data point belongs. Classes refer to categories and are also called targets or labels. For example, spam detection in email service providers involves binary classification (only 2 classes). The `MNIST` dataset contains a set of images, where each image is a single digit, which means there are 10 labels. Some applications in classification include: credit approval, medical diagnosis, and target marketing.

6.1.1 What Are Classifiers?

In the previous chapter, you learned that linear regression uses supervised learning in conjunction with numeric data: the goal is to train a model that can make numeric predictions (e.g., the price of stock tomorrow, the temperature of a system, its barometric pressure, and so forth). By contrast, classifiers use supervised learning in conjunction with nonnumeric classes of data: the goal is to train a model that can make categorical predictions.

For instance, suppose that each row in a dataset is a specific wine, and each column pertains to a specific wine feature (tannin, acidity, and so forth). Suppose further that there are five classes of wine in the dataset: for simplicity, let's label them A, B, C, D, and E. Given a new data point, which is to say a new row of data, a classifier for this dataset attempts to determine the label for this wine.

Some of the classifiers in this chapter can perform categorical classification and also make numeric predictions (i.e., they can be used for regression as well as classification).

6.1.2 Common Classifiers

Some of the most popular classifiers for machine learning are listed here (in no particular order):

- linear classifiers
- kNN
- logistic regression
- decision trees
- random forests
- SVMs
- Bayesian classifiers
- CNNs (deep learning)

Keep in mind that different classifiers have different advantages and disadvantages, which often involves a trade-off between complexity and accuracy, similar to algorithms in fields that are outside of AI.

In the case of deep learning, convolutional neural networks (CNNs) perform image classification, which makes them classifiers (they can also be used for audio and text processing).

The upcoming sections provide a brief description of the ML classifiers that are listed in the previous list.

6.1.3 Binary versus Multiclass Classification

Binary classifiers work with dataset that have two classes, whereas multiclass classifiers (sometimes called multinomial classifiers) distinguish more than two classes. Random forest classifiers and naïve Bayes classifiers support multiple classes, whereas SVMs and linear classifiers are binary classifiers (but multi-class extensions exist for SVM).

In addition, there are techniques for multiclass classification that are based on binary classifiers: one-versus-all (OvA) and one-versus-one (OvO).

The OvA technique (also called one-versus-the-rest) involves multiple binary classifiers that is equal to the number of classes. For example, if a dataset has five classes, then OvA uses five binary classifiers, each of which detects one of the five classes. In order to classify a data point in this particular dataset, select the binary classifier that has output the highest score.

The OvO technique also involves multiple binary classifiers, but in this case a binary classifier is used to train on a pair of classes. For instance, if the classes are A, B, C, D, and E, then 10 binary classifiers are required: one for A and B, one for A and C, one for A and D, and so forth, until the last binary classifier for D and E.

In general, if there are `n` classes, then `n*(n-1)/2` binary classifiers are required. Although the OvO technique requires considerably more binary classifiers (e.g., 190 are required for 20 classes) than the OvA technique (e.g., a mere 20 binary classifiers for 20 classes), the OvO technique has the advantage that each binary classifier is only trained on the portion of the dataset that pertains to its two chosen classes.

6.1.4 Multilabel Classification

Multilabel classification involves assigning multiple labels to an instance from a dataset. Hence, multilabel classification generalizes multiclass clas-

sification (discussed in the previous section), where the latter involves assigning a single label to an instance belonging to a dataset that has multiple classes. An article involving multilabel classification that contains `Keras`-based code is here:

https://medium.com/@vijayabhaskar96/multi-label-image-classification-tutorial-with-keras-imagedatagenerator-cd541f8eaf24

You can also perform an online search for articles that involve SKLearn or PyTorch for multilabel classification tasks.

6.2 What are Linear Classifiers?

A linear classifier separates a dataset into two classes. A linear classifier is a line for 2D points, a plane for 3D points, and a hyperplane (a generalization of a plane) for higher dimensional points.

Linear classifiers are often the fastest classifiers, so they are often used when the speed of classification is of high importance. Linear classifiers usually work well when the input vectors are sparse (i.e., mostly zero values) or when the number of dimensions is large.

6.3 What is kNN?

The k nearest neighbor (kNN) algorithm is a classification algorithm. In brief, data points that are "near" each other are classified as belonging to the same class. When a new point is introduced, it's added to the class of the majority of its nearest neighbor. For example, suppose that k equals 3, and a new data point is introduced. Look at the class of its 3 nearest neighbors: let's say they are A, A, and B. Then by majority vote, the new data point is labeled as a data point of class A.

The kNN algorithm is essentially a heuristic and not a technique with complex mathematical underpinnings, and yet it's still an effective and useful algorithm.

Try the kNN algorithm if you want to use a simple algorithm, or when you believe that the nature of your dataset is highly unstructured. The kNN algorithm can produce highly nonlinear decisions despite being very simple. You can use kNN in search applications where you are searching for "similar" items.

Measure similarity by creating a vector representation of the items, and then compare the vectors using an appropriate distance metric (such as Euclidean distance).

Some concrete examples of kNN search include searching for semantically similar documents.

6.3.1 How to Handle a Tie in kNN

An odd value for k is less likely to result in a tie vote, but it's not impossible. For example, suppose that k equals 7, and when a new data point is introduced, its 7 nearest neighbors belong to the set {A,B,A,B,A,B,C}. As you can see, there is no majority vote, because there are 3 points in class A, 3 points in class B, and 1 point in class C.

There are several techniques for handling a tie in kNN, as listed here:

- Assign higher weights to closer points
- Increase the value of k until a winner is determined
- Decrease the value of k until a winner is determined
- Randomly select one class

If you reduce k until it equals 1, it's still possible to have a tie vote: there might be two points that are equally distant from the new point, so you need a mechanism for deciding which of those two points to select as the 1-neighbor.

If there is a tie between classes A and B, then randomly select either class A or class B. Another variant is to keep track of the "tie" votes, and alternate round-robin style to make ensure a more even distribution.

6.4 What are Decision Trees?

Decision trees are another type of classification algorithm that involves a tree-like structure. In a "generic" tree, the placement of a data point is determined by simple conditional logic. As a simple illustration, suppose that a dataset contains a set of numbers that represent ages of people, and let's also suppose that the first number is 50. This number is chosen as the root of the tree, and all numbers that are smaller than 50 are added on the left branch of the tree, whereas all numbers that are greater than 50 are added on the right branch of the tree.

For example, suppose we have the sequence of numbers is {50, 25, 70, 40}. Then we can construct a tree as follows: 50 is the root node; 25 is the left child of 50; 70 is the right child of 50; and 40 is the right child of 20. Each additional numeric value that we add to this dataset is processed to determine which direction to proceed ("left or right") at each node in the tree.

Listing 6.1 displays the contents of `sklearn_tree2.py` that defines a set of 2D points in the Euclidean plane, along with their labels, and then predicts the label (i.e., the class) of several other 2D points in the Euclidean plane.

Listing 6.1: sklearn_tree2.py

```
from sklearn import tree

# X = pairs of 2D points and Y = the class of each point
X = [[0, 0], [1, 1], [2,2]]
Y = [0, 1, 1]

tree_clf = tree.DecisionTreeClassifier()
tree_clf = tree_clf.fit(X, Y)

#predict the class of samples:
print("predict class of [-1., -1.]:")
print(tree_clf.predict([[-1., -1.]]))

print("predict class of [2., 2.]:")
print(tree_clf.predict([[2., 2.]]))

# the percentage of training samples of the same class
# in a leaf note equals the probability of each class
print("probability of each class in [2.,2.]:")
print(tree_clf.predict_proba([[2., 2.]]))
```

Listing 6.1 imports the tree class from `sklearn` and then initializes the arrays `X` and `y` with data values. Next, the variable `tree_clf` is initialized as an instance of the `DecisionTreeClassifier` class, after which it is trained by invoking the `fit()` method with the values of `X` and `y`.

Now launch the code in Listing 6.3 and you will see the following output:

```
predict class of [-1., -1.]:
[0]
predict class of [2., 2.]:
[1]
probability of each class in [2.,2.]:
[[0. 1.]]
```

As you can see, the points [-1,-1] and [2,2] are correctly labeled with the values 0 and 1, respectively, which is probably what you expected.

Listing 6.2 displays the contents of `sklearn_tree3.py` that extends the code in Listing 6.1 by adding a third label, and also by predicting the label of three points instead of two points in the Euclidean plane (the modifications are shown in bold).

Listing 6.2: sklearn_tree3.py

```
from sklearn import tree

# X = pairs of 2D points and Y = the class of each point
X = [[0, 0], [1, 1], [2,2]]
Y = [0, 1, 2]

tree_clf = tree.DecisionTreeClassifier()
tree_clf = tree_clf.fit(X, Y)

#predict the class of samples:
print("predict class of [-1., -1.]:")
print(tree_clf.predict([[-1., -1.]]))

print("predict class of [0.8, 0.8]:")
print(tree_clf.predict([[0.8, 0.8]]))

print("predict class of [2., 2.]:")
print(tree_clf.predict([[2., 2.]]))

# the percentage of training samples of the same class
# in a leaf note equals the probability of each class
print("probability of each class in [2.,2.]:")
print(tree_clf.predict_proba([[2., 2.]]))
```

Now launch the code in Listing 6.2 and you will see the following output:

```
predict class of [-1., -1.]:
[0]
predict class of [0.8, 0.8]:
[1]
predict class of [2., 2.]:
[2]
probability of each class in [2.,2.]:
[[0. 0. 1.]]
```

As you can see, the points [-1,-1], [0.8, 0.8], and [2,2] are correctly labeled with the values 0, 1, and 2, respectively, which again is probably what you expected.

Listing 6.3 displays a portion of the dataset `partial_wine.csv`, which contains two features and a label column (there are three classes). The total row count for this dataset is 178.

Listing 6.3: partial_wine.csv

```
Alcohol, Malic acid, class
14.23,1.71,1
13.2,1.78,1
13.16,2.36,1
14.37,1.95,1
13.24,2.59,1
14.2,1.76,1
```

Listing 6.4 displays contents of `tree_classifier.py` that uses a decision tree in order to train a model on the dataset `partial_wine.csv`.

Listing 6.4: tree_classifier.py

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

# Importing the dataset
dataset = pd.read_csv('partial_wine.csv')
X = dataset.iloc[:, [0, 1]].values
y = dataset.iloc[:, 2].values

# split the dataset into a training set and a test set
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X,
y, test_size = 0.25, random_state = 0)

# Feature Scaling
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)

# ====> INSERT YOUR CLASSIFIER CODE HERE <====
from sklearn.tree import DecisionTreeClassifier
classifier = DecisionTreeClassifier(criterion='entropy',
random_state=0)
classifier.fit(X_train, y_train)
# ====> INSERT YOUR CLASSIFIER CODE HERE <====

# predict the test set results
```

```
y_pred = classifier.predict(X_test)

# generate the confusion matrix
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_test, y_pred)
print("confusion matrix:")
print(cm)
```

Listing 6.4 contains some `import` statements and then populates the `Pandas` DataFrame `dataset` with the contents of the CSV file `partial_wine.csv`. Next, the variable `X` is initialized with the first two columns (and all the rows) of dataset, and the variable `y` is initialized with the third column (and all the rows) of dataset.

Next, the variables `X_train, X_test, y_train, y_test` are populated with data from `X` and `y` using a 75/25 split proportion. Notice that the variable `sc` (which is an instance of the `StandardScalar` class) performs a scaling operation on the variables `X_train` and `X_test`.

The code block shown in bold in Listing 6.4 is where we create an instance of the `DecisionTreeClassifier` class and then train the instance with the data in the variables `X_train` and `X_test`.

The next portion of Listing 6.4 populates the variable `y_pred` with a set of predictions that are generated from the data in the `X_test` variable. The last portion of Listing 6.4 creates a confusion matrix based on the data in `y_test` and the predicted data in `y_pred`.

Remember that all the diagonal elements of a confusion matrix are correct predictions (such as true positive and true negative); all the other cells contain a numeric value that specifies the number of predictions that are incorrect (such as false positive and false negative).

Now launch the code in Listing 6.4 and you will see the following output for the confusion matrix in which there are 36 correct predictions and 9 incorrect predictions (with an accuracy of 80%):

```
confusion matrix:
[[13  1  2]
 [ 0 17  4]
 [ 1  1  6]]
from sklearn.metrics import confusion_matrix
```

There is a total of 45 entries in the preceding 3x3 matrix, and the diagonal entries are correctly identified labels. Hence the accuracy is 36/45 = 0.80.

6.5 What are Random Forests?

Random forests are a generalization of decision trees: this classification algorithm involves multiple trees (the number is specified by you). If the data involves making a numeric prediction, the average of the predictions of the trees is computed. If the data involves a categorical prediction, the mode of the predictions of the trees is determined.

By way of analogy, random forests operate in a manner similar to financial portfolio diversification: the goal is to balance the losses with higher gains. Random forests use a "majority vote" to make predictions, which operates under the assumption that selecting the majority vote is more likely to be correct (and more often) than any individual prediction from a single tree.

You can easily modify the code in Listing 6.4 to use a random forest by replacing the two lines shown in bold with the following code:

```
from sklearn.ensemble import RandomForestClassifier
classifier = RandomForestClassifier(n_estimators = 10,
criterion='entropy', random_state = 0)
```

Make this code change, launch the code, and examine the confusion matrix to compare its accuracy with the accuracy of the decision tree in Listing 6.4.

6.6 What are SVMs?

Support vector machines (SVM) involve a supervised ML algorithm and can be used for classification or regression problems. SVM can work with nonlinearly separable data as well as linearly separable data. SVM uses a technique called the "kernel trick" to transform data and then finds an optimal boundary the transform involves higher dimensionality. This technique results in a separation of the transformed data, after which it's possible to find a hyperplane that separates the data into two classes.

SVMs are more common in classification tasks than regression tasks. Some use cases for SVMs include:

- text classification tasks: category assignment
- detecting spam / sentiment analysis
- used for image recognition: aspect-based recognition color-based classification

- handwritten digit recognition (postal automation)

6.6.1 Tradeoffs of SVMs

Although SVMs are extremely powerful, there are tradeoffs involved. Some of the advantages of SVMs are listed here:

- high accuracy
- works well on smaller cleaner datasets
- can be more efficient because it uses a subset of training points
- an alternative to CNNs in cases of limited datasets
- captures more complex relationships between data points

Despite the power of SVMS, there are some disadvantages of SVMs, which are listed here:

- not suited to larger datasets: training time can be lengthy
- less effective on noisier datasets with overlapping classes

SVMs involve more parameters than decision trees and random forests

Suggestion: modify Listing 6.4 to use an SVM by replacing the two lines shown in bold with the following two lines shown in bold:

```
from sklearn.svm import SVC
classifier = SVC(kernel = 'linear', random_state = 0)
```

You now have an SVM-based model, simply by making the previous code update. Make the code change, then launch the code and examine the confusion matrix in order to compare its accuracy with the accuracy of the decision tree model and the random forest model earlier in this chapter.

6.7 What is Bayesian Inference?

Bayesian inference is an important technique in statistics that involves statistical inference and Bayes' theorem to update the probability for a hypothesis as more information becomes available. Bayesian inference is often called *Bayesian probability*, and it's important in dynamic analysis of sequential data.

6.7.1 Bayes Theorem

Given two sets A and B, let's define the following numeric values (all of them are between 0 and 1):

```
P(A) = probability of being in set A
P(B) = probability of being in set B
P(Both) = probability of being in A intersect B
P(A|B) = probability of being in A (given you're in B)
P(B|A) = probability of being in B (given you're in A)
```

Then the following formulas are also true:

```
P(A|B) = P(Both)/P(B) (#1)
P(B|A) = P(Both)/P(A) (#2)
```

Multiply the preceding pair of equations by the term that appears in the denominator and we get these equations:

```
P(B)*P(A|B) = P(Both) (#3)
P(A)*P(B|A) = P(Both) (#4)
```

Now set the left-side of equations #3 and #4 equal to each another and that gives us this equation:

```
P(B)*P(A|B) = P(A)*P(B|A) (#5)
```

Divide both sides of #5 by P(B) and we get this well-known equation:

```
P(A|B) = P(A)*P(A|B)/P(B) (#6)
```

6.7.2 Some Bayesian Terminology

In the previous section, we derived the following relationship:

```
P(h|d) = (P(d|h) * P(h)) / P(d)
```

There is a name for each of the four terms in the preceding equation, and they are:

First, the *posterior probability* is `P(h|d)`, which is the probability of hypothesis `h` given the data `d`.

Second, `P(d|h)` is the probability of data `d` given that the hypothesis `h` was true.

Third, the *prior probability* of `h` is `P(h)`, which is the probability of hypothesis `h` being true (regardless of the data).

Finally, `P(d)` is the probability of the data (regardless of the hypothesis).

We are interested in calculating the posterior probability of P(h|d) from the prior probability p(h) with P(d) and P(d|h).

6.7.3 What Is MAP?

The maximum a posteriori (MAP) hypothesis is the hypothesis with the highest probability, which is the maximum probable hypothesis. This can be written as follows:

```
MAP(h) = max(P(h|d))
or:
MAP(h) = max((P(d|h) * P(h)) / P(d))
or:
MAP(h) = max(P(d|h) * P(h))
```

6.7.4 Why Use Bayes Theorem?

Bayes' theorem describes the probability of an event based on the prior knowledge of the conditions that might be related to the event. If we know the conditional probability, we can use Bayes rule to find out the reverse probabilities. The previous statement is the general representation of the Bayes rule.

6.8 What is a Bayesian Classifier?

A naïve Bayes (NB) classifier is a probabilistic classifier inspired by the Bayes theorem. An NB classifier assumes the attributes are conditionally independent and it works well even when assumption is not true. This assumption greatly reduces computational cost, and it's a simple algorithm to implement that only requires linear time. Moreover, an NB classifier is easily scalable to larger datasets and good results are obtained in most cases. Other advantages of an NB classifier include:

- can be used for binary and multiclass classification
- provides different types of NB algorithms
- good choice for text classification problems
- a popular choice for spam email classification
- can be easily trained on small datasets

As you can probably surmise, NB classifiers do have some disadvantages, as listed here:

- all features are assumed unrelated
- it cannot learn relationships between features
- it can suffer from "the zero probability problem"

The *zero probability problem* refers to the case when the conditional probability is zero for an attribute, it fails to give a valid prediction. However, can be fixed explicitly using a Laplacian estimator.

6.8.1 Types of Naïve Bayes Classifiers

There are three major types of NB classifiers:

- Gaussian Naïve Bayes
- multinomialNB Naïve Bayes
- Bernoulli Naïve Bayes

Details of these classifiers are beyond the scope of this chapter, but you can perform an online search for more information.

6.9 Training Classifiers

Some common techniques for training classifiers are here:

- holdout method
- k-fold cross-validation

The *holdout method* is the most common method, which starts by dividing the dataset into two partitions called train and test (80% and 20%, respectively). The train set is used for training the model, and the test data tests its predictive power.

The *k-fold cross-validation* technique is used to verify that the model is not over-fitted. The dataset is randomly partitioned into k mutually exclusive subsets, where each partition has equal size. One partition is for testing and the other partitions are for training. Iterate throughout the whole of the k-folds.

6.10 Evaluating Classifiers

Whenever you select a classifier for a dataset, it's obviously important to evaluate the accuracy of that classifier. Some common techniques for evaluating classifiers are listed here:

- precision and recall
- receiver operating characteristics (ROC) curve

Precision and recall are discussed in Chapter 2 and reproduced here for your convenience. Let's define the following variables:

```
TP = the number of true positive results
FP = the number of false positive results
TN = the number of true negative results
FN = the number of false negative results
```

Then the definitions of precision, accuracy, and recall are given by the following formulas:

```
precision = TP/(TN + FP)
accuracy  = (TP + TN)/[P + N]
recall    = TP/[TP + FN]
```

The *receiver operating characteristics* (ROC) curve is used for visual comparison of classification models that shows the trade-off between the true positive rate and the false positive rate. The area under the ROC curve is a measure of the accuracy of the model. When a model is closer to the diagonal, it is less accurate and the model with perfect accuracy will have an area of 1.0.

The ROC curve plots true positive rate versus false positive rate. Another type of curve is the precision-recall (PR) curve that plots rrecision versus recall. When dealing with highly skewed datasets (strong class imbalance), PR curves give better results.

Later in this chapter you will see many of the `Keras`-based classes (located in the `tf.keras.metrics` namespace) that correspond to common statistical terms, which includes some of the terms in this section.

This concludes the portion of the chapter pertaining to statistical terms and techniques for measuring the validity of a dataset. Now let's look at activation functions in machine learning, which is the topic of the next section.

6.11 What are Activation Functions?

A one-sentence description: an activation function is (usually) a nonlinear function that introduces nonlinearity into a neural network, thereby preventing a *consolidation* of the hidden layers in neural network. Specifically, suppose that every pair of adjacent layers in a neural network involves just a matrix transformation and no activation function. *Such a network is a linear system, which means that its layers can be consolidated into a much smaller system.*

First, the weights of the edges that connect the input layer with the first hidden layer can be represented by a matrix: let's call it `W1`. Next, the weights of the edges that connect the first hidden layer with the second hidden layer can also be represented by a matrix: let's call it `W2`. Repeat this process until we reach the edges that connect the final hidden layer with the output layer: let's call this matrix `Wk`. Since we do not have an activation function, we can simply multiply the matrices `W1`, `W2`, …, `Wk` together and produce one matrix: let's call it `W`. We have now replaced the original neural network with an equivalent neural network that contains one input layer, a single matrix of weights `W`, and an output layer. In other words, we no longer have our original multilayered neural network.

Fortunately, we can prevent the previous scenario from happening when we specify an activation function between every pair of adjacent layers. In other words, an activation function at each layer prevents this *matrix consolidation*. Hence, we can maintain all the intermediate hidden layers during the process of training the neural network.

For simplicity, let's assume that we have the same activation function between every pair of adjacent layers (we'll remove this assumption shortly). The process for using an activation function in a neural network is initially a "three step," after which it's a "two-step," as described here:

1. Start with an input vector `x1` of numbers.
2. Multiply `x1` by the matrix of weights W1 that represents the edges that connect the input layer with the first hidden layer: the result is a new vector `x2`.
3. "Apply" the activation function to each element of `x2` to create another vector `x3`.

Now repeat steps 2 and 3, except that we use the "starting" vector x3 and the weights matrix `W2` for the edges that connect the first hidden layer with the second hidden layer (or just the output layer if there is only one hidden layer).

After completing the preceding process, we have *preserved* the neural network, which means that it can be trained on a dataset. One other thing: instead of using the same activation function at each step, you can replace each activation function by a different activation function (the choice is yours).

6.11.1 Why Do We Need Activation Functions?

The previous section outlines the process for transforming an input vector from the input layer and then through the hidden layers until it reaches the output layer. The purpose of activation functions in neural networks is vitally important, so it's worth repeating here: activation functions "maintain" the structure of neural networks and prevent them from being reduced to an input layer and an output layer. In other words, if we specify a nonlinear activation function between every pair of consecutive layers, then the neural network cannot be replaced with a neural network that contains fewer layers.

Without a nonlinear activation function, we simply multiply a weight matrix for a given pair of consecutive layers with the output vector that is produced from the previous pair of consecutive layers. We repeat this simple multiplication until we reach the output layer of the neural network. After reaching the output layer, we have effectively replaced multiple matrices with a single matrix that connects the numbers in the input layer with the numbers in the output layer.

6.11.2 How Do Activation Functions Work?

If this is the first time you have encountered the concept of an activation function, it's probably confusing, so here's an analogy that might be helpful. Suppose you're driving your car late at night and there's nobody else on the highway. You can drive at a constant speed for as long as there are no obstacles (stop signs, traffic lights, and so forth). However, suppose you drive into the parking lot of a large grocery store. When you approach a speed bump you must slow down, cross the speed bump, and increase speed again, and repeat this process for every speed bump.

Think of the nonlinear activation functions in a neural network as the counterpart to the speed bumps: you simply cannot maintain a constant speed, which (by analogy) means that you cannot first multiply all the weight

matrices together and "collapse" them into a single weight matrix. Another analogy involves a road with multiple toll booths: you must slow down, pay the toll, and then resume driving until you reach the next toll booth. These are only analogies (and hence imperfect) to help you understand the need for nonlinear activation functions.

6.12 Common Activation Functions

Although there are many activation functions (and you can define your own if you know how to do so), here is a list of common activation functions, followed by brief descriptions:

- Sigmoid
- Tanh
- ReLU
- ReLU6
- ELU
- SELU

The `sigmoid` activation function is based on Euler's constant e, with a range of values between 0 and 1, and its formula is shown here:

```
1/[1+e^(-x)]
```

The `tanh` activation function is also based on Euler's constant e, and its formula is shown here:

```
[e^x - e^(-x)]/[e^x+e^(-x)]
```

One way to remember the preceding formula is to note that the numerator and denominator have the same pair of terms: they are separated by a "-" sign in the numerator and a "+" sign in the denominator. The `tanh` function has a range of values between -1 and 1.

The rectified linear unit (ReLU) activation function is straightforward: if x is negative then ReLU(x) is 0; for all other values of x, ReLU(x) equals x. ReLU6 is specific to TensorFlow, and it's a variation of ReLU(x): the additional constraint is that ReLU(x) equals 6 when x >= 6 (hence its name).

exponential linear unit (ELU) is the exponential "envelope" of ReLU, which replaces the two linear segments of ReLU with an Exponential activation function that is differentiable for all values of x (including x = 0).

Scaled exponential linear unit (SELU)is slightly more complicated than the other activation functions (and used less frequently). For a thorough explanation of these and other activation functions (along with graphs that depict their shape), navigate to the following Wikipedia link:

https://en.wikipedia.org/wiki/Activation_function

The preceding link provides a long list of activation functions as well as their derivatives.

6.12.1 Activation Functions in Python

Listing 6.5 displays contents of the file `activations.py` that contains the formulas for various activation functions.

Listing 6.5: activations.py

```
import numpy as np

# Python sigmoid example:
z = 1/(1 + np.exp(-np.dot(W, x)))

# Python tanh example:
z = np.tanh(np.dot(W,x))

# Python ReLU example:
z = np.maximum(0, np.dot(W, x))
```

Listing 6.5 contains Python code that use `NumPy` methods in order to define a sigmoid function, a `tanh` function, and a ReLU function. Note that you need to specify values for `x` and `W` in order to launch the code in Listing 6.5.

6.12.2 `Keras` Activation Functions

TensorFlow (and many other frameworks) provide implementations for many activation functions, which saves you the time and effort from writing your own implementation of activation functions.

Here is a list of TF 2/Keras APIs for activation functions that are located in the tf.keras.layers namespace:

- tf.keras.layers.leaky_relu
- tf.keras.layers.relu
- tf.keras.layers.relu6
- tf.keras.layers.selu
- tf.keras.layers.sigmoid

- tf.keras.layers.sigmoid_cross_entropy_with_logits
- tf.keras.layers.softmax
- tf.keras.layers.softmax_cross_entropy_with_logits_v2
- tf.keras.layers.softplus
- tf.keras.layers.softsign
- tf.keras.layers.softmax_cross_entropy_with_logits
- tf.keras.layers.tanh
- tf.keras.layers.weighted_cross_entropy_with_logits

The following subsections provide additional information regarding some of the activation functions in the preceding list. Keep the following point in mind: for simple neural networks, use ReLU as your first preference.

6.13 The ReLU and ELU Activation Functions

Currently ReLU is often the recommended activation function: previously the preferred activation function was `tanh` (and before `tanh` it was `sigmoid`). ReLU behaves close to a linear unit and provides the best training accuracy and validation accuracy.

ReLU is like a switch for linearity: it's "off" if you don't need it, and its derivative is 1 when it's active, which makes ReLU the simplest of all the current activation functions. Note that the second derivative of the function is 0 everywhere: it's a very simple function that simplifies optimization. In addition, the gradient is large whenever you need large values, and it never saturates (i.e., it does not shrink to zero on the positive horizontal axis).

Rectified linear units and generalized versions are based on the principle that linear models are easier to optimize. Use the ReLU activation function or one of its related alternatives (discussed later).

6.13.1 The Advantages and Disadvantages of ReLU

The following list contains the advantages of the ReLU activation function:

- It does not saturate in the positive region.
- It's very efficient in terms of computation.
- Models with ReLU typically converge faster those with other activation functions.

However, ReLU does have a disadvantage when the activation value of a ReLU neuron becomes 0: then the gradients of the neuron will also be 0 during back-propagation. You can mitigate this scenario by judiciously assigning the values for the initial weights as well as the learning rate.

6.13.2 ELU

Exponential linear unit (ELU) is based on ReLU: the key difference is that ELU is differentiable at the origin (ReLU is a continuous function but *not* differentiable at the origin). However, keep in mind several points. First, ELU's trade computational efficiency for "immortality" (immunity to dying): read the following paper for more details: arxiv.org/abs/1511.07289. Secondly, RELUs are still popular and preferred over ELU because the use of ELU introduces an additional new hyper-parameter.

6.14 Sigmoid, Softmax, and Hardmax Similarities

The `sigmoid` activation function has a range in (0,1), and it saturates and "kills" gradients. Unlike the `tanh` activation function, `sigmoid` outputs are not zero-centered. In addition, both `sigmoid` and `softmax` (discussed later) are discouraged for vanilla feed forward implementation. (See Chapter 6 of the online book *Deep Learning* by Ian Goodfellow et al. 2015). However, the `sigmoid` activation function is still used in LSTMs (specifically for the forget gate, input gate, and the output gate), gated recurrent units (GRUs), and probabilistic models. Moreover, some autoencoders have additional requirements that preclude the use of piecewise linear activation functions.

6.14.1 Softmax

The `softmax` activation function maps the values in a dataset to another set of values that are between 0 and 1, and whose sum equals 1. Thus, `softmax` creates a probability distribution. In the case of image classification with convolutional neural networks (CNNs), the `softmax` activation function maps the values in the final hidden layer to the 10 neurons in the output layer. The index of the position that contains the largest probability is matched with the index of the number 1 in the one-hot encoding of the input image. If the index values are equal, then the image has been classified, otherwise it's considered a mismatch.

6.14.2 Softplus

The `softplus` activation function is a smooth (i.e., differentiable) approximation to the ReLU activation function. Recall that the origin is the only nondifferentiable point of the ReLU function, which is "smoothed" by the `softmax` activation whose equation is here:

```
f(x) = ln(1 + e^x)
```

6.14.3 Tanh

The `tanh` activation function has a range in (-1,1), whereas the `sigmoid` function has a range in (0,1). Both of these two activations saturate, but unlike the `sigmoid` neuron the `tanh` output is zero-centered. Therefore, in practice the `tanh` nonlinearity is always preferred to the `sigmoid` nonlinearity.

The `sigmoid` and `tanh` activation functions appear in LSTMs (sigmoid for the three gates and tanh for the internal cell state) as well as GRUs during the calculations pertaining to input gates, forget gates, and output gates (discussed in more detail in the next chapter).

6.15 Sigmoid, Softmax, and HardMax Differences

This section briefly discusses some of the differences among these three functions. First, the `sigmoid` function is used for binary classification in logistic regression model, as well as the gates in LSTMs and GRUs. The `sigmoid` function is used as activation function while building neural networks, but keep in mind that the sum of the probabilities is *not* necessarily equal to 1.

Second, the `softmax` function generalizes the `sigmoid` function: it's used for multiclassification in logistic regression model. The `softmax` function is the activation function for the *fully connected layer* in CNNs, which is the right-most hidden layer and the output layer. Unlike the sigmoid function, the sum of the probabilities *must* equal 1. You can use either the sigmoid function or `softmax` for binary (n=2) classification.

Third, the so-called "`hardmax`" function assigns 0 or 1 to output values (similar to a step function). For example, suppose that we have three classes `{c1, c2, c3}` whose scores are `[1, 7, 2]`, respectively. The `hardmax` probabilities are `[0, 1, 0]`, whereas the `softmax` probabilities are `[0.1,`

`0.7, 0.2]`. Notice that the sum of the `hardmax` probabilities is 1, which is also true of the sum of the `softmax` probabilities. However, the `hardmax` probabilities are all-or-nothing, whereas the `softmax` probabilities are analogous to receiving "partial credit."

6.16 What is Logistic Regression?

Despite its name, logistic regression is a classifier and a linear model with a binary output. Logistic regression works with multiple independent variables and involves a sigmoid function for calculating probabilities. Logistic regression is essentially the result of applying the `sigmoid` activation function to linear regression in order to perform binary classification.

Logistic regression is useful in a variety of unrelated fields. Such fields include machine learning, various medical fields, and social sciences. Logistic regression can be used to predict the risk of developing a given disease, based on various observed characteristics of the patient. Other fields that use logistic regression include engineering, marketing, and economics.

Logistic regression can be binomial (only two outcomes for a dependent variable), multinomial (three or more), or ordinal (ordered dependent variables), but mainly used for binomial cases. For instance, suppose that a dataset consists of data that belong either to class A or to class B. If you are given a new data point, logistic regression predicts whether that new data point belongs to class A or to class B. By contrast, linear regression predicts a numeric value, such as the next-day value of a stock.

6.16.1 Setting a Threshold Value

The *threshold value* is a numeric value that determines which data points belong to class A and which points belong to class B. For instance, a pass/fail threshold might be 0.70. A pass/fail threshold for passing a writing driver's test in California is 0.85.

As another example, suppose that p = 0.5 is the "cutoff" probability. Then we can assign class A to the data points that occur with probability > 0.5 and assign class B to data points that occur with probability <= 0.5. Since there are only two classes, we do have a classifier.

A similar (yet slightly different) scenario involves tossing a well-balanced coin. We know that there is a 50% chance of throwing heads (let's label this outcome as class A) and a 50% chance of throwing tails (let's label this outcome as class B). If we have a dataset that consists of labeled out-

comes, then we have the expectation that approximately 50% of them are class A and class B.

However, we have no way to determine (in advance) what percentage of people will pass their written driver's test, or the percentage of people who will pass their course. Datasets containing outcomes for these types of scenarios need to be trained, and logistic regression can be a suitable technique for doing so.

6.16.2 Logistic Regression: Important Assumptions

Logistic regression requires the observations to be independent of each other. In addition, logistic regression requires little or no multicollinearity among the independent variables. Logistic regression handles numeric, categorical, and continuous variables, and also assumes linearity of independent variables and log odds, which is defined here:

```
odds = p/(1-p) and logit = log(odds)
```

This analysis does not require the dependent and independent variables to be related linearly; however, another requirement is that independent variables are linearly related to the log odds.

Logistic regression is used to obtain odds ratio in the presence of more than one explanatory variable. The procedure is quite similar to multiple linear regression, with the exception that the response variable is binomial. The result is the impact of each variable on the odds ratio of the observed event of interest.

6.16.3 Linearly Separable Data

Linearly separable data is data that can be separated by a line (in 2D), a plane (in 3D), or a hyperplane (in higher dimensions). Linearly nonseparable data is data (clusters) that cannot be separated by a line or a hyperplane. For example, the XOR function involves data points that cannot be separated by a line. If you create a truth table for an `XOR` function with two inputs, the points (0,0) and (1,1) belong to class 0, whereas the points (0,1) and (1,0) belong to class 1 (draw these points in a 2D plane to convince yourself). The solution involves transforming the data in a higher dimension so that it becomes linearly separable, which is the technique used in SVMS (discussed earlier in this chapter).

6.17 `Keras`, Logistic Regression, and Iris Dataset

Listing 6.6 displays the contents of `tf2_keras_iris.py` that defines a `Keras`-based model to perform logistic regression.

Listing 6.6: tf2_keras_iris.py

```
import tensorflow as tf
import matplotlib.pyplot as plt

from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import OneHotEncoder,
StandardScaler

iris = load_iris()
X = iris['data']
y = iris['target']

#you can view the data and the labels:
#print("iris data:",X)
#print("iris target:",y)

# scale the X values so they are between 0 and 1
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

X_train, X_test, y_train, y_test = train_test_split(X_
scaled, y, test_size = 0.2)

model = tf.keras.models.Sequential()
model.add(tf.keras.layers.Dense(activation='relu',
input_dim=4,
          units=4, kernel_initializer='uniform'))

model.add(tf.keras.layers.Dense(activation='relu',
units=4,
                    kernel_initializer='uniform'))
model.add(tf.keras.layers.Dense(activation='sigmoid',
units=1,
                    kernel_initializer='uniform'))
#model.add(tf.keras.layers.Dense(1,
activation='softmax'))

model.compile(optimizer='adam', loss='mean_squared_
error', metrics=['accuracy'])
model.fit(X_train, y_train, batch_size=10, epochs=100)

# Predicting values from the test set
y_pred = model.predict(X_test)

# scatter plot of test values-vs-predictions
fig, ax = plt.subplots()
```

```
ax.scatter(y_test, y_pred)
ax.plot([y_test.min(), y_test.max()], [y_test.min(), y_
test.max()], 'r*--')

ax.set_xlabel('Calculated')
ax.set_ylabel('Predictions')
plt.show()
```

Listing 6.6 starts with an assortment of `import` statements, and then initializes the variable `iris` with the `Iris` dataset. The variable X contains the first three columns (and all the rows) of the `Iris` dataset, and the variable `y` contains the fourth column (and all the rows) of the `Iris` dataset.

The next portion of Listing 6.6 initializes the training set and the test set using an 80/20 data split. Next, the `Keras`-based model contains three `Dense` layers, where the first two specify the relu activation function and the third layer specifies the sigmoid activation function.

The next portion of Listing 6.6 compiles the model, trains the model, and then calculates the accuracy of the model via the test data. Launch the code in Listing 6.6 and you will see the following output:

```
Train on 120 samples
Epoch 1/100120/120 [==============================] - 0s
980us/sample - loss: 0.9819 - accuracy: 0.3167
Epoch 2/100
120/120 [==============================] - 0s 162us/
sample - loss: 0.9789 - accuracy: 0.3083
Epoch 3/100
120/120 [==============================] - 0s 204us/
sample - loss: 0.9758 - accuracy: 0.3083
Epoch 4/100
120/120 [==============================] - 0s 166us/
sample - loss: 0.9728 - accuracy: 0.3083
Epoch 5/100
120/120 [==============================] - 0s 160us/
sample - loss: 0.9700 - accuracy: 0.3083
// details omitted for brevity
Epoch 96/100
120/120 [==============================] - 0s 128us/
sample - loss: 0.3524 - accuracy: 0.6500
Epoch 97/100
120/120 [==============================] - 0s 184us/
sample - loss: 0.3523 - accuracy: 0.6500
```