(dt_graph.create_png())

```

Imports all the necessary libraries to generate and visualize the binary tree

Initializes a string buffer to store the binary tree/graph in DOT format

Exports the decision tree model as a binary tree in DOT format

Generates an image of the binary tree using the DOT format string

Visualizes the binary tree using the Image class

Because decision trees learn a nonlinear relationship between the input features and the target, it is hard to understand what effects changes to each of the inputs have on the output. It is not as straightforward as linear regression. We can, however, compute the relative importance of each of the features in predicting the target at a global level. To compute the feature importance, we first need to compute the importance of a node in the binary tree. The importance of a node is computed as the decrease in the cost function or impurity measure for that node weighted by the probability of reaching that node in the tree. This is shown mathematically next:

$$\underbrace{I_k^{\text{node}}}_{\text{Importance of node } k} = \underbrace{p_k}_{\text{Proportion of samples to reach node } k} \cdot \underbrace{m_k}_{\text{Impurity measure of node } k} - \underbrace{p_k^{(\text{left})}}_{\text{Proportion of samples to reach left subtree of node } k} \cdot \underbrace{m_k^{(\text{left})}}_{\text{Impurity measure of left subtree of node } k} - \underbrace{p_k^{(\text{right})}}_{\text{Proportion of samples to reach right subtree of node } k} \cdot \underbrace{m_k^{(\text{right})}}_{\text{Impurity measure of right subtree of node } k}$$

We can then compute the feature importance by summing up the importance of the nodes that split on that feature normalized by the importance of all the nodes in the tree. This is shown mathematically next. The feature importance for the decision tree is between 0 and 1, where a higher value implies greater importance:

$$\underbrace{I_i^{\text{feature}}}_{\text{Importance of feature } i} = \frac{\overbrace{\sum_{j \in \mathbb{J}} I_j^{\text{node}}}^{\text{Sum of importance of all nodes } j \text{ that split on feature } i}}{\underbrace{\sum_{k \in \mathbb{K}} I_k^{\text{node}}}^{\text{Sum of importance of all nodes } k \text{ in the decision tree}}}$$

In Python, the feature importance can be obtained from the Scikit-Learn decision tree model and plotted as follows:

Gets feature importance from the trained decision tree model

```
weights = dt_model.feature_importances_
```

Sorts indices of feature weights in descending order of importance

```
feature_importance_idx = np.argsort(np.abs(weights))[:,::-1]
feature_importance = [feature_names[idx].upper() for idx in
    feature_importance_idx]
feature_importance_values = [weights[idx] for idx in
    feature_importance_idx]
```

Gets the feature names and feature weights in descending order of importance

```
f, ax = plt.subplots(figsize=(10, 8))
sns.barplot(x=feature_importance_values, y=feature_importance, ax=ax)
ax.grid(True)
ax.set_xlabel('Feature Weights')
ax.set_ylabel('Features')
```

Generates the plot shown in figure 2.14

The features ordered in descending order of importance and their corresponding weights are shown in figure 2.14. As can be seen from the figure, the order of important features is different from linear regression. The most important feature is BMI,

accounting for roughly 42% of the overall model importance. The Glaucoma measurement is the next most important feature, accounting for roughly 15% of the model importance. These importance values are useful in determining what features have the most signal in predicting the target variable. Decision trees are immune to the problem of multicollinearity because the algorithm picks the feature that is highly correlated with the target and that most reduces the cost function or impurity. As a data scientist, it is important to visualize the learned decision tree, as shown in figure 2.13, because this will help you understand how the model arrived at the final prediction. You could reduce the complexity of the tree by setting the *max_depth* hyperparameter or by pruning the number of features you feed into the model. You can determine what features to prune by visualizing the global feature importance, as shown in figure 2.14.

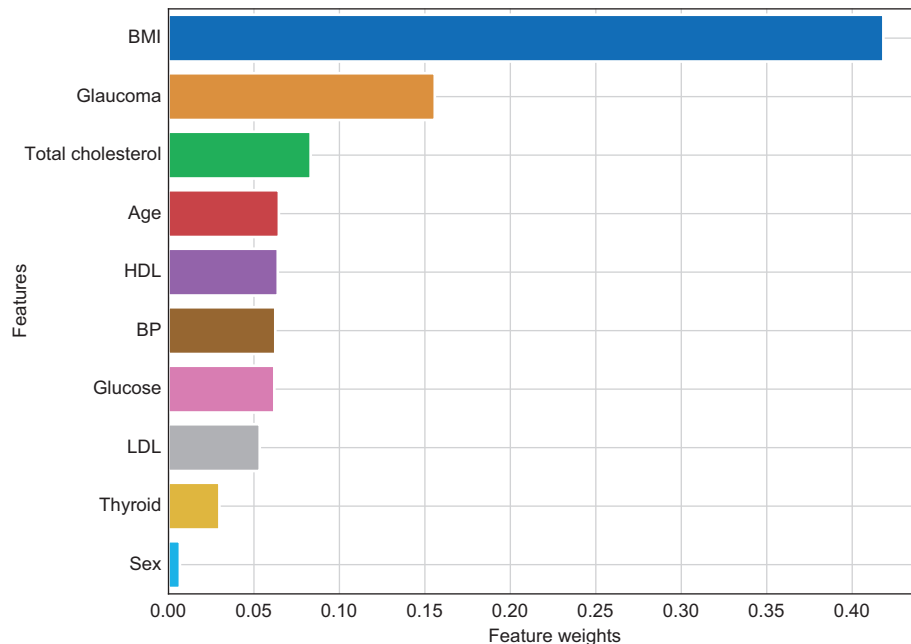


Figure 2.14 Diabetes feature importance for decision tree

2.4.2 Limitations of decision trees

Decision trees are quite versatile because they can be applied to both regression and classification tasks, and they also have the ability to model nonlinear relationships. The algorithm, however, is prone to the problem of *overfitting* and the model is said to have *high variance*.

The problem of overfitting occurs when the model fits the training data almost perfectly and, therefore, does not generalize well to data that it hasn't seen before,

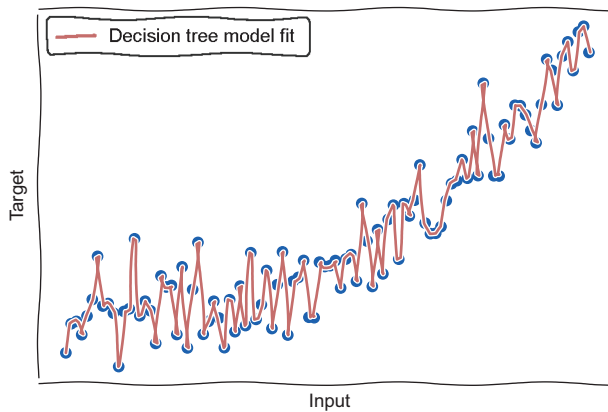


Figure 2.15 The problem of overfitting (high variance)

such as the test set. This is illustrated in figure 2.15. When a model overfits, you will notice really good performance on the training set but poor performance on the test set. This could explain why the decision tree model trained on the diabetes dataset performed poorer than the linear regression model.

The problem of overfitting can be overcome by tuning certain hyperparameters in the decision tree, like *max_depth*, and the minimum number of samples required for the leaf nodes. As shown in the visualization of the decision tree model in figure 2.13, one leaf node accounts for only 0.8% of the samples. This means that the prediction for this node is based on the data from roughly only three patients. By increasing the minimum number of samples required to 5 or 10, we could improve the performance of the model on the test set.

2.5 Generalized additive models (GAMs)

Diagnostics+ and the doctors are reasonably happy with the two models built so far, but the performance is not that good. By interpreting the models, we have also uncovered some shortcomings. The linear regression model does not seem to handle features that are highly correlated with each other, such as Total Cholesterol, LDL, and HDL. The decision tree model performs worse than linear regression, and it seems to have overfit on the training data.

Let's look at one specific feature from the diabetes data. Figure 2.16 shows a contrived example of a nonlinear relationship between age and the target variable, where both variables are normalized. How would you best model this relationship without overfitting? One possible approach is to extend the linear regression model where the target variable is modeled as an n^{th} degree polynomial of the feature set. This form of regression is called *polynomial regression*.

Polynomial regression for various-degree polynomials is shown in the following equations. In these equations, we are considering only one feature, x_1 , to model the target variable y . The degree 1 polynomial is the same as linear regression. For the degree 2 polynomial, we would add an additional feature, which is the square of x_1 .

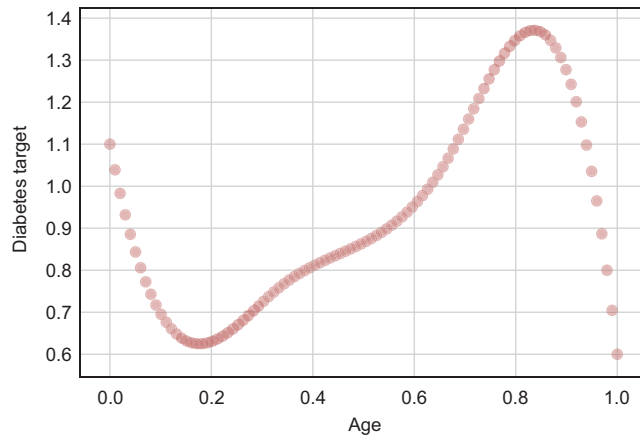


Figure 2.16 An illustration of a nonlinear relationship for Diagnostics+ AI

For the degree 3 polynomial, we would add two additional features—one that is the square of x_1 and the other that is the cube of x_1 :

$$y = w_0 + w_1x_1 \text{ (Degree 1)}$$

$$y = w_0 + w_1x_1 + w_2x_1^2 \text{ (Degree 2)}$$

$$y = w_0 + w_1x_1 + w_2x_1^2 + w_3x_1^3 \text{ (Degree 3)}$$

The weights for the polynomial regression model can be obtained using the same algorithm as linear regression, that is, the method of least squares using gradient descent. The best fit learned by each of the three polynomials is shown in figure 2.17.

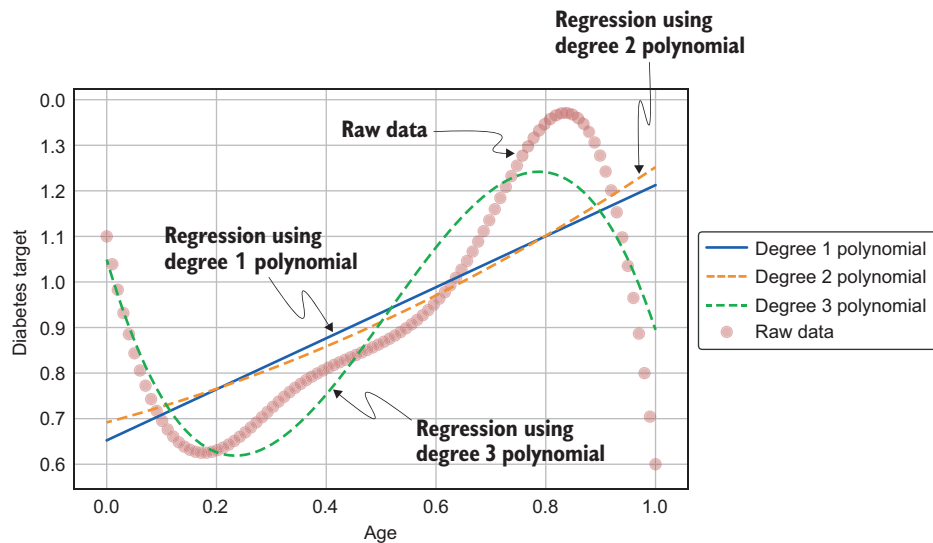


Figure 2.17 Polynomial regression for modeling a nonlinear relationship

We can see that the degree 3 polynomial fits the raw data better than degrees 2 and 1. We can interpret a polynomial regression model the same way as we would a linear regression because the model is essentially a linear combination of the features including the higher degree features.

Polynomial regression has some limitations, however. The complexity of the model increases as the number of features or the dimension of the feature space increases. It, therefore, has a tendency to overfit on the data. It is also hard to determine the degree for each feature in the polynomial, especially in a higher-dimensional feature space.

So, what model can be applied to overcome all these limitations and is also interpretable? Enter generalized additive models (GAMs)! GAMs are models with medium to high predictive power and are highly interpretable. Nonlinear relationships are modeled by using smoothing functions for each feature and adding all of them, as shown in the following equation:

$$y = w_0 + \underbrace{f_1(x_1)}_{\text{Smoothing Function for Feature } x_1} + \underbrace{f_2(x_2)}_{\text{Smoothing Function for Feature } x_2} + \dots + \underbrace{f_n(x_n)}_{\text{Smoothing Function for Feature } x_n}$$

In this equation, each feature has its own associated smoothing function that best models the relationship between that feature and the target. You can choose from many types of smoothing functions, but a widely used smoothing function is called *regression splines* because it is practical and computationally efficient. I will focus on regression splines in this book. Let's now go deep into the world of GAMs using regression splines!

2.5.1 Regression splines

Regression splines are represented as a weighted sum of basis functions. This is shown mathematically in the next equation. In this equation, f_j is the function that models the relationship between the feature x_j and the target variable. This function is represented as a weighted sum of basis functions where the weight is represented as w_k and the basis function is represented as b_k . In the context of GAMs, the function f_j is called a smoothing function.

$$f_j(x_j) = \underbrace{\sum_{k=1}^K w_k b_k(x_j)}_{\text{Smoothing Function represented as a weighted sum of basis functions}}$$

Now, what is a basis function? A basis function is a family of transformations that can be used to capture a general shape or nonlinear relationship. For regression splines, as the name suggests, splines are used as the basis function. A spline is a polynomial of

degree n with $n - 1$ derivatives. It will be much easier to understand splines using an illustration. Figure 2.18 shows splines of various degrees. The top-left graph shows the simplest spline of degree 0, from which higher degree splines can be generated. As you can see from the top-left graph, six splines have been placed on a grid. The idea is to split the distribution of the data into portions and fit a spline on each of those portions. So, in this illustration, the data has been split into six portions, and we are modeling each portion as a degree 0 spline.

A degree 1 spline, shown in the top-right graph, can be generated by convolving a degree 0 spline with itself. Convolution is a mathematical operation that takes in two functions and creates a third function that represents the correlation of the first function and a delayed copy of the second function. When we convolve a function with itself, we are essentially looking at the correlation of the function with a delayed copy of itself. There is a nice blog post by Christopher Olah on convolutions (<http://mng.bz/5Kdq>). By convolving a degree 0 spline with itself, we get a degree 1 spline, which is triangular, and this has a continuous 0th-order derivative.

If we now convolve a degree 1 spline with itself, we will get a degree 2 spline, shown in the bottom-left graph. This degree 2 spline has a first-order derivative. Similarly, we can get a degree 3 spline by convolving a degree 2 spline, and this has a second-order derivative. In general, a degree n spline has an $n - 1$ derivative. In the limit, as n

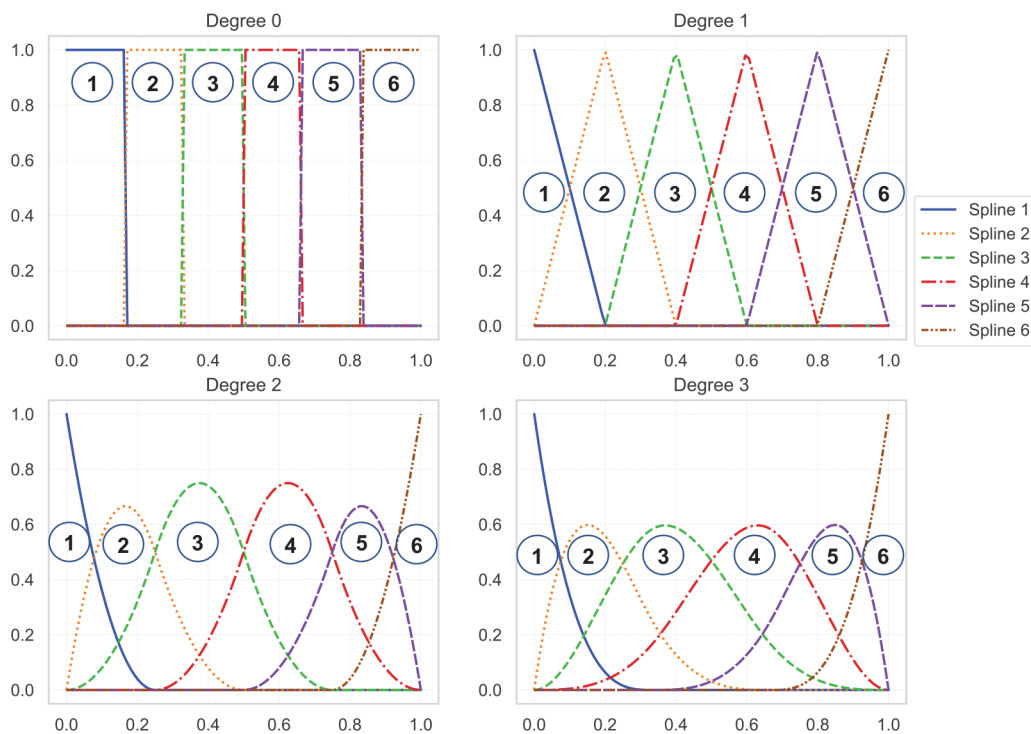


Figure 2.18 An illustration of degree 0, degree 1, degree 2, and degree 3 splines

approaches infinity, we will obtain a spline that has the shape of a Gaussian distribution. In practice, a *degree 3 spline*, or *cubic spline*, is used because it can capture most general shapes.

As mentioned earlier, in figure 2.18, we have divided the distribution of data into six portions and have placed six splines on the grid. In the earlier mathematical equation, the number of portions or splines was represented as variable K . The idea behind regression splines is to learn the weights for each of the splines so that you can model the distribution of the data in each of the portions. The number of portions or splines in the grid, K , is also called *degrees of freedom*. In general, if we place these K splines on a grid, we will have $K + 3$ points of division, also known as *knots*.

Let's now zoom in on cubic splines, as shown in figure 2.19. We can see that there are six splines, or six degrees of freedom, resulting in nine points of division or knots.

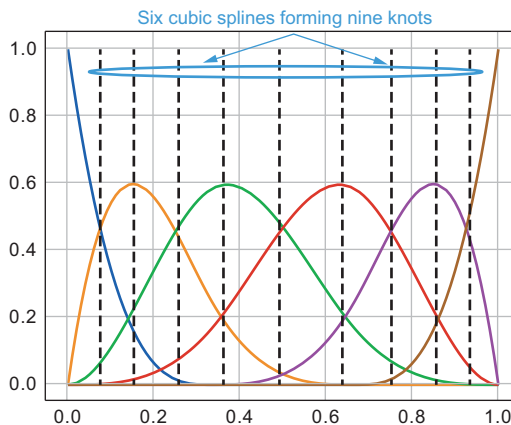


Figure 2.19 An illustration of splines and knots

To capture a general shape, we need to take a weighted sum of the splines. We will use cubic splines here. In figure 2.20, we are using the same six splines overlaid to create nine knots. For the graph on the left, I have set the same weights for all six splines. As you can imagine, if we take an equally weighted sum of all six splines, we will get a horizontal straight line. This is an illustration of a poor fit to the raw data. For the graph on the right, however, I have taken an unequal weighted sum of the six splines generating a shape that perfectly fits the raw data. This shows the power of regression splines and GAMs. By increasing the number of splines or by dividing the data into more portions, we can model more complex nonlinear relationships. In GAMs based on regression splines, we individually model nonlinear relationships of each feature with the target variable and then add them all up to come up with the final prediction.

In figure 2.20, the weights were determined using trial and error to best describe the raw data. But, how do you algorithmically determine the weights for a regression spline that best captures the relationship between the features and the target? Recall from the start of this section that a regression spline is a weighted sum of basis functions or splines. This is essentially a linear regression problem, and you can learn the

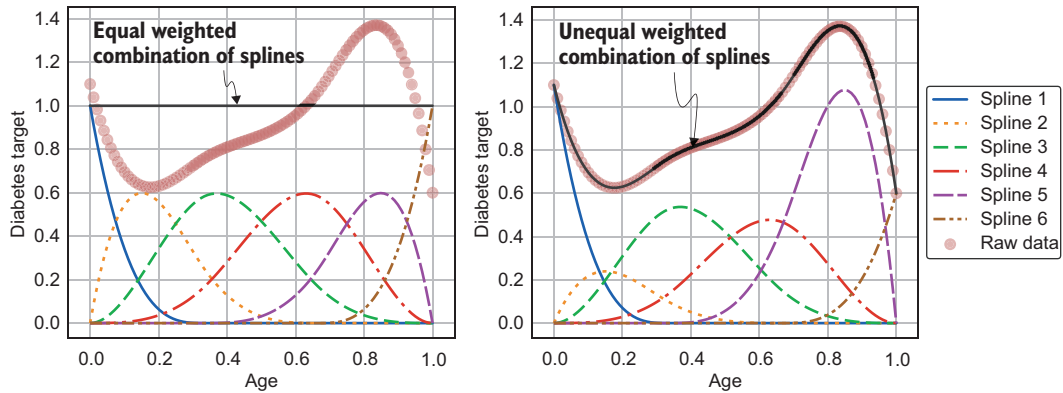


Figure 2.20 Splines for modeling a nonlinear relationship

weights using the method of least squares and gradient descent. We would, however, need to specify the number of knots, or degrees of freedom. We can treat this as a hyperparameter and determine it using a technique called *cross-validation*. Using cross-validation, we would remove a portion of the data and fit a regression spline with a certain number of predetermined knots on the remaining data. This regression spline is then evaluated on the held-out set. The optimum number of knots is the one that results in the best performance on the held-out set.

In GAMs, you can easily overfit by increasing the number of splines or degrees of freedom. If the number of splines is high, the resulting smoothing function, which is a weighted sum of the splines, would be quite “wiggly”—it would start to fit some of the noise in the data. How can we control this wiggleness or prevent overfitting? We can use a technique called *regularization*. In regularization, we would add a term to the least square cost function that quantifies the wiggleness. We could then quantify the wiggleness of a smoothing function by taking the integral of the square of the second-order derivative of the function. Then, using a hyperparameter (also called regularization parameter) represented by λ , we could adjust the intensity of wiggleness. A high value for λ penalizes wiggleness heavily. We can determine λ the same way we determine other hyperparameters using cross-validation.

Summary of GAMs

A GAM is a powerful model where the target variable is represented as a sum of smoothing functions representing the relationship of each of the features and the target. We can use the smoothing function to capture any nonlinear relationship. This is shown mathematically here:

$$y = w_0 + f_1(x_1) + f_2(x_2) + \dots + f_n(x_n)$$

(continued)

This is a white-box model—we can easily see how each feature is transformed to the output using the smoothing function. A common way of representing the smoothing function is by using regression splines. A regression spline is represented as a simple weighted sum of basis functions. A basis function that is widely used for GAMs is the cubic spline. By increasing the number of splines or degrees of freedom, we can divide the distribution of data into small portions and model each portion piecewise. This way, we can capture very complex nonlinear relationships. The learning algorithm essentially has to determine the weights for the regression spline. We can do this the same way as for linear regression, using the method of least squares and gradient descent. We can determine the number of splines using the cross-validation technique. As the number of splines increases, GAMs have a tendency to overfit on the data. We can safeguard against this by using the regularization technique. Using a regularization parameter λ , we can control the amount of wiggleness. A higher λ ensures a smoother function. The parameter λ can also be determined using cross-validation.

GAMs can also be used to model interactions between variables. GA2M, shown mathematically next, is a type of GAM that models pairwise interactions:

$$y = w_0 + f_1(x_1) + f_2(x_2) + \underbrace{f_3(x_1, x_2)}_{\substack{\text{Modeling interaction} \\ \text{between } x_1 \text{ and } x_2}} + f_4(x_4) + \dots + f_n(x_n)$$

With the help of subject matter experts (SMEs)—the doctors in the Diagnostics+ example—you can determine what feature interactions need to be modeled. You could also look at the correlation between features to understand what features need to be modeled together.

In Python, you can use a package called pyGAM to build and train GAMs. It is inspired by the GAM implementation in the popular mgcv package in R. You can install pyGAM in your Python environment using the pip package as follows:

```
pip install pygam
```

2.5.2 **GAM for Diagnostics+ diabetes**

Let's now go back to the Diagnostics+ example to train a GAM to predict diabetes progression using all 10 features. Note that the Sex of the patient is a categorical or discrete feature. It does not make sense to model this feature using a smoothing function. We can treat such categorical features in GAMs as factor terms. We can train the GAM using the pyGAM package as follows. As with decision trees, I'm not going to repeat the code that loads the diabetes dataset and splits it into the train and test sets. Please refer to section 2.2 for that snippet of code:


```

Imports the LinearGAM class from
pygam that can be used to train a
GAM for regression tasks
> from pygam import LinearGAM
from pygam import s
from pygam import f

Imports the smoothing term function
to be used for numerical features
Imports the factor term function to
be used for categorical features

# Load data using the code snippet in Section 2.2

Cubic spline term for the Age feature
> gam = LinearGAM(s(0) +
f(1) +
s(2) +
s(3) +
s(4) +
s(5) +
s(6) +
s(7) +
s(8) +
s(9),
n_splines=35)
Factor term for the Sex feature, which is categorical
Cubic spline term for the BP feature
Cubic spline term for the Total Cholesterol feature
Cubic spline term for the LDL feature
Cubic spline term for the HDL feature
Cubic spline term for the Thyroid feature
Cubic spline term for the Glucose feature
Maximum number of splines to be used for each feature

gam.gridsearch(X_train, y_train)
Uses grid search to perform training and cross-validation to determine the number of splines, the regularization parameter lambda, and the optimum weights for the regression splines for each feature

y_pred = gam.predict(X_test)
Uses the trained GAM model to predict on the test

mae = np.mean(np.abs(y_test - y_pred))
Evaluates the performance of the model on the test set using the MAE metric

```

Now for the moment of truth! How did the GAM perform? The MAE performance of the GAM is 41.4—a pretty good improvement when compared to the linear regression and decision tree models. A comparison of the performance of all three models is summarized in table 2.2. I have also included the performance of a baseline model that Diagnostics+ and the doctors have been using where they look at the median diabetes progression across all patients. All models are compared against the baseline to show how much of an improvement the models give to the doctors. It looks like GAM is the best model across all performance metrics.

Table 2.2 Performance comparison of linear regression, decision tree, and GAM against a baseline for Diagnostics+ A

	MAE	RMSE	MAPE
Baseline	62.2	74.7	51.6
Linear regression	42.8 (−19.4)	53.8 (−20.9)	37.5 (−14.1)
Decision tree	48.6 (−13.6)	60.5 (−14.2)	44.4 (−7.2)
GAM	41.4 (−20.8)	52.2 (−22.5)	35.7 (−15.9)

We have now seen the predictive power of GAMs. We could potentially get further improvement in the performance by modeling feature interactions, especially the cholesterol features with each other and with other features that are potentially highly correlated, like BMI. As an exercise, I encourage you to try modeling feature interactions using GAMs.

GAMs are white box and can be easily interpreted. In the following section, we will see how GAMs can be interpreted.

GAMs for classification tasks

GAMs can also be used to train a binary classifier by using the logistic link function where the response y can be either 0 or 1. In the pyGAM package, you can make use of the logistic GAM for binary classification problems as follows:

```
from pygam import LogisticGAM
gam = LogisticGAM()
gam.gridsearch(X_train, y_train)
```

2.5.3 Interpreting GAMs

Although each smoothing function is obtained as a linear combination of basis functions, the final smoothing function for each feature is nonlinear, and, therefore, we cannot interpret the weights the same way as we do for linear regression. We can, however, easily visualize the effects of each feature on the target using partial dependence or partial effects plots. Partial dependence looks at the effect of each feature by marginalizing on the rest. It is highly interpretable because we can see the average effect of each feature value on the target variable. We can see whether the target response to the feature is linear, nonlinear, monotonic, or nonmonotonic. Figure 2.21 shows the effect of each of the patient features on the target variable. The 95% confidence interval around them have also been plotted. This will help us determine the sensitivity of the model to data points with a low sample size.

Let's now look at a couple of features in figure 2.21, namely, BMI and BP. The effect of BMI on the target variable is shown in the bottom-left graph. On the x -axis, we see the normalized values of BMI, and on the y -axis, we see the effect that BMI has on the progression of diabetes for the patient. We see that as BMI increases, the effect on the progression of diabetes also increases. We see a similar trend for BP shown by the bottom-right graph. We see that the higher the BP, the higher the impact on the progression of diabetes. If we look at the 95% confidence interval lines (the dashed lines in figure 2.21), we see a wider confidence interval around the lower and higher ends of BMI and BP. This is because fewer samples of patients exist at this range of values, resulting in higher uncertainty in understanding the effects of these features at those ranges.

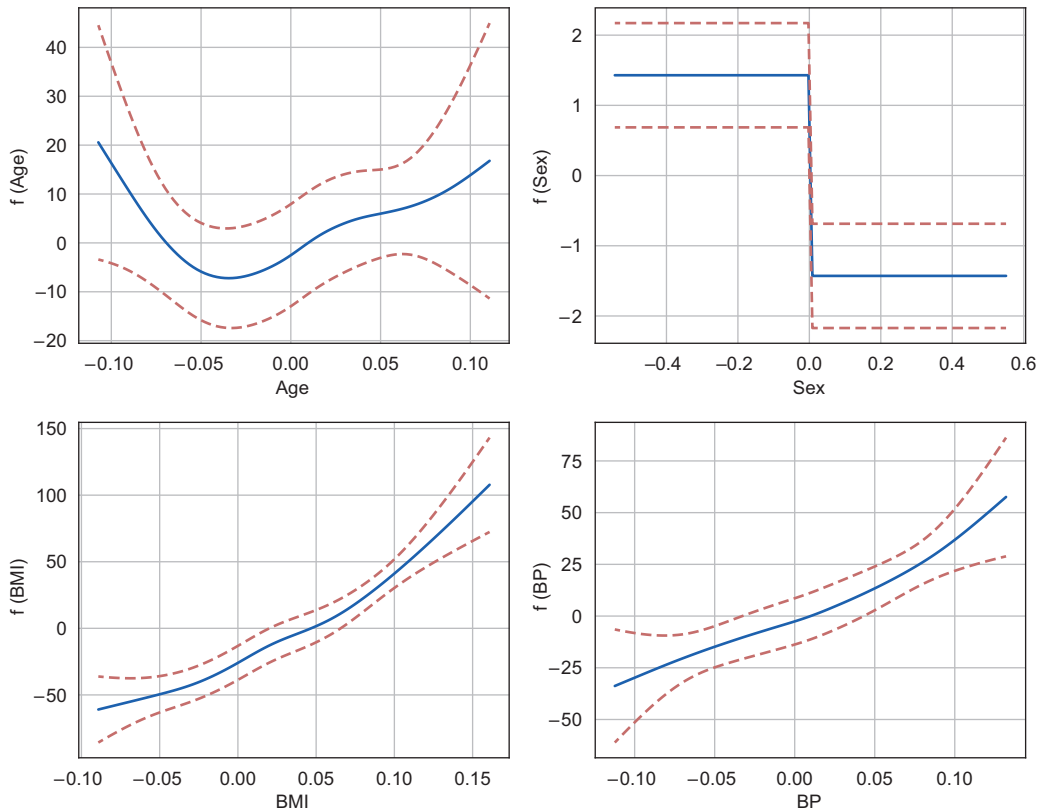


Figure 2.21 The effect of each of the patient features on the target variable

The code to generate figure 2.21 follows:

```

Locations of the four graphs in the 2x2 Matplotlib grid
grid_locs1 = [(0, 0), (0, 1),
               (1, 0), (1, 1)]

fig, ax = plt.subplots(2, 2, figsize=(10, 8))
for i, feature in enumerate(feature_names[:4]):
    gl = grid_locs1[i]  # Gets the location of feature in the 2x2 grid
    XX = gam.generate_X_grid(term=i)
    ax[gl[0], gl[1]].plot(XX[:, i], gam.partial_dependence(term=i, X=XX))
    ax[gl[0], gl[1]].plot(XX[:, i], gam.partial_dependence(term=i, X=XX,
    width=.95)[1], c='r', ls='--')
    ax[gl[0], gl[1]].set_xlabel('%s' % feature)
    ax[gl[0], gl[1]].set_ylabel('f ( %s )' % feature)

Plots the 95% confidence interval around the partial dependence values as a dashed line
Iterates through the four patient metadata features
Generates the partial dependence of the feature values with the target marginalizing on the other features
Plots the partial dependence values as a solid line
Adds labels for the x- and y-axes

```

Figure 2.22 shows the effect of each of the six blood test measurements on the target. As an exercise, observe the effects that features like Total Cholesterol, LDL, HDL, and Glaucoma have on the progression of diabetes. What can you say about the impact of

higher LDL values (or bad cholesterol) on the target variable? Why does higher Total Cholesterol have a lower impact on the target variable? To answer these questions, let's look at a few patient cases with very high cholesterol values. The following code snippet will help you zoom in on those patients:

```
print(df_data[(df_data['Total Cholesterol'] > 0.15) &
              (df_data['LDL'] > 0.19)])
```

If you execute this code, you will see only one patient out of 442 that has a Total Cholesterol reading greater than 0.15 and an LDL reading greater than 0.19. The fasting glucose level for this patient one year out (the target variable) seems to be 84, which is in the normal range. This could explain why in figure 2.22 we are seeing a very large negative effect for Total Cholesterol on the target variable for a range that is greater

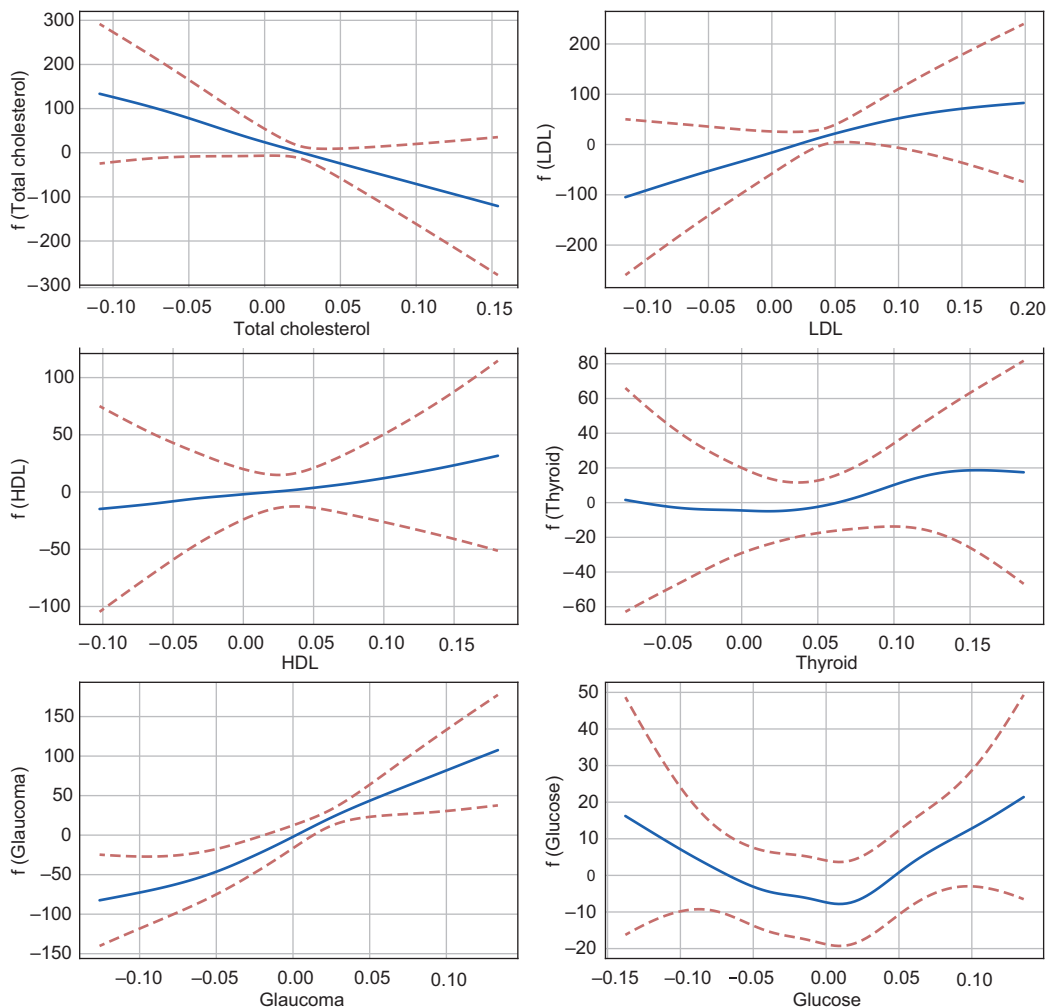


Figure 2.22 The effect of each of the blood test measurements on the target variable

than 0.15. The negative effect of Total Cholesterol seems to be greater than the positive effect the bad LDL cholesterol seems to have on the target. The confidence interval seems much wider in these range of values. The model may have overfit on this one outlier patient record, and so, we should not read too much into these effects. By observing these effects, we can identify cases or a range of values where the model is sure of the prediction and cases where there is high uncertainty. For high uncertainty cases, we can go back to the diagnostics center to collect more patient data so that we have a representative sample.

The code to generate figure 2.22 follows:

```

Iterates through the six blood
test measurement features
grid_locs2 = [(0, 0), (0, 1),
              (1, 0), (1, 1),
              (2, 0), (2, 1)]
Locations of the six graphs in
the 3 × 2 Matplotlib grid
fig2, ax2 = plt.subplots(3, 2, figsize=(12, 12))
Creates a 3 × 2 grid
of Matplotlib graphs
for i, feature in enumerate(feature_names[4:]):
    idx = i + 4
    Gets the location of the
    feature in the 3 × 2 grid
    gl = grid_locs2[i]
    XX = gam.generate_X_grid(term=idx)
    Generates the partial
    dependence of the feature
    values with the target
    marginalizing on the
    other features grid of
    Matplotlib graphs
    ax2[gl[0], gl[1]].plot(XX[:, idx], gam.partial_dependence(term=idx,
        Plots the partial
        dependence
        values as a
        solid line
        X=XX))
    ax2[gl[0], gl[1]].plot(XX[:, idx], gam.partial_dependence(term=idx, X=XX,
        width=.95)[1], c='r', ls='--')
    Plots the 95% confidence
    interval around the partial
    dependence values as a
    dashed line
    ax2[gl[0], gl[1]].set_xlabel('%s' % feature)
    Adds labels for the x- and y-axes
    ax2[gl[0], gl[1]].set_ylabel('f ( %s )' % feature)

```

Through figures 2.21 and 2.22, we can gain a much deeper understanding of the marginal effect of each of the feature values on the target. The partial dependence plots are useful for debugging any issues with the model. By plotting the 95% confidence interval around the partial dependence values, we can also see data points with low sample sizes. If a feature value with a low sample size has a dramatic effect on the target, then there could be an overfitting problem. We can also visualize the wiggleness of the smoothing function to determine whether the model has fit on the noise in the data. We can fix these overfitting problems by increasing the value of the regularization parameter. These partial dependence plots can also be shared with the SME—doctors, in this case—for validation which will help gain their trust.

2.5.4 Limitations of GAMs

We have so far seen the advantages of GAMs in terms of predictive power and interpretability. GAMs have a tendency to overfit, although this can be overcome with regularization. You do need to be aware of the following other limitations, however:

- GAMs are sensitive to feature values outside of the range in the training set and tend to lose predictive power when exposed to outlier values.
- For mission-critical tasks, GAMs may sometimes have limited predictive power, in which case you may need to consider more powerful black-box models.

2.6 Looking ahead to black-box models

Black-box models are models with really high predictive power and are typically applied in tasks for which model performance (such as accuracy) is extremely important. They are, however, inherently opaque, and the characteristics that make them opaque include the following:

- The machine learning process is complicated, and you can't easily understand how the input features are transformed into the output or target variable.
- You can't easily identify the most important features to predict the target variable.

Examples of black-box models are tree ensembles such as random forest and gradient-boosted trees, deep neural networks (DNNs), convolutional neural networks (CNNs), and recurrent neural networks (RNNs). Table 2.3 shows the machine learning tasks for which these models are typically applied.

Table 2.3 Mapping of black-box model to machine learning tasks

Black-box model	Machine learning tasks
Tree ensembles (random forest, gradient-boosted trees)	Regression and classification
Deep neural networks (DNNs)	Regression and classification
Convolutional neural networks (CNNs)	Image classification, object detection
Recurrent neural networks (RNNs)	Sequence modeling, language understanding

I have now plotted in the black-box models in the same predictive power versus interpretability plane as introduced in section 2.1, shown in figure 2.23.

The black-box models are clustered in the top left of the plane because they have high predictive power but low interpretability. For mission-critical tasks, it is important not to trade off model performance (such as accuracy) for interpretability by applying white-box models. We will need to apply black-box models for such tasks and will need to find ways to interpret them. We can interpret black-box models in multiple ways, and doing so is the main focus of the remaining chapters in this book. In the next

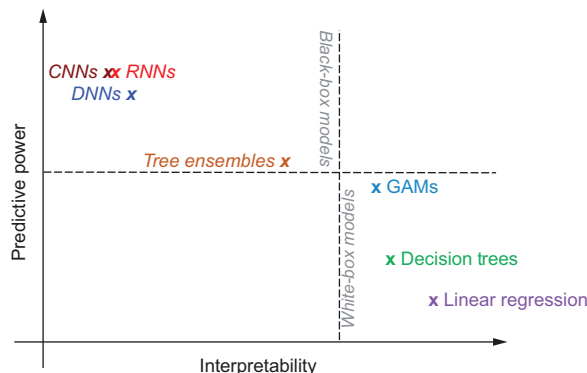


Figure 2.23 Black-box models on the interpretability versus predictive power plane

chapter, we will specifically focus on tree ensembles and how to interpret them using global, model-agnostic techniques.

Summary

- White-box models are inherently transparent. The machine learning process is straightforward to understand, and you can clearly interpret how the input features are transformed into the output. Using white-box models, you can identify the most important features, and those features are understandable.
- Linear regression is one of the simplest white-box models, where the target variable is modeled as a linear combination of the input features. You can determine the weights using the method of least squares and gradient descent.
- We can implement linear regression in Python using the `LinearRegression` class in the Scikit-Learn package. You can interpret the model by inspecting the coefficients or learned weights. The weights can also be used to determine the importance of each of the features. Linear regression, however, suffers from the problems of multicollinearity and underfitting.
- A decision tree is a slightly more advanced white-box model that can be used for both regression and classification tasks. You can predict the target variable by splitting the data across all features to minimize a cost function. You have learned the CART algorithm to learn the splits.
- A decision tree for regression tasks can be implemented in Python using the `DecisionTreeRegressor` class in Scikit-Learn. You can implement a decision tree for classification tasks using the `DecisionTreeClassifier` class in Scikit-Learn. You can interpret a decision tree learned using CART by visualizing it as a binary tree. The Scikit-Learn implementation also computes the feature importance for you. A decision tree can be used to model nonlinear relationships but tends to suffer from overfitting.
- GAMs are a powerful white-box model where the target variable is represented as a sum of smoothing functions representing the relationship of each of the features and the target. You know that regression splines and cubic splines are widely used to represent the smoothing function.
- Regression splines and GAMs can be implemented using the `pyGAM` package in Python. We can use the `LinearGAM` class for regression tasks and the `LogisticGAM` class for classification tasks. You can interpret a GAM by plotting the partial dependence of each of the features on the target. GAMs have a tendency to overfit, but this problem can be mitigated through regularization.
- Black-box models are models with really high predictive power and are typically applied to tasks for which model performance (such as accuracy) is extremely important. They are, however, inherently opaque. The machine learning process is complicated, and you can't easily understand how the input features are transformed into the output or target variable. As a result, you can't easily identify the most important features to predict the target variable.