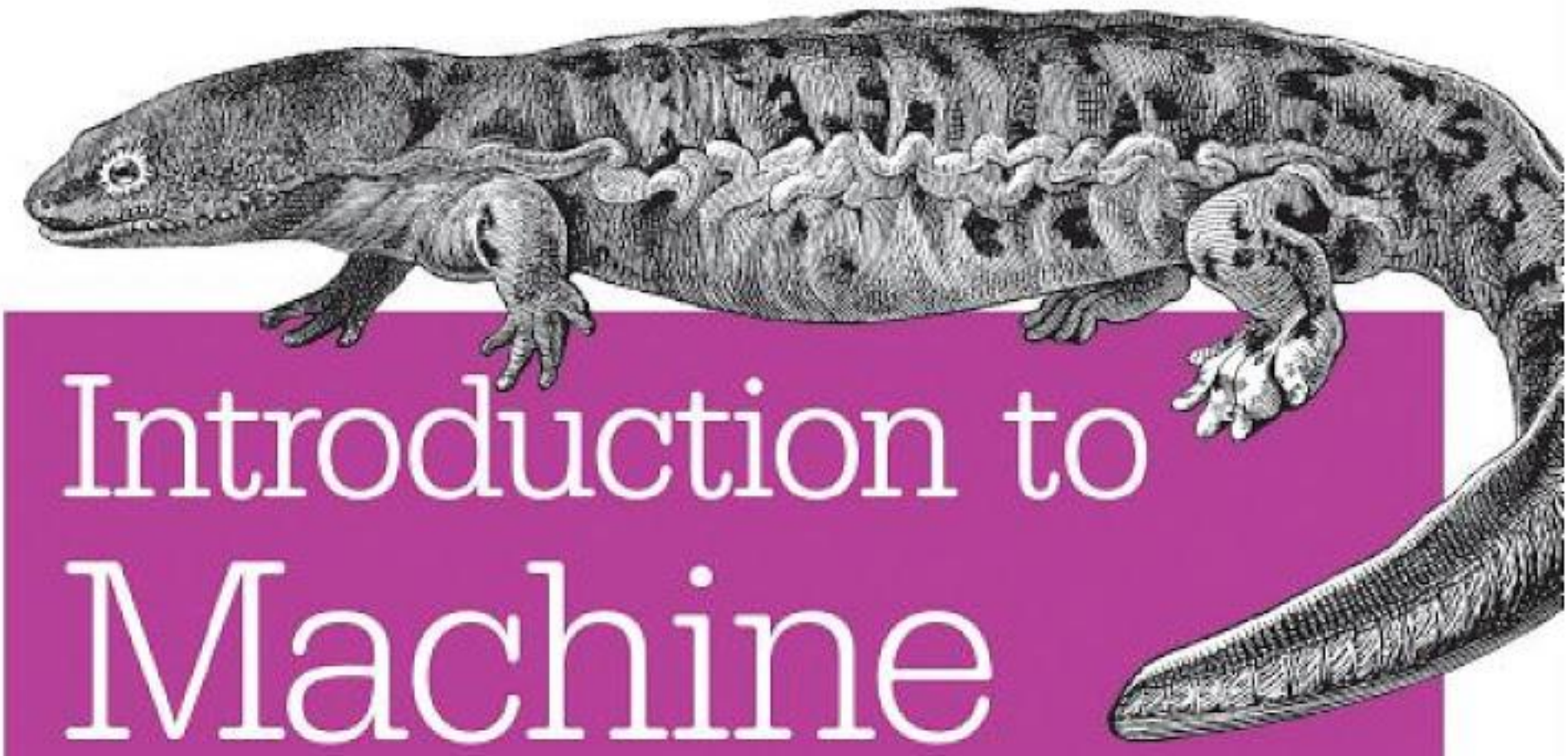
O'REILLY®



Introduction to Machine Learning with Python

A GUIDE FOR DATA SCIENTISTS

powered by

jupyter

Andreas C. Müller & Sarah Guido

Introduction to Machine Learning with Python

by Andreas C. Müller and Sarah Guido

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://oreilly.com/safari). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editor: Dawn Schanafelt

Production Editor: Kristen Brown

Copyeditor: Rachel Head

Proofreader: Jasmine Kwityn

Indexer: Judy McConville

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Rebecca Demarest

```
columns=iris_dataset.feature_names)

pd.plotting.scatter_matrix(iris_dataframe, c=y_train, figsize=(15,
15),
                           marker='o', hist_kwds={'bins': 20},
s=60,
                           alpha=.8, cmap=mglearn.cm3)
```

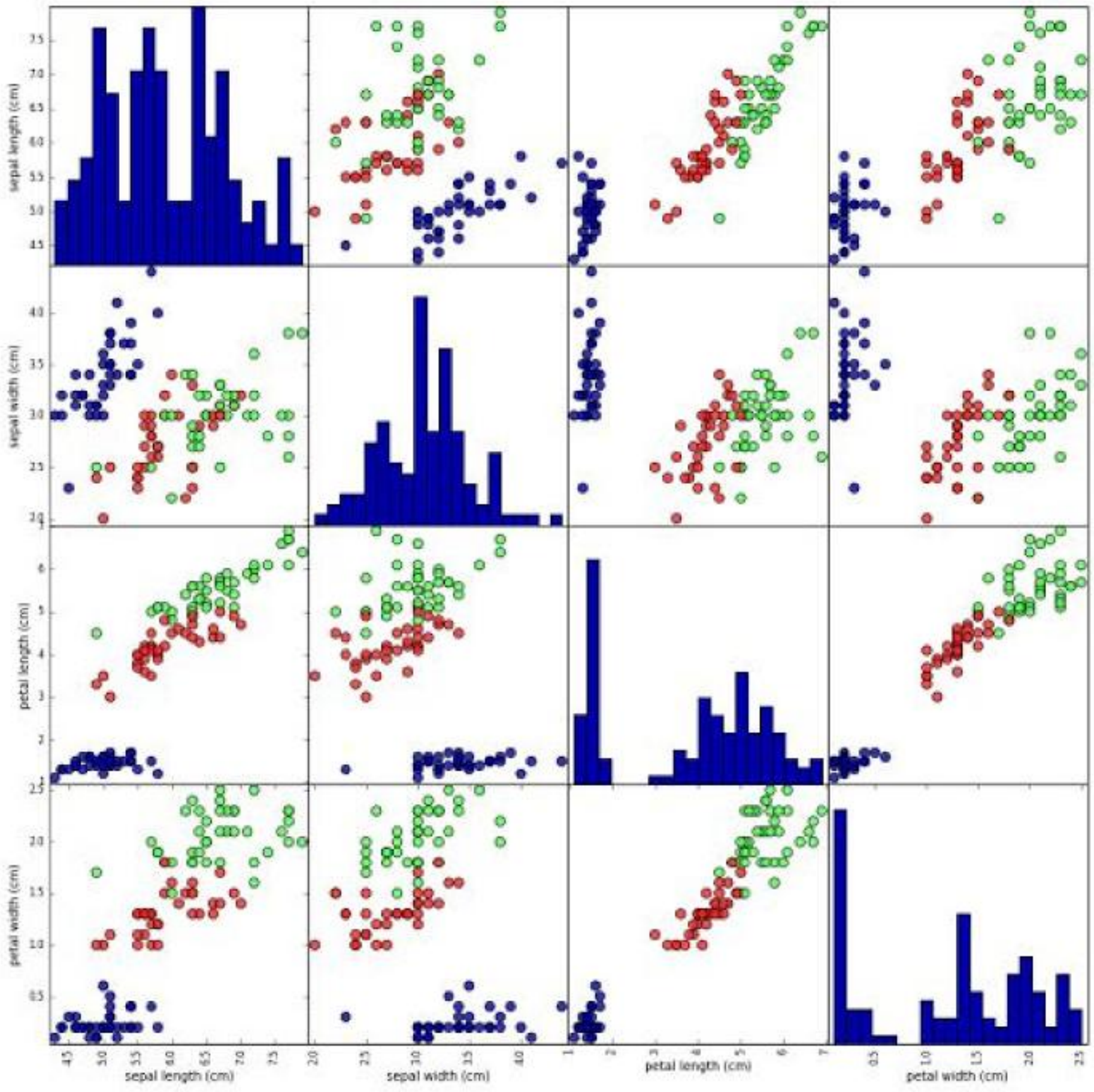


Figure 1-3. Pair plot of the Iris dataset, colored by class label

From the plots, we can see that the three classes seem to be relatively well separated using the sepal and petal measurements. This means that a machine learning model will likely be able to learn to separate them.

Building Your First Model: k-Nearest Neighbors

Now we can start building the actual machine learning model. There are many classification algorithms in `scikit-learn` that we could use. Here we will use a *k*-nearest neighbors classifier, which is easy to understand. Building this model only consists of storing the training set. To make a prediction for a new data point, the algorithm finds the point in the training set that is closest to the new point. Then it assigns the label of this training point to the new data point.

The *k* in *k*-nearest neighbors signifies that instead of using only the closest neighbor to the new data point, we can consider any fixed number *k* of neighbors in the training (for example, the closest three or five neighbors). Then, we can make a prediction using the majority class among these neighbors. We will go into more detail about this in Chapter 2; for now, we'll use only a single neighbor.

All machine learning models in `scikit-learn` are implemented in their own classes, which are called `Estimator` classes. The *k*-nearest neighbors classification algorithm is implemented in the `KNeighborsClassifier` class in the `neighbors` module. Before we can use the model, we need to instantiate the class into an object. This is when we will set any parameters of the model. The most important parameter of `KNeighborsClassifier` is the number of neighbors, which we will set to 1:

In[24]:

```
from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier(n_neighbors=1)
```

The `knn` object encapsulates the algorithm that will be used to build the model from the training data, as well the algorithm to make predictions on new data points. It will also hold the information that the algorithm has

extracted from the training data. In the case of `KNeighborsClassifier`, it will just store the training set.

To build the model on the training set, we call the `fit` method of the `knn` object, which takes as arguments the NumPy array `X_train` containing the training data and the NumPy array `y_train` of the corresponding training labels:

In[25]:

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

Out[25]:

```
KNeighborsClassifier(algorithm='auto', leaf_size=30,
metric='minkowski',
metric_params=None, n_jobs=1, n_neighbors=1, p=2,
weights='uniform')
```

The `fit` method returns the `knn` object itself (and modifies it in place), so we get a string representation of our classifier. The representation shows us which parameters were used in creating the model. Nearly all of them are the default values, but you can also find `n_neighbors=1`, which is the parameter that we passed. Most models in `scikit-learn` have many parameters, but the majority of them are either speed optimizations or for very special use cases. You don't have to worry about the other parameters shown in this representation. Printing a `scikit-learn` model can yield very long strings, but don't be intimidated by these. We will cover all the important parameters in Chapter 2. In the remainder of this book, we will not show the output of `fit` because it doesn't contain any new information.

Making Predictions

We can now make predictions using this model on new data for which we might not know the correct labels. Imagine we found an iris in the wild with a sepal length of 5 cm, a sepal width of 2.9 cm, a petal length of 1 cm, and a petal width of 0.2 cm. What species of iris would this be? We can put this data into a NumPy array, again by calculating the shape—that is, the number of samples (1) multiplied by the number of features (4):

In[26]:

```
X_new = np.array([[5, 2.9, 1, 0.2]])
print("X_new.shape: {}".format(X_new.shape))
```

Out[26]:

```
X_new.shape: (1, 4)
```

Note that we made the measurements of this single flower into a row in a two-dimensional NumPy array, as `scikit-learn` always expects two-dimensional arrays for the data.

To make a prediction, we call the `predict` method of the `knn` object:

In[27]:

```
prediction = knn.predict(X_new)
print("Prediction: {}".format(prediction))
print("Predicted target name: {}".format(
       iris_dataset['target_names'][prediction]))
```

Out[27]:

```
Prediction: [0]
Predicted target name: ['setosa']
```

Our model predicts that this new iris belongs to the class 0, meaning its species is *setosa*. But how do we know whether we can trust our model? We don't know the correct species of this sample, which is the whole point of building the model!

Evaluating the Model

This is where the test set that we created earlier comes in. This data was not used to build the model, but we do know what the correct species is for each iris in the test set.

Therefore, we can make a prediction for each iris in the test data and compare it against its label (the known species). We can measure how well the model works by computing the *accuracy*, which is the fraction of flowers for which the right species was predicted:

In[28]:

```
y_pred = knn.predict(X_test)
print("Test set predictions:\n {}".format(y_pred))
```

Out[28]:

```
Test set predictions:
 [2 1 0 2 0 2 0 1 1 1 2 1 1 1 1 0 1 1 0 0 2 1 0 0 2 0 0 1 1 0 2 1
0 2 2 1 0 2]
```

In[29]:

```
print("Test set score: {:.2f}".format(np.mean(y_pred == y_test)))
```

Out[29]:

```
Test set score: 0.97
```

We can also use the `score` method of the `knn` object, which will compute the test set accuracy for us:

In[30]:

```
print("Test set score: {:.2f}".format(knn.score(X_test, y_test)))
```

Out[30]:

```
Test set score: 0.97
```

For this model, the test set accuracy is about 0.97, which means we made the right prediction for 97% of the irises in the test set. Under some mathematical assumptions, this means that we can expect our model to be correct 97% of the time for new irises. For our hobby botanist application, this high level of accuracy means that our model may be trustworthy enough to use. In later chapters we will discuss how we can improve performance, and what caveats there are in tuning a model.

Summary and Outlook

Let's summarize what we learned in this chapter. We started with a brief introduction to machine learning and its applications, then discussed the distinction between supervised and unsupervised learning and gave an overview of the tools we'll be using in this book. Then, we formulated the task of predicting which species of iris a particular flower belongs to by using physical measurements of the flower. We used a dataset of

measurements that was annotated by an expert with the correct species to build our model, making this a supervised learning task. There were three possible species, *setosa*, *versicolor*, or *virginica*, which made the task a three-class classification problem. The possible species are called *classes* in the classification problem, and the species of a single iris is called its *label*.

The Iris dataset consists of two NumPy arrays: one containing the data, which is referred to as `X` in `scikit-learn`, and one containing the correct or desired outputs, which is called `y`. The array `X` is a two-dimensional array of features, with one row per data point and one column per feature. The array `y` is a one-dimensional array, which here contains one class label, an integer ranging from 0 to 2, for each of the samples.

We split our dataset into a *training set*, to build our model, and a *test set*, to evaluate how well our model will generalize to new, previously unseen data.

We chose the *k*-nearest neighbors classification algorithm, which makes predictions for a new data point by considering its closest neighbor(s) in the training set. This is implemented in the `KNeighborsClassifier` class, which contains the algorithm that builds the model as well as the algorithm that makes a prediction using the model. We instantiated the class, setting parameters. Then we built the model by calling the `fit` method, passing the training data (`X_train`) and training outputs (`y_train`) as parameters. We evaluated the model using the `score` method, which computes the accuracy of the model. We applied the `score` method to the test set data and the test set labels and found that our model is about 97% accurate, meaning it is correct 97% of the time on the test set.

This gave us the confidence to apply the model to new data (in our example, new flower measurements) and trust that the model will be correct

about 97% of the time.

Here is a summary of the code needed for the whole training and evaluation procedure:

In[31]:

```
X_train, X_test, y_train, y_test = train_test_split(
    iris_dataset['data'], iris_dataset['target'], random_state=0)

knn = KNeighborsClassifier(n_neighbors=1)
knn.fit(X_train, y_train)

print("Test set score: {:.2f}".format(knn.score(X_test, y_test)))
```

Out[31]:

```
Test set score: 0.97
```

This snippet contains the core code for applying any machine learning algorithm using `scikit-learn`. The `fit`, `predict`, and `score` methods are the common interface to supervised models in `scikit-learn`, and with the concepts introduced in this chapter, you can apply these models to many machine learning tasks. In the next chapter, we will go into more depth about the different kinds of supervised models in `scikit-learn` and how to apply them successfully.

[1] If you are unfamiliar with NumPy or `matplotlib`, we recommend reading the first chapter of the SciPy Lecture Notes.

[2] The `six` package can be very handy for that.

Chapter 2. Supervised Learning

As we mentioned earlier, supervised machine learning is one of the most commonly used and successful types of machine learning. In this chapter, we will describe supervised learning in more detail and explain several popular supervised learning algorithms. We already saw an application of supervised machine learning in Chapter 1: classifying iris flowers into several species using physical measurements of the flowers.

Remember that supervised learning is used whenever we want to predict a certain outcome from a given input, and we have examples of input/output pairs. We build a machine learning model from these input/output pairs, which comprise our training set. Our goal is to make accurate predictions for new, never-before-seen data. Supervised learning often requires human effort to build the training set, but afterward automates and often speeds up an otherwise laborious or infeasible task.

Classification and Regression

There are two major types of supervised machine learning problems, called *classification* and *regression*.

In classification, the goal is to predict a *class label*, which is a choice from a predefined list of possibilities. In Chapter 1 we used the example of classifying irises into one of three possible species. Classification is sometimes separated into *binary classification*, which is the special case of distinguishing between exactly two classes, and *multiclass classification*, which is classification between more than two classes. You can think of binary classification as trying to answer a yes/no question. Classifying emails as either spam or not spam is an example of a binary classification problem. In this binary classification task, the yes/no question being asked would be “Is this email spam?”

Note

In binary classification we often speak of one class being the *positive* class and the other class being the *negative* class. Here, positive doesn't represent having benefit or value, but rather what the object of the study is. So, when looking for spam, "positive" could mean the spam class. Which of the two classes is called positive is often a subjective matter, and specific to the domain.

The iris example, on the other hand, is an example of a multiclass classification problem. Another example is predicting what language a website is in from the text on the website. The classes here would be a pre-defined list of possible languages.

For regression tasks, the goal is to predict a continuous number, or a *floating-point number* in programming terms (or *real number* in mathematical terms). Predicting a person's annual income from their education, their age, and where they live is an example of a regression task. When predicting income, the predicted value is an *amount*, and can be any number in a given range. Another example of a regression task is predicting the yield of a corn farm given attributes such as previous yields, weather, and number of employees working on the farm. The yield again can be an arbitrary number.

An easy way to distinguish between classification and regression tasks is to ask whether there is some kind of continuity in the output. If there is continuity between possible outcomes, then the problem is a regression problem. Think about predicting annual income. There is a clear continuity in the output. Whether a person makes $40,000 or $40,001 a year does not make a tangible difference, even though these are different amounts of money; if our algorithm predicts $39,999 or $40,001 when it should have predicted $40,000, we don't mind that much.

By contrast, for the task of recognizing the language of a website (which is

a classification problem), there is no matter of degree. A website is in one language, or it is in another. There is no continuity between languages, and there is no language that is *between* English and French.[1]

Generalization, Overfitting, and Underfitting

In supervised learning, we want to build a model on the training data and then be able to make accurate predictions on new, unseen data that has the same characteristics as the training set that we used. If a model is able to make accurate predictions on unseen data, we say it is able to *generalize* from the training set to the test set. We want to build a model that is able to generalize as accurately as possible.

Usually we build a model in such a way that it can make accurate predictions on the training set. If the training and test sets have enough in common, we expect the model to also be accurate on the test set. However, there are some cases where this can go wrong. For example, if we allow ourselves to build very complex models, we can always be as accurate as we like on the training set.

Let's take a look at a made-up example to illustrate this point. Say a novice data scientist wants to predict whether a customer will buy a boat, given records of previous boat buyers and customers who we know are not interested in buying a boat.[2] The goal is to send out promotional emails to people who are likely to actually make a purchase, but not bother those customers who won't be interested.

Suppose we have the customer records shown in Table 2-1.

Table 2-1. Example data about customers

Age	Number of cars owned	Owns house	Number of children	Marital status	Owns a dog	Bought a boat
66	1	yes	2	widowed	no	yes
52	2	yes	3	married	no	yes
22	0	no	0	married	yes	no
25	1	no	1	single	no	no
44	0	no	2	divorced	yes	no
39	1	yes	2	married	yes	no
26	1	no	2	single	no	no
40	3	yes	1	married	yes	no
53	2	yes	2	divorced	no	yes
64	2	yes	3	divorced	no	no
58	2	yes	2	married	yes	yes
33	1	no	1	single	no	no

After looking at the data for a while, our novice data scientist comes up with the following rule: “If the customer is older than 45, and has less than 3 children or is not divorced, then they want to buy a boat.” When asked how well this rule of his does, our data scientist answers, “It’s 100 percent

accurate!" And indeed, on the data that is in the table, the rule is perfectly accurate. There are many possible rules we could come up with that would explain perfectly if someone in this dataset wants to buy a boat. No age appears twice in the data, so we could say people who are 66, 52, 53, or 58 years old want to buy a boat, while all others don't. While we can make up many rules that work well on this data, remember that we are not interested in making predictions for this dataset; we already know the answers for these customers. We want to know if *new customers* are likely to buy a boat. We therefore want to find a rule that will work well for new customers, and achieving 100 percent accuracy on the training set does not help us there. We might not expect that the rule our data scientist came up with will work very well on new customers. It seems too complex, and it is supported by very little data. For example, the "or is not divorced" part of the rule hinges on a single customer.

The only measure of whether an algorithm will perform well on new data is the evaluation on the test set. However, intuitively[3] we expect simple models to generalize better to new data. If the rule was "People older than 50 want to buy a boat," and this would explain the behavior of all the customers, we would trust it more than the rule involving children and marital status in addition to age. Therefore, we always want to find the simplest model. Building a model that is too complex for the amount of information we have, as our novice data scientist did, is called *overfitting*. Overfitting occurs when you fit a model too closely to the particularities of the training set and obtain a model that works well on the training set but is not able to generalize to new data. On the other hand, if your model is too simple—say, "Everybody who owns a house buys a boat"—then you might not be able to capture all the aspects of and variability in the data, and your model will do badly even on the training set. Choosing too simple a model is called *underfitting*.

The more complex we allow our model to be, the better we will be able to predict on the training data. However, if our model becomes too complex, we start focusing too much on each individual data point in our training set, and the model will not generalize well to new data.

There is a sweet spot in between that will yield the best generalization performance. This is the model we want to find.

The trade-off between overfitting and underfitting is illustrated in Figure 2-1.

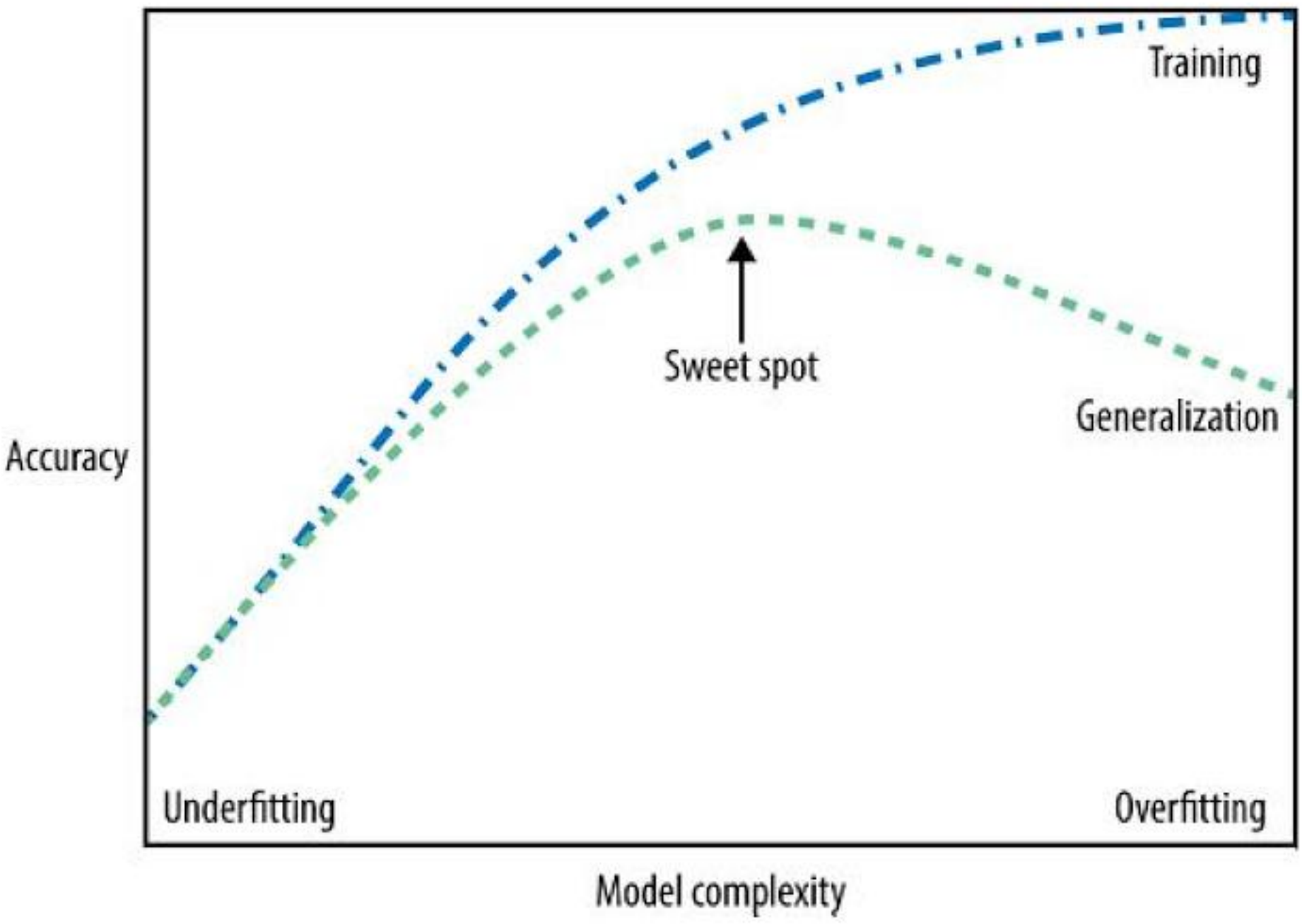


Figure 2-1. Trade-off of model complexity against training and test accuracy

Relation of Model Complexity to Dataset Size

It's important to note that model complexity is intimately tied to the

variation of inputs contained in your training dataset: the larger variety of data points your dataset contains, the more complex a model you can use without overfitting. Usually, collecting more data points will yield more variety, so larger datasets allow building more complex models. However, simply duplicating the same data points or collecting very similar data will not help.

Going back to the boat selling example, if we saw 10,000 more rows of customer data, and all of them complied with the rule "If the customer is older than 45, and has less than 3 children or is not divorced, then they want to buy a boat," we would be much more likely to believe this to be a good rule than when it was developed using only the 12 rows in Table 2-1.

Having more data and building appropriately more complex models can often work wonders for supervised learning tasks. In this book, we will focus on working with datasets of fixed sizes. In the real world, you often have the ability to decide how much data to collect, which might be more beneficial than tweaking and tuning your model. Never underestimate the power of more data.

Supervised Machine Learning Algorithms

We will now review the most popular machine learning algorithms and explain how they learn from data and how they make predictions. We will also discuss how the concept of model complexity plays out for each of these models, and provide an overview of how each algorithm builds a model. We will examine the strengths and weaknesses of each algorithm, and what kind of data they can best be applied to. We will also explain the meaning of the most important parameters and options.[4] Many algorithms have a classification and a regression variant, and we will describe both.

It is not necessary to read through the descriptions of each algorithm in detail, but understanding the models will give you a better feeling for the

different ways machine learning algorithms can work. This chapter can also be used as a reference guide, and you can come back to it when you are unsure about the workings of any of the algorithms.

Some Sample Datasets

We will use several datasets to illustrate the different algorithms. Some of the datasets will be small and synthetic (meaning made-up), designed to highlight particular aspects of the algorithms. Other datasets will be large, real-world examples.

An example of a synthetic two-class classification dataset is the `forge` dataset, which has two features. The following code creates a scatter plot (Figure 2-2) visualizing all of the data points in this dataset. The plot has the first feature on the x-axis and the second feature on the y-axis. As is always the case in scatter plots, each data point is represented as one dot. The color and shape of the dot indicates its class:

In[1]:

```
X, y = mglearn.datasets.make_forge()

mglearn.discrete_scatter(X[:, 0], X[:, 1], y)
plt.legend(["Class 0", "Class 1"], loc=4)
plt.xlabel("First feature")
plt.ylabel("Second feature")
print("X.shape: {}".format(X.shape))
```

Out[1]:

```
X.shape: (26, 2)
```

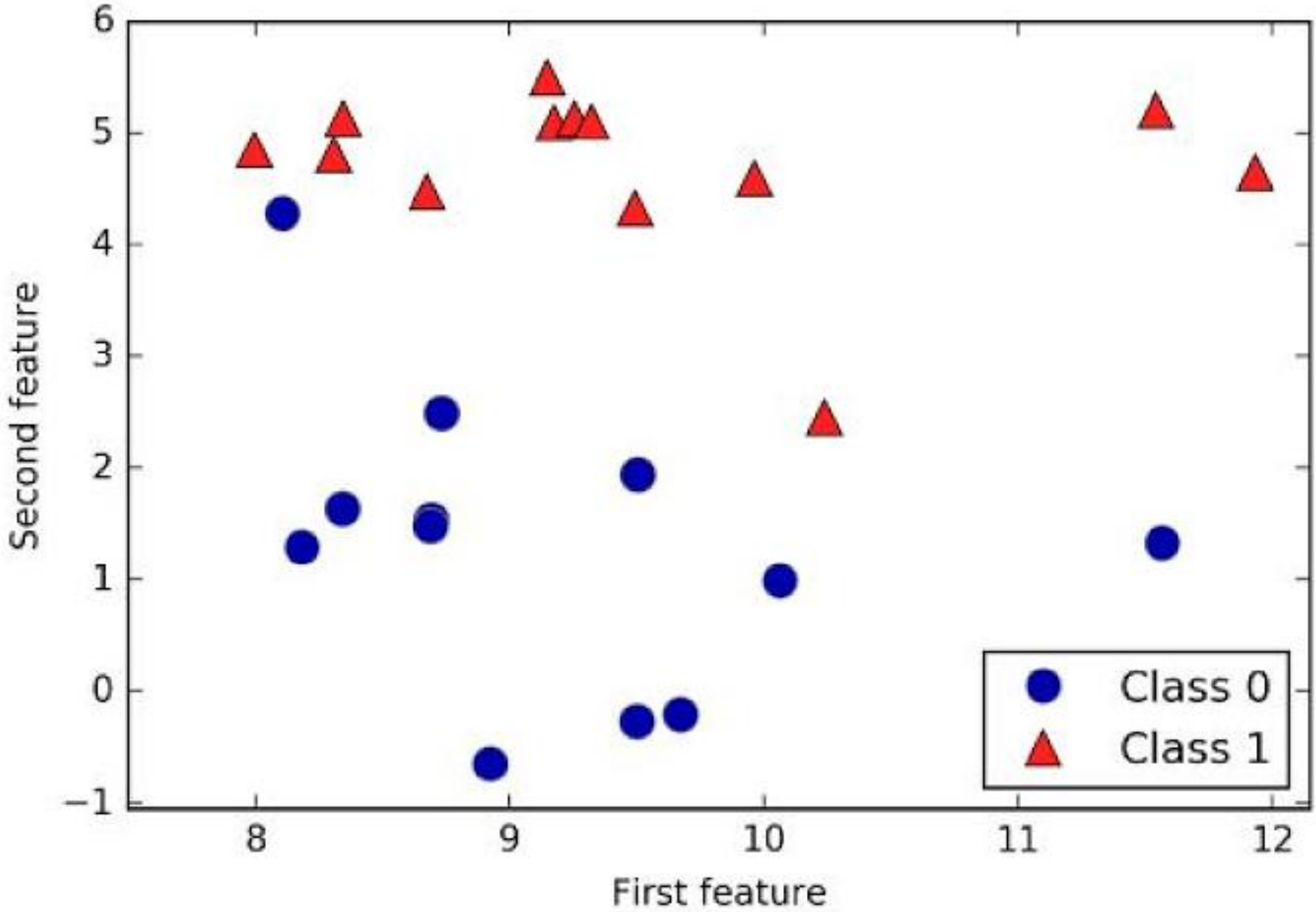


Figure 2-2. Scatter plot of the forge dataset

As you can see from `X.shape`, this dataset consists of 26 data points, with 2 features.

To illustrate regression algorithms, we will use the synthetic `wave` dataset. The `wave` dataset has a single input feature and a continuous target variable (or *response*) that we want to model. The plot created here (Figure 2-3) shows the single feature on the x-axis and the regression target (the output) on the y-axis:

`In[2]:`

```
X, y = mglearn.datasets.make_wave(n_samples=40)
```

```
plt.plot(X, y, 'o')
plt.ylim(-3, 3)
plt.xlabel("Feature")
plt.ylabel("Target")
```

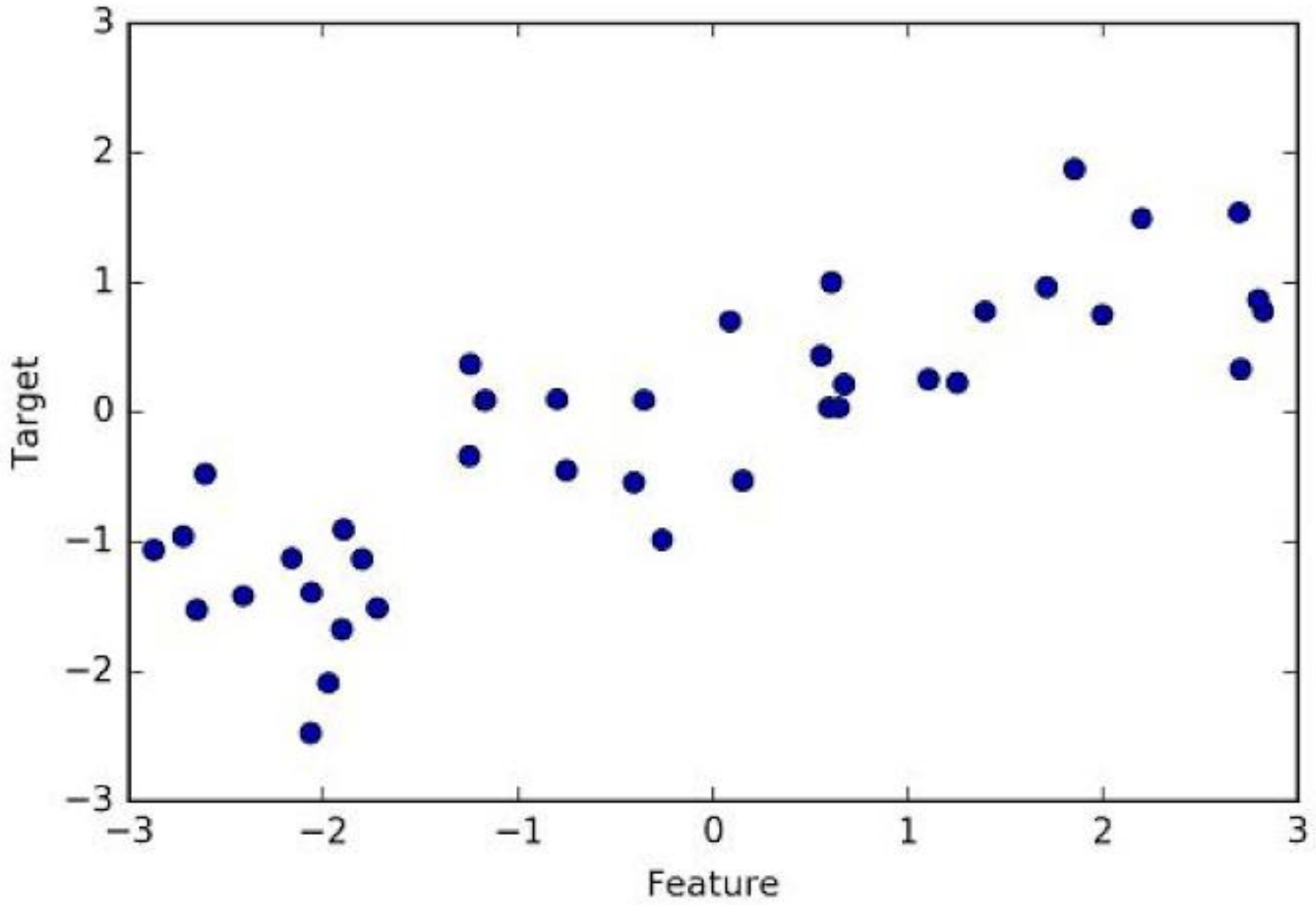


Figure 2-3. Plot of the wave dataset, with the x-axis showing the feature and the y-axis showing the regression target

We are using these very simple, low-dimensional datasets because we can easily visualize them—a printed page has two dimensions, so data with more than two features is hard to show. Any intuition derived from datasets with few features (also called *low-dimensional* datasets) might not hold in datasets with many features (*high-dimensional* datasets). As long as you keep that in mind, inspecting algorithms on low-dimensional datasets can be very instructive.

We will complement these small synthetic datasets with two real-world datasets that are included in `scikit-learn`. One is the Wisconsin Breast Cancer dataset (`cancer`, for short), which records clinical measurements of breast cancer tumors. Each tumor is labeled as "benign" (for harmless tumors) or "malignant" (for cancerous tumors), and the task is to learn to predict whether a tumor is malignant based on the measurements of the tissue.

The data can be loaded using the `load_breast_cancer` function from `scikit-learn`:

In[3]:

```
from sklearn.datasets import load_breast_cancer
cancer = load_breast_cancer()
print("cancer.keys(): \n{}".format(cancer.keys()))
```

Out[3]:

```
cancer.keys():
dict_keys(['feature_names', 'data', 'DESCR', 'target',
'target_names'])
```

Note

Datasets that are included in `scikit-learn` are usually stored as `Bunch` objects, which contain some information about the dataset as well as the actual data. All you need to know about `Bunch` objects is that they behave like dictionaries, with the added benefit that you can access values using a dot (as in `bunch.key` instead of `bunch['key']`).

The dataset consists of 569 data points, with 30 features each:

In[4]:

```
print("Shape of cancer data: {}".format(cancer.data.shape))
```

Out[4]:

```
Shape of cancer data: (569, 30)
```

Of these 569 data points, 212 are labeled as malignant and 357 as benign:

In[5]:

```
print("Sample counts per class:\n{}".format(
      {n: v for n, v in zip(cancer.target_names,
np.bincount(cancer.target))}))
```

Out[5]:

```
Sample counts per class:
{'benign': 357, 'malignant': 212}
```

To get a description of the semantic meaning of each feature, we can have a look at the `feature_names` attribute:

In[6]:

```
print("Feature names:\n{}".format(cancer.feature_names))
```

Out[6]:

```
Feature names:
['mean radius' 'mean texture' 'mean perimeter' 'mean area'
 'mean smoothness' 'mean compactness' 'mean concavity'
 'mean concave points' 'mean symmetry' 'mean fractal dimension'
 'radius error' 'texture error' 'perimeter error' 'area error'
 'smoothness error' 'compactness error' 'concavity error'
 'concave points error' 'symmetry error' 'fractal dimension error'
 'worst radius' 'worst texture' 'worst perimeter' 'worst area'
 'worst smoothness' 'worst compactness' 'worst concavity'
 'worst concave points' 'worst symmetry' 'worst fractal
dimension']
```

You can find out more about the data by reading `cancer.DESCR` if you are interested.

We will also be using a real-world regression dataset, the Boston Housing dataset. The task associated with this dataset is to predict the median value of homes in several Boston neighborhoods in the 1970s, using information such as crime rate, proximity to the Charles River, highway accessibility, and so on. The dataset contains 506 data points, described by 13 features:

In[7]:

```
from sklearn.datasets import load_boston
boston = load_boston()
print("Data shape: {}".format(boston.data.shape))
```

Out[7]:

```
Data shape: (506, 13)
```

Again, you can get more information about the dataset by reading the `DESCR` attribute of `boston`. For our purposes here, we will actually expand this dataset by not only considering these 13 measurements as input

features, but also looking at all products (also called *interactions*) between features. In other words, we will not only consider crime rate and highway accessibility as features, but also the product of crime rate and highway accessibility. Including derived feature like these is called *feature engineering*, which we will discuss in more detail in Chapter 4. This derived dataset can be loaded using the `load_extended_boston` function:

In[8]:

```
X, y = mglearn.datasets.load_extended_boston()
print("X.shape: {}".format(X.shape))
```

Out[8]:

```
X.shape: (506, 104)
```

The resulting 104 features are the 13 original features together with the 91 possible combinations of two features within those 13 (with replacement).[5]

We will use these datasets to explain and illustrate the properties of the different machine learning algorithms. But for now, let's get to the algorithms themselves. First, we will revisit the *k*-nearest neighbors (*k*-NN) algorithm that we saw in the previous chapter.

k-Nearest Neighbors

The *k*-NN algorithm is arguably the simplest machine learning algorithm. Building the model consists only of storing the training dataset. To make a prediction for a new data point, the algorithm finds the closest data points in the training dataset—its "nearest neighbors."

k-Neighbors classification

In its simplest version, the *k*-NN algorithm only considers exactly one

The plot shows the training and test set accuracy on the y-axis against the setting of `n_neighbors` on the x-axis. While real-world plots are rarely very smooth, we can still recognize some of the characteristics of overfitting and underfitting (note that because considering fewer neighbors corresponds to a more complex model, the plot is horizontally flipped relative to the illustration in Figure 2-1). Considering a single nearest neighbor, the prediction on the training set is perfect. But when more neighbors are considered, the model becomes simpler and the training accuracy drops. The test set accuracy for using a single neighbor is lower than when using more neighbors, indicating that using the single nearest neighbor leads to a model that is too complex. On the other hand, when considering 10 neighbors, the model is too simple and performance is even worse. The best performance is somewhere in the middle, using around six neighbors. Still, it is good to keep the scale of the plot in mind. The worst performance is around 88% accuracy, which might still be acceptable.

features, with weights (which can be negative) given by the entries of *w*.

Trying to learn the parameters *w*[0] and *b* on our one-dimensional `wave` dataset might lead to the following line (see Figure 2-11):

In[24]:

```
mglearn.plots.plot_linear_regression_wave()
```

Out[24]:

```
w[0]: 0.393906  b: -0.031804
```

have a single input feature in the `wave` dataset, `lr.coef_` only has a single entry.

Let's look at the training set and test set performance:

In[27]:

```
print("Training set score: {:.2f}".format(lr.score(X_train,
y_train)))
print("Test set score: {:.2f}".format(lr.score(X_test, y_test)))
```

Out[27]:

```
Training set score: 0.67
Test set score: 0.66
```

An R^2 of around 0.66 is not very good, but we can see that the scores on the training and test sets are very close together. This means we are likely underfitting, not overfitting. For this one-dimensional dataset, there is little danger of overfitting, as the model is very simple (or restricted). However, with higher-dimensional datasets (meaning datasets with a large number of features), linear models become more powerful, and there is a higher chance of overfitting. Let's take a look at how `LinearRegression` performs on a more complex dataset, like the Boston Housing dataset. Remember that this dataset has 506 samples and 105 derived features. First, we load the dataset and split it into a training and a test set. Then we build the linear regression model as before:

In[28]:

```
X, y = mglearn.datasets.load_extended_boston()
```

```
print("Test set score: {:.2f}".format(ridge01.score(X_test,
y_test)))
```

Out[32]:

```
Training set score: 0.93
Test set score: 0.77
```

Here, `alpha=0.1` seems to be working well. We could try decreasing `alpha` even more to improve generalization. For now, notice how the parameter `alpha` corresponds to the model complexity as shown in Figure 2-1. We will discuss methods to properly select parameters in Chapter 5.

We can also get a more qualitative insight into how the `alpha` parameter changes the model by inspecting the `coef_` attribute of models with different values of `alpha`. A higher `alpha` means a more restricted model, so we expect the entries of `coef_` to have smaller magnitude for a high value of `alpha` than for a low value of `alpha`. This is confirmed in the plot in Figure 2-12:

In[33]:

```
plt.plot(ridge.coef_, 's', label="Ridge alpha=1")
plt.plot(ridge10.coef_, '^', label="Ridge alpha=10")
plt.plot(ridge01.coef_, 'v', label="Ridge alpha=0.1")

plt.plot(lr.coef_, 'o', label="LinearRegression")
plt.xlabel("Coefficient index")
plt.ylabel("Coefficient magnitude")
plt.hlines(0, 0, len(lr.coef_))
plt.ylim(-25, 25)
plt.legend()
```

enough data, ridge and linear regression will have the same performance (the fact that this happens here when using the full dataset is just by chance). Another interesting aspect of Figure 2-13 is the decrease in training performance for linear regression. If more data is added, it becomes harder for a model to overfit, or memorize the data.

Lasso

An alternative to `Ridge` for regularizing linear regression is `Lasso`. As with ridge regression, using the lasso also restricts coefficients to be close to zero, but in a slightly different way, called L1 regularization.[8] The consequence of L1 regularization is that when using the lasso, some coefficients are *exactly zero*. This means some features are entirely ignored by the model. This can be seen as a form of automatic feature selection. Having some coefficients be exactly zero often makes a model easier to interpret, and can reveal the most important features of your model.

Let's apply the lasso to the extended Boston Housing dataset:

In[35]:

```
from sklearn.linear_model import Lasso

lasso = Lasso().fit(X_train, y_train)
print("Training set score: {:.2f}".format(lasso.score(X_train,
y_train)))
print("Test set score: {:.2f}".format(lasso.score(X_test,
y_test)))
print("Number of features used: {}".format(np.sum(lasso.coef_ !=
0)))
```

Out[35]:

```
Training set score: 0.29
Test set score: 0.21
```

coefficients nonzero and of large magnitude. For comparison, the best `Ridge` solution is shown as circles. The `Ridge` model with `alpha=0.1` has similar predictive performance as the lasso model with `alpha=0.01`, but using `Ridge`, all coefficients are nonzero.

In practice, ridge regression is usually the first choice between these two models. However, if you have a large amount of features and expect only a few of them to be important, `Lasso` might be a better choice. Similarly, if you would like to have a model that is easy to interpret, `Lasso` will provide a model that is easier to understand, as it will select only a subset of the input features. `scikit-learn` also provides the `ElasticNet` class, which combines the penalties of `Lasso` and `Ridge`. In practice, this combination works best, though at the price of having two parameters to adjust: one for the L1 regularization, and one for the L2 regularization.

Linear models for classification

Linear models are also extensively used for classification. Let's look at binary classification first. In this case, a prediction is made using the following formula:

$$= w[0] * x[0] + w[1] * x[1] + \ldots + w[p] * x[p] + b > 0$$

The formula looks very similar to the one for linear regression, but instead of just returning the weighted sum of the features, we threshold the predicted value at zero. If the function is smaller than zero, we predict the class –1; if it is larger than zero, we predict the class +1. This prediction rule is common to all linear models for classification. Again, there are many different ways to find the coefficients (*w*) and the intercept (*b*).

For linear models for regression, the output, , is a linear function of the features: a line, plane, or hyperplane (in higher dimensions). For linear

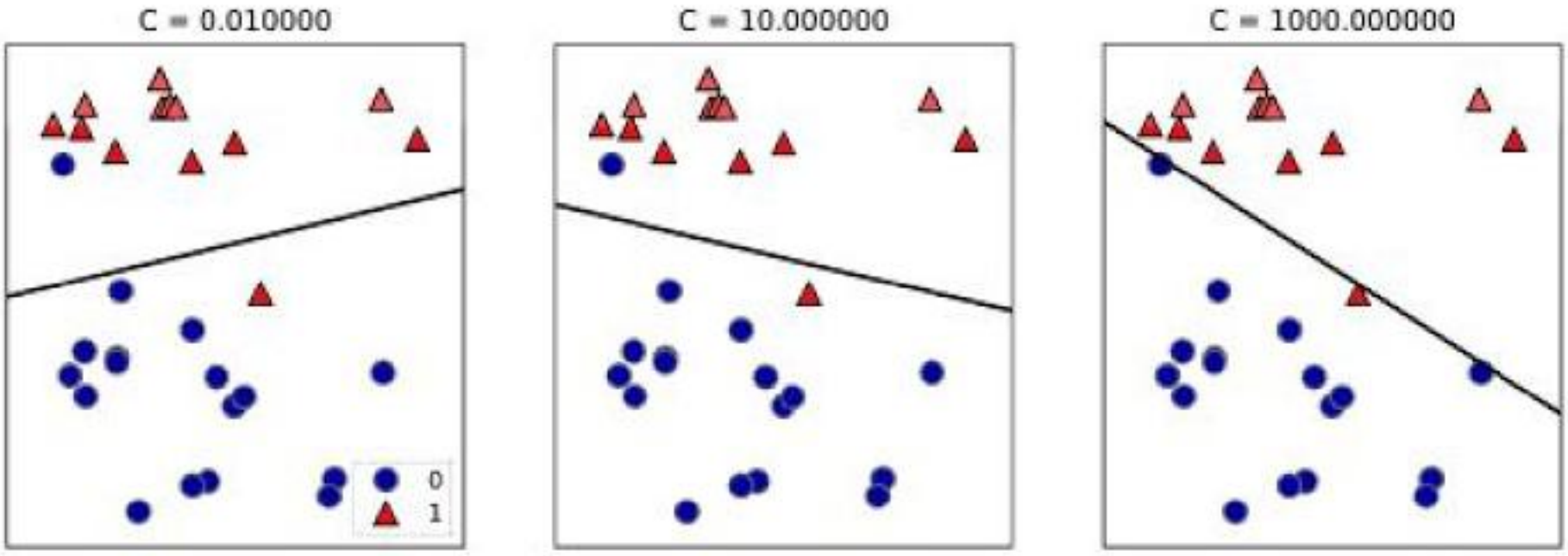


Figure 2-16. Decision boundaries of a linear SVM on the forge dataset for different values of C

On the lefthand side, we have a very small `C` corresponding to a lot of regularization. Most of the points in class 0 are at the bottom, and most of the points in class 1 are at the top. The strongly regularized model chooses a relatively horizontal line, misclassifying two points. In the center plot, `C` is slightly higher, and the model focuses more on the two misclassified samples, tilting the decision boundary. Finally, on the righthand side, the very high value of `C` in the model tilts the decision boundary a lot, now correctly classifying all points in class 0. One of the points in class 1 is still misclassified, as it is not possible to correctly classify all points in this dataset using a straight line. The model illustrated on the righthand side tries hard to correctly classify all points, but might not capture the overall layout of the classes well. In other words, this model is likely overfitting.

Similarly to the case of regression, linear models for classification might seem very restrictive in low-dimensional spaces, only allowing for decision boundaries that are straight lines or planes. Again, in high dimensions, linear models for classification become very powerful, and guarding against overfitting becomes increasingly important when considering more features.

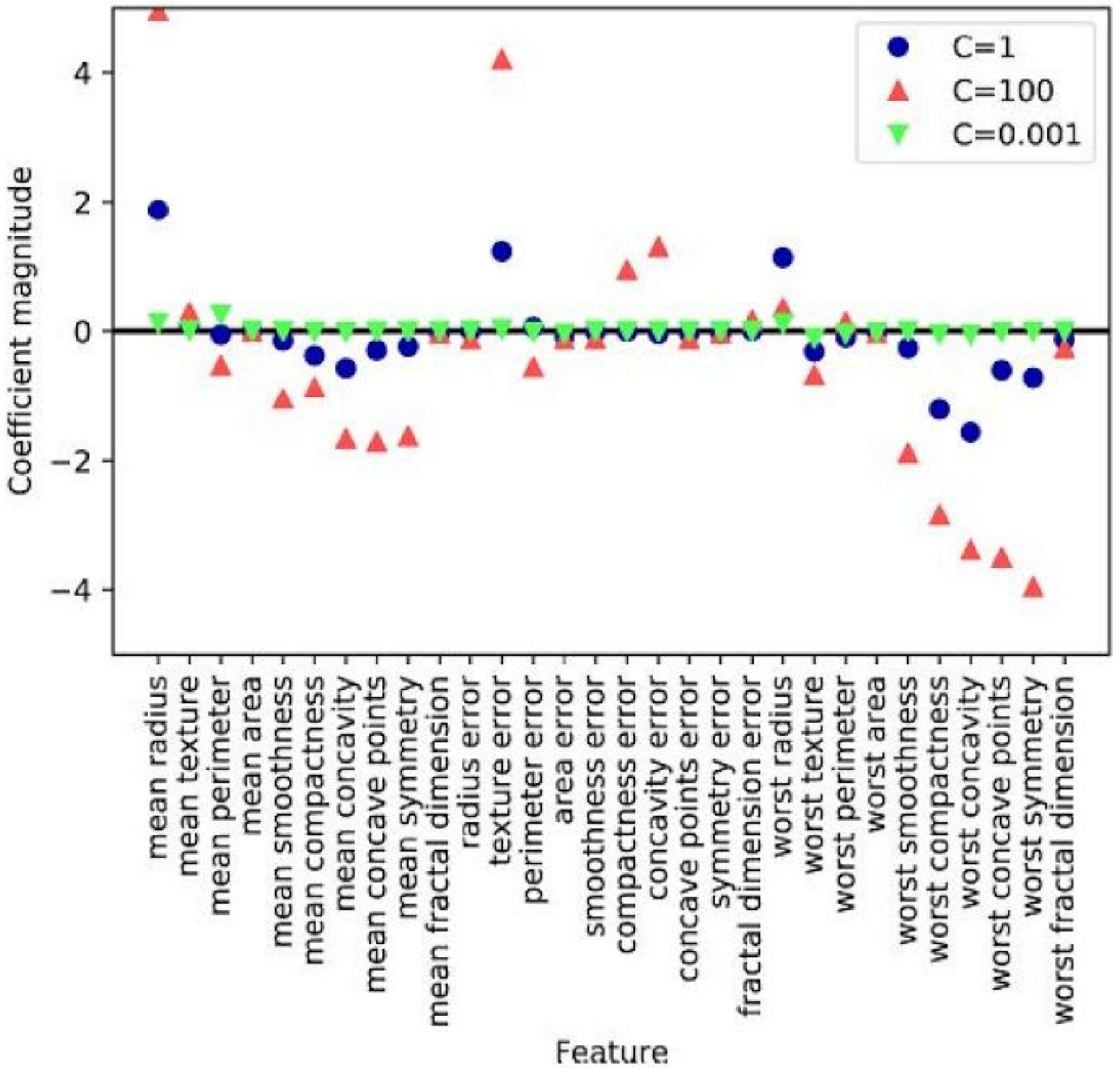


Figure 2-17. Coefficients learned by logistic regression on the Breast Cancer dataset for different values of C

If we desire a more interpretable model, using L1 regularization might help, as it limits the model to using only a few features. Here is the coefficient plot and classification accuracies for L1 regularization (Figure 2-18):

In[45]:

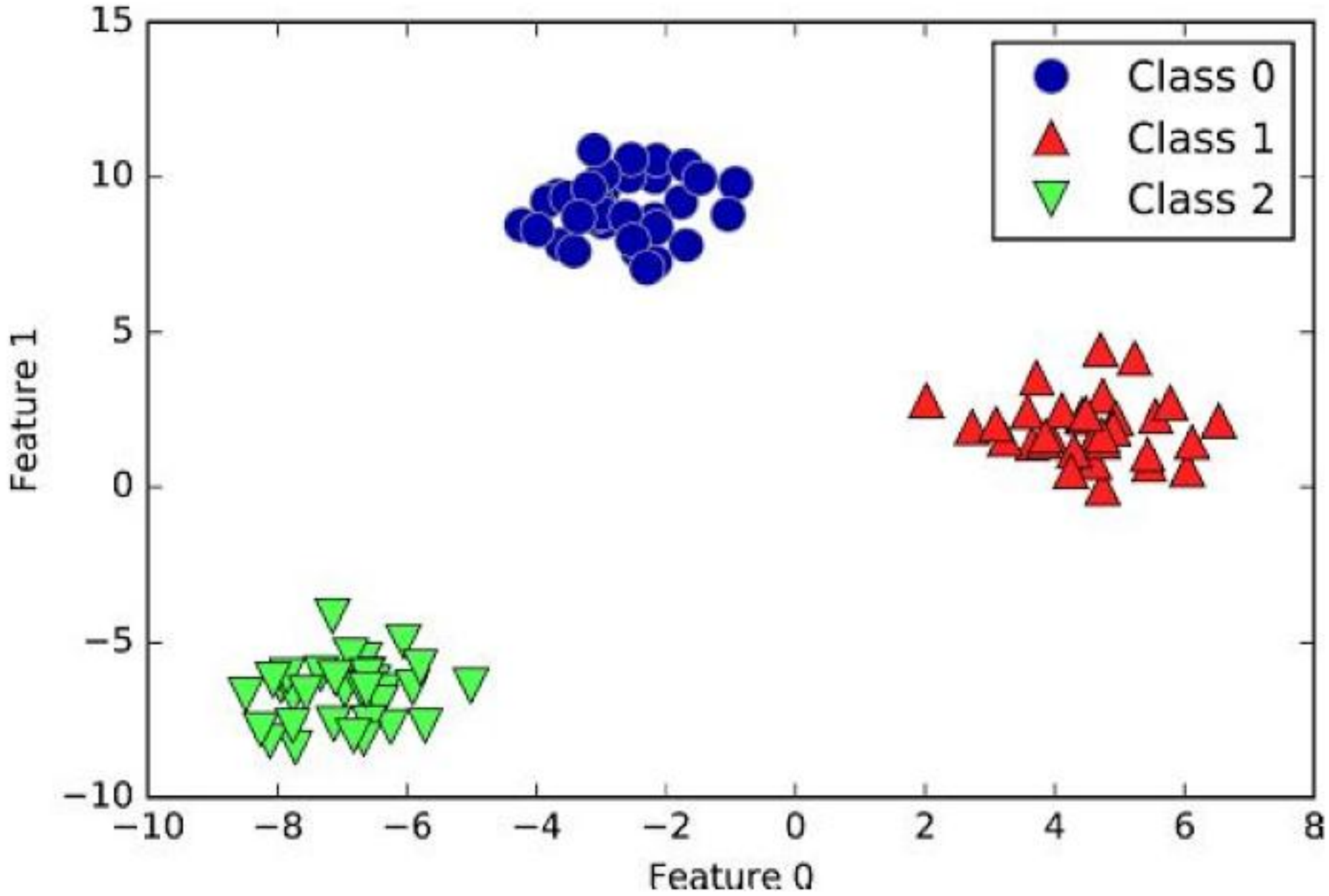


Figure 2-19. Two-dimensional toy dataset containing three classes

Now, we train a `LinearSVC` classifier on the dataset:

In[47]:

```
linear_svm = LinearSVC().fit(X, y)
print("Coefficient shape: ", linear_svm.coef_.shape)
print("Intercept shape: ", linear_svm.intercept_.shape)
```

Out[47]:

```
Coefficient shape:  (3, 2)
Intercept shape:  (3,)
```

Otherwise, you should default to L2. L1 can also be useful if interpretability of the model is important. As L1 will use only a few features, it is easier to explain which features are important to the model, and what the effects of these features are.

Linear models are very fast to train, and also fast to predict. They scale to very large datasets and work well with sparse data. If your data consists of hundreds of thousands or millions of samples, you might want to investigate using the `solver='sag'` option in `LogisticRegression` and `Ridge`, which can be faster than the default on large datasets. Other options are the `SGDClassifier` class and the `SGDRegressor` class, which implement even more scalable versions of the linear models described here.

Another strength of linear models is that they make it relatively easy to understand how a prediction is made, using the formulas we saw earlier for regression and classification. Unfortunately, it is often not entirely clear why coefficients are the way they are. This is particularly true if your dataset has highly correlated features; in these cases, the coefficients might be hard to interpret.

Linear models often perform well when the number of features is large compared to the number of samples. They are also often used on very large datasets, simply because it's not feasible to train other models. However, in lower-dimensional spaces, other models might yield better generalization performance. We will look at some examples in which linear models fail in "Kernelized Support Vector Machines".

Method Chaining

The `fit` method of all `scikit-learn` models returns `self`. This allows you to write code like the following, which we've already used

Strengths, weaknesses, and parameters

`MultinomialNB` and `BernoulliNB` have a single parameter, `alpha`, which controls model complexity. The way `alpha` works is that the algorithm adds to the data `alpha` many virtual data points that have positive values for all the features. This results in a "smoothing" of the statistics. A large `alpha` means more smoothing, resulting in less complex models. The algorithm's performance is relatively robust to the setting of `alpha`, meaning that setting `alpha` is not critical for good performance. However, tuning it usually improves accuracy somewhat.

`GaussianNB` is mostly used on very high-dimensional data, while the other two variants of naive Bayes are widely used for sparse count data such as text. `MultinomialNB` usually performs better than `BernoulliNB`, particularly on datasets with a relatively large number of nonzero features (i.e., large documents).

The naive Bayes models share many of the strengths and weaknesses of the linear models. They are very fast to train and to predict, and the training procedure is easy to understand. The models work very well with high-dimensional sparse data and are relatively robust to the parameters. Naive Bayes models are great baseline models and are often used on very large datasets, where training even a linear model might take too long.

Decision Trees

Decision trees are widely used models for classification and regression tasks. Essentially, they learn a hierarchy of if/else questions, leading to a decision.

These questions are similar to the questions you might ask in a game of 20 Questions. Imagine you want to distinguish between the following four animals: bears, hawks, penguins, and dolphins. Your goal is to get to the

one that is most informative about the target variable. Figure 2-24 shows the first test that is picked. Splitting the dataset horizontally at `x[1]=0.0596` yields the most information; it best separates the points in class 0 from the points in class 1. The top node, also called the *root*, represents the whole dataset, consisting of 75 points belonging to class 0 and 75 points belonging to class 1. The split is done by testing whether `x[1] <= 0.0596`, indicated by a black line. If the test is true, a point is assigned to the left node, which contains 2 points belonging to class 0 and 32 points belonging to class 1. Otherwise the point is assigned to the right node, which contains 48 points belonging to class 0 and 18 points belonging to class 1. These two nodes correspond to the top and bottom regions shown in Figure 2-24. Even though the first split did a good job of separating the two classes, the bottom region still contains points belonging to class 0, and the top region still contains points belonging to class 1. We can build a more accurate model by repeating the process of looking for the best test in both regions. Figure 2-25 shows that the most informative next split for the left and the right region is based on `x[0]`.

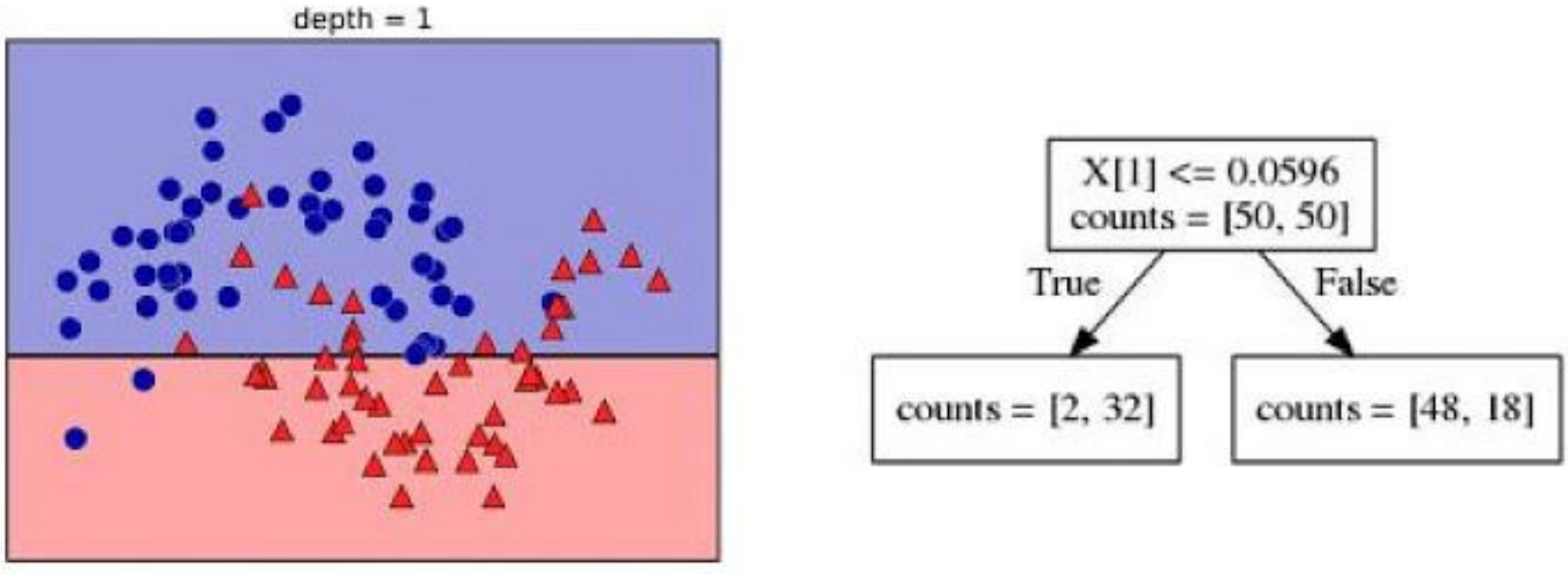


Figure 2-24. Decision boundary of tree with depth 1 (left) and corresponding tree (right)

```
print("Accuracy on test set: {:.3f}".format(tree.score(X_test,
y_test)))
```

Out[56]:

```
Accuracy on training set: 1.000
Accuracy on test set: 0.937
```

As expected, the accuracy on the training set is 100%—because the leaves are pure, the tree was grown deep enough that it could perfectly memorize all the labels on the training data. The test set accuracy is slightly worse than for the linear models we looked at previously, which had around 95% accuracy.

If we don't restrict the depth of a decision tree, the tree can become arbitrarily deep and complex. Unpruned trees are therefore prone to overfitting and not generalizing well to new data. Now let's apply pre-pruning to the tree, which will stop developing the tree before we perfectly fit to the training data. One option is to stop building the tree after a certain depth has been reached. Here we set `max_depth=4`, meaning only four consecutive questions can be asked (cf. Figures 2-24 and 2-26). Limiting the depth of the tree decreases overfitting. This leads to a lower accuracy on the training set, but an improvement on the test set:

In[57]:

```
tree = DecisionTreeClassifier(max_depth=4, random_state=0)
tree.fit(X_train, y_train)

print("Accuracy on training set:
{:.3f}".format(tree.score(X_train, y_train)))
print("Accuracy on test set: {:.3f}".format(tree.score(X_test,
y_test)))
```

```
def plot_feature_importances_cancer(model):
    n_features = cancer.data.shape[1]
    plt.barh(range(n_features), model.feature_importances_,
align='center')
    plt.yticks(np.arange(n_features), cancer.feature_names)
    plt.xlabel("Feature importance")
    plt.ylabel("Feature")
    plt.ylim(-1, n_features)

plot_feature_importances_cancer(tree)
```

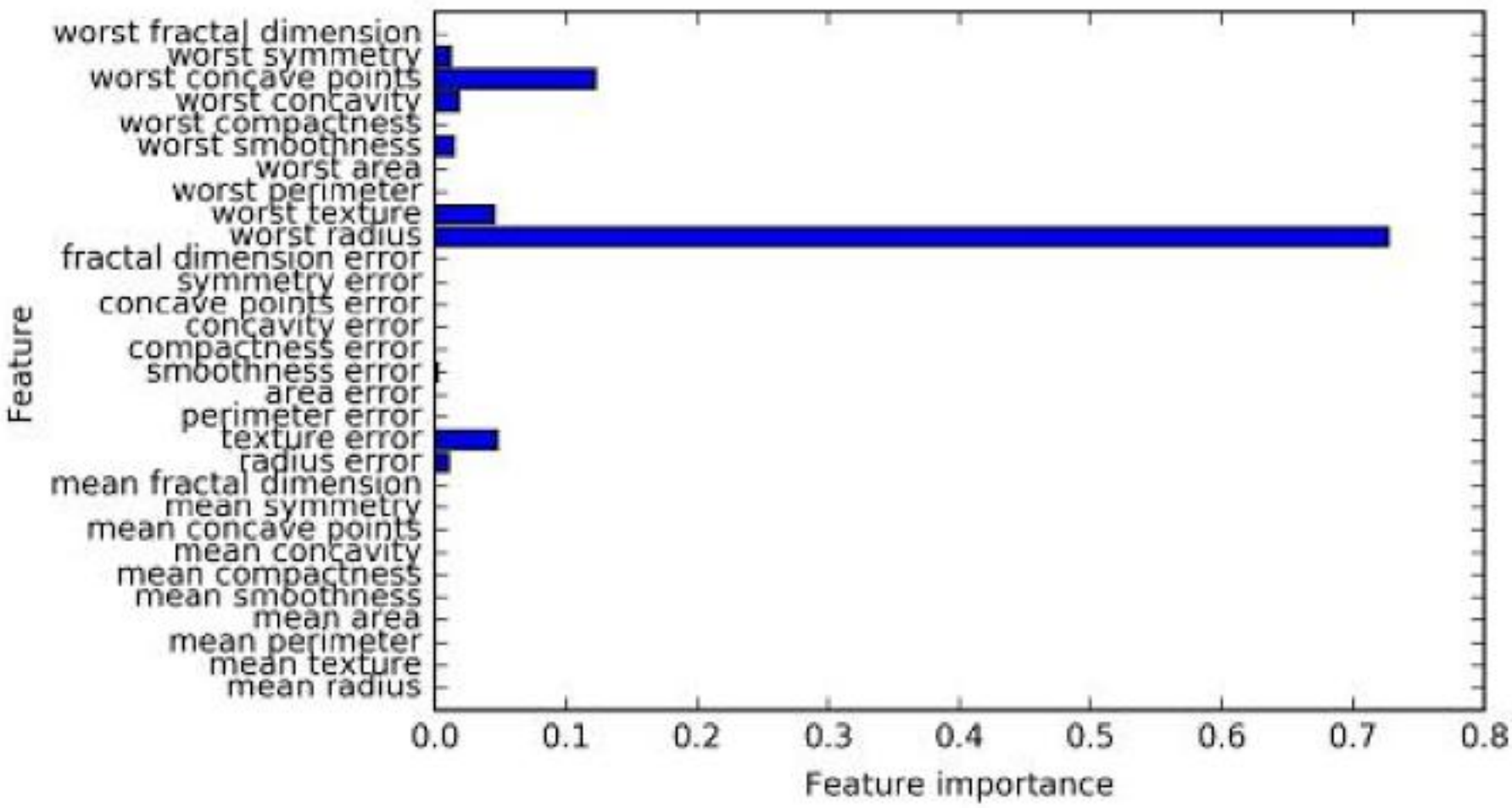


Figure 2-28. Feature importances computed from a decision tree learned on the Breast Cancer dataset

Here we see that the feature used in the top split (“worst radius”) is by far the most important feature. This confirms our observation in analyzing the tree that the first level already separates the two classes fairly well.

However, if a feature has a low value in `feature_importance_`, it doesn’t mean that this feature is uninformative. It only means that the feature was not picked by the tree, likely because another feature encodes the same

all that was said is similarly true for decision trees for regression, as implemented in `DecisionTreeRegressor`. The usage and analysis of regression trees is very similar to that of classification trees. There is one particular property of using tree-based models for regression that we want to point out, though. The `DecisionTreeRegressor` (and all other tree-based regression models) is not able to *extrapolate*, or make predictions outside of the range of the training data.

Let's look into this in more detail, using a dataset of historical computer memory (RAM) prices. Figure 2-31 shows the dataset, with the date on the x-axis and the price of one megabyte of RAM in that year on the y-axis:

In[63]:

```
import os
ram_prices = pd.read_csv(os.path.join(mglearn.datasets.DATA_PATH,
                                      "ram_price.csv"))

plt.semilogy(ram_prices.date, ram_prices.price)
plt.xlabel("Year")
plt.ylabel("Price in $/Mbyte")
```

seen in the training data. This shortcoming applies to all models based on trees.[9]

Strengths, weaknesses, and parameters

As discussed earlier, the parameters that control model complexity in decision trees are the pre-pruning parameters that stop the building of the tree before it is fully developed. Usually, picking one of the pre-pruning strategies—setting either `max_depth`, `max_leaf_nodes`, or `min_samples_leaf`—is sufficient to prevent overfitting.

Decision trees have two advantages over many of the algorithms we've discussed so far: the resulting model can easily be visualized and understood by nonexperts (at least for smaller trees), and the algorithms are completely invariant to scaling of the data. As each feature is processed separately, and the possible splits of the data don't depend on scaling, no preprocessing like normalization or standardization of features is needed for decision tree algorithms. In particular, decision trees work well when you have features that are on completely different scales, or a mix of binary and continuous features.

The main downside of decision trees is that even with the use of pre-pruning, they tend to overfit and provide poor generalization performance. Therefore, in most applications, the ensemble methods we discuss next are usually used in place of a single decision tree.

Ensembles of Decision Trees

Ensembles are methods that combine multiple machine learning models to create more powerful models. There are many models in the machine learning literature that belong to this category, but there are two ensemble models that have proven to be effective on a wide range of datasets for classification and regression, both of which use decision trees as their building blocks: random forests and gradient boosted decision trees.

(Figure 2-33):

In[67]:

```
fig, axes = plt.subplots(2, 3, figsize=(20, 10))
for i, (ax, tree) in enumerate(zip(axes.ravel(),
forest.estimators_)):
    ax.set_title("Tree {}".format(i))
    mglearn.plots.plot_tree_partition(X_train, y_train, tree,
ax=ax)

mglearn.plots.plot_2d_separator(forest, X_train, fill=True,
ax=axes[-1, -1],
                                alpha=.4)
axes[-1, -1].set_title("Random Forest")
mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1], y_train)
```

You can clearly see that the decision boundaries learned by the five trees are quite different. Each of them makes some mistakes, as some of the training points that are plotted here were not actually included in the training sets of the trees, due to the bootstrap sampling.

The random forest overfits less than any of the trees individually, and provides a much more intuitive decision boundary. In any real application, we would use many more trees (often hundreds or thousands), leading to even smoother boundaries.

trees is if you need a compact representation of the decision-making process. It is basically impossible to interpret tens or hundreds of trees in detail, and trees in random forests tend to be deeper than decision trees (because of the use of feature subsets). Therefore, if you need to summarize the prediction making in a visual way to nonexperts, a single decision tree might be a better choice. While building random forests on large datasets might be somewhat time consuming, it can be parallelized across multiple CPU cores within a computer easily. If you are using a multi-core processor (as nearly all modern computers do), you can use the `n_jobs` parameter to adjust the number of cores to use. Using more CPU cores will result in linear speed-ups (using two cores, the training of the random forest will be twice as fast), but specifying `n_jobs` larger than the number of cores will not help. You can set `n_jobs=-1` to use all the cores in your computer.

You should keep in mind that random forests, by their nature, are random, and setting different random states (or not setting the `random_state` at all) can drastically change the model that is built. The more trees there are in the forest, the more robust it will be against the choice of random state. If you want to have reproducible results, it is important to fix the `random_state`.

Random forests don't tend to perform well on very high dimensional, sparse data, such as text data. For this kind of data, linear models might be more appropriate. Random forests usually work well even on very large datasets, and training can easily be parallelized over many CPU cores within a powerful computer. However, random forests require more memory and are slower to train and to predict than linear models. If time and memory are important in an application, it might make sense to use a linear model instead.

The important parameters to adjust are `n_estimators`, `max_features`, and

```
Accuracy on training set: 0.988
Accuracy on test set: 0.965
```

Both methods of decreasing the model complexity reduced the training set accuracy, as expected. In this case, lowering the maximum depth of the trees provided a significant improvement of the model, while lowering the learning rate only increased the generalization performance slightly.

As for the other decision tree–based models, we can again visualize the feature importances to get more insight into our model (Figure 2-35). As we used 100 trees, it is impractical to inspect them all, even if they are all of depth 1:

In[73]:

```
gbrt = GradientBoostingClassifier(random_state=0, max_depth=1)
gbrt.fit(X_train, y_train)

plot_feature_importances_cancer(gbrt)
```

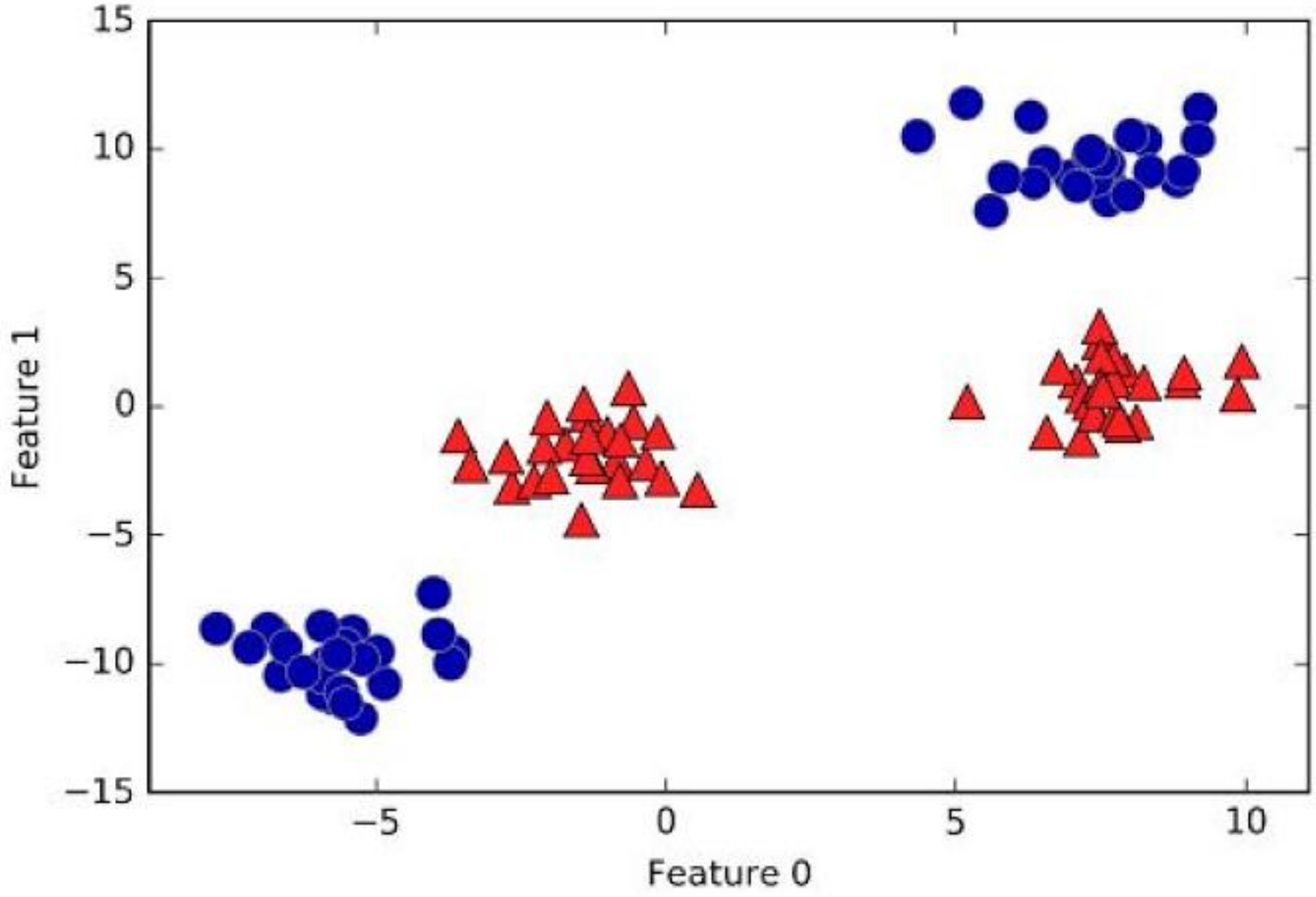


Figure 2-36. Two-class classification dataset in which classes are not linearly separable

A linear model for classification can only separate points using a line, and will not be able to do a very good job on this dataset (see Figure 2-37):

In[75]:

```
from sklearn.svm import LinearSVC
linear_svm = LinearSVC().fit(X, y)

mglearn.plots.plot_2d_separator(linear_svm, X)
mglearn.discrete_scatter(X[:, 0], X[:, 1], y)
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
```

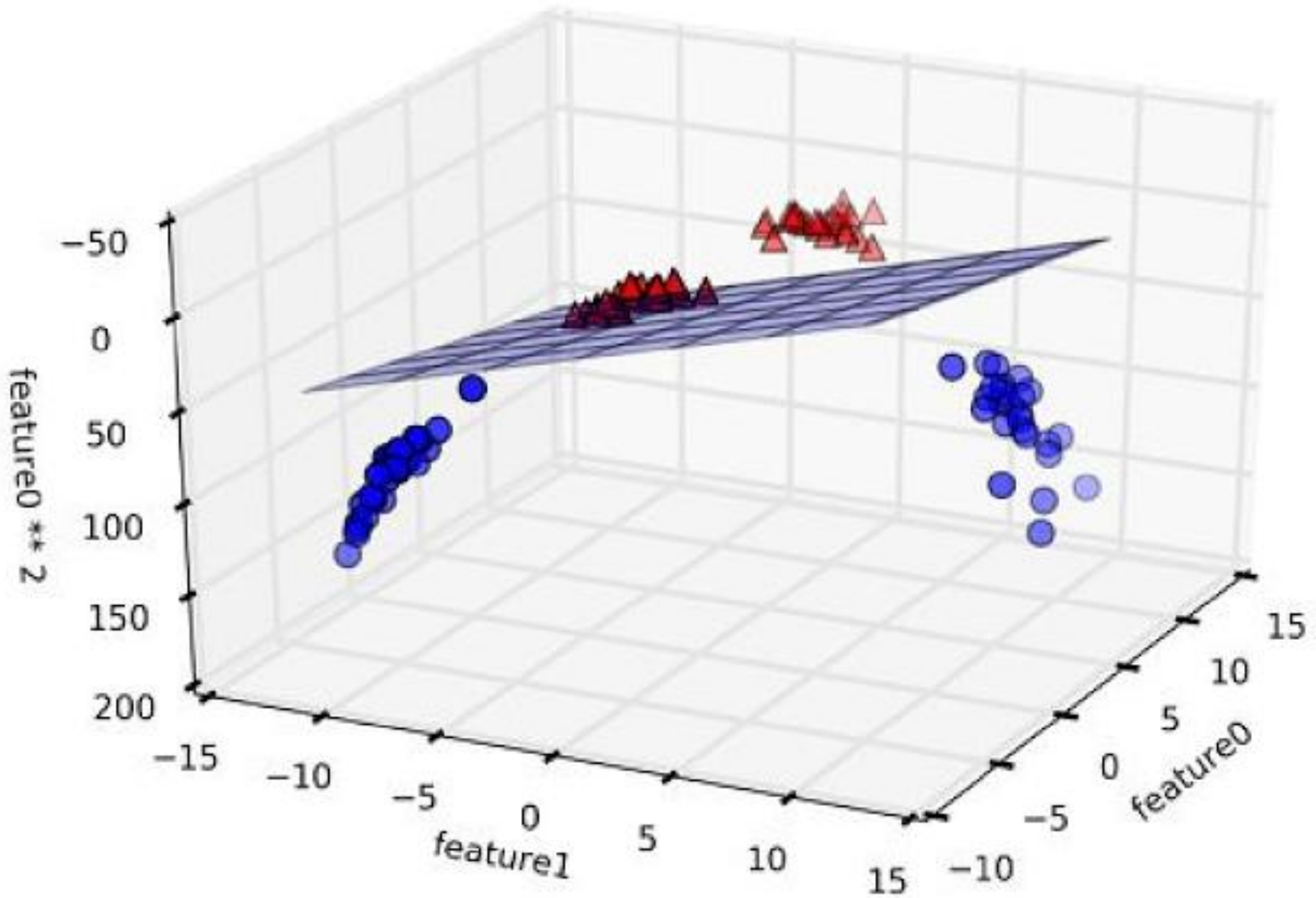


Figure 2-39. Decision boundary found by a linear SVM on the expanded three-dimensional dataset

As a function of the original features, the linear SVM model is not actually linear anymore. It is not a line, but more of an ellipse, as you can see from the plot created here (Figure 2-40):

In[78]:

```
ZZ = YY ** 2
dec = linear_svm_3d.decision_function(np.c_[XX.ravel(),
YY.ravel(), ZZ.ravel()])
plt.contourf(XX, YY, dec.reshape(XX.shape), levels=[dec.min(), 0,
dec.max()],
             cmap=mglearn.cm2, alpha=0.5)
mglearn.discrete_scatter(X[:, 0], X[:, 1], y)
plt.xlabel("Feature 0")
```

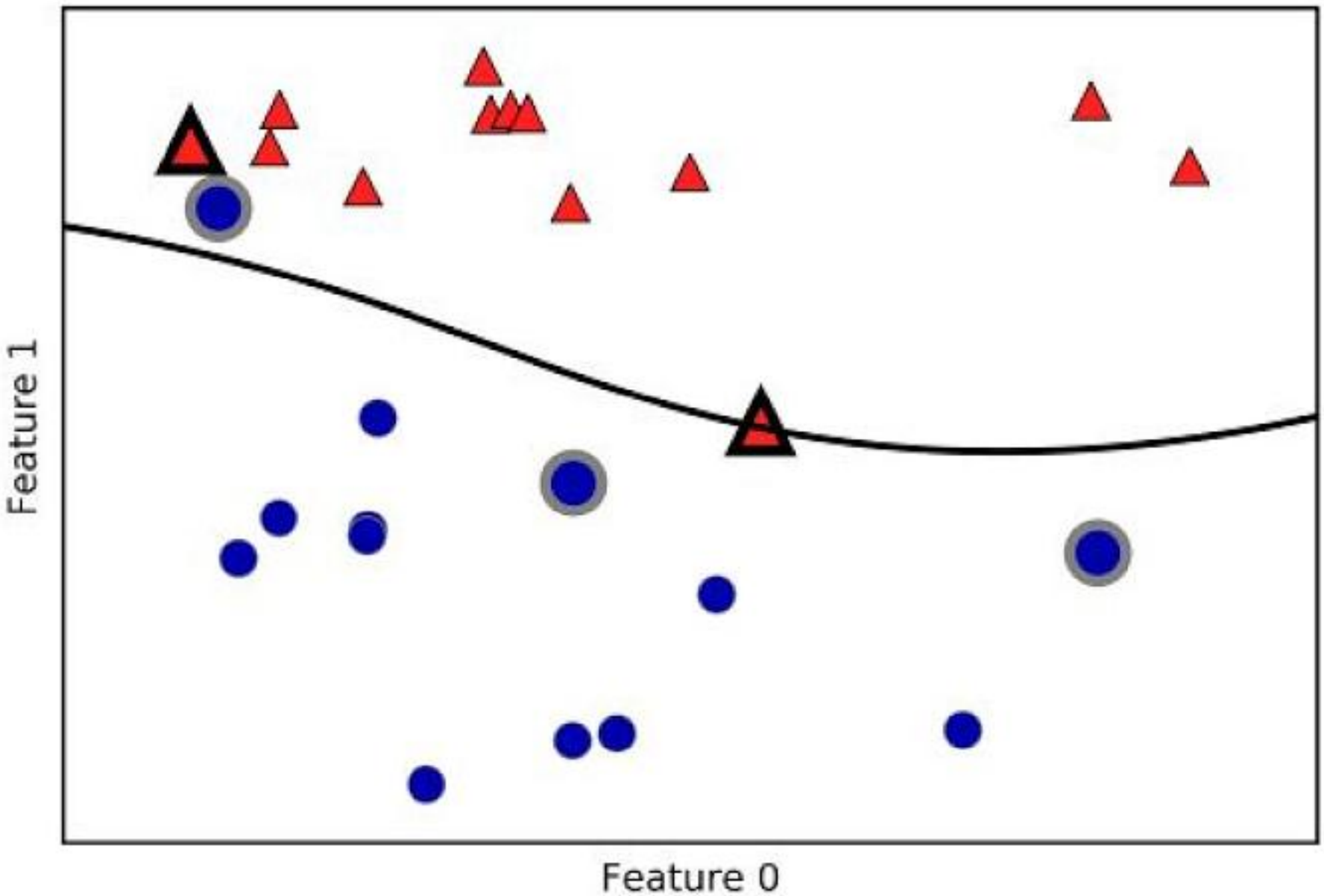


Figure 2-41. Decision boundary and support vectors found by an SVM with RBF kernel

In this case, the SVM yields a very smooth and nonlinear (not a straight line) boundary. We adjusted two parameters here: the `C` parameter and the `gamma` parameter, which we will now discuss in detail.

Tuning SVM parameters

The `gamma` parameter is the one shown in the formula given in the previous section, which corresponds to the inverse of the width of the Gaussian kernel. Intuitively, the gamma parameter determines how far the influence of a single training example reaches, with low values meaning corresponding to a far reach, and high values to a limited reach. In other words, the wider the radius of the Gaussian kernel, the further the influence

In[82]:

```
plt.boxplot(X_train, manage_xticks=False)
plt.yscale("symlog")
plt.xlabel("Feature index")
plt.ylabel("Feature magnitude")
```

From this plot we can determine that features in the Breast Cancer dataset are of completely different orders of magnitude. This can be somewhat of a problem for other models (like linear models), but it has devastating effects for the kernel SVM. Let's examine some ways to deal with this issue.

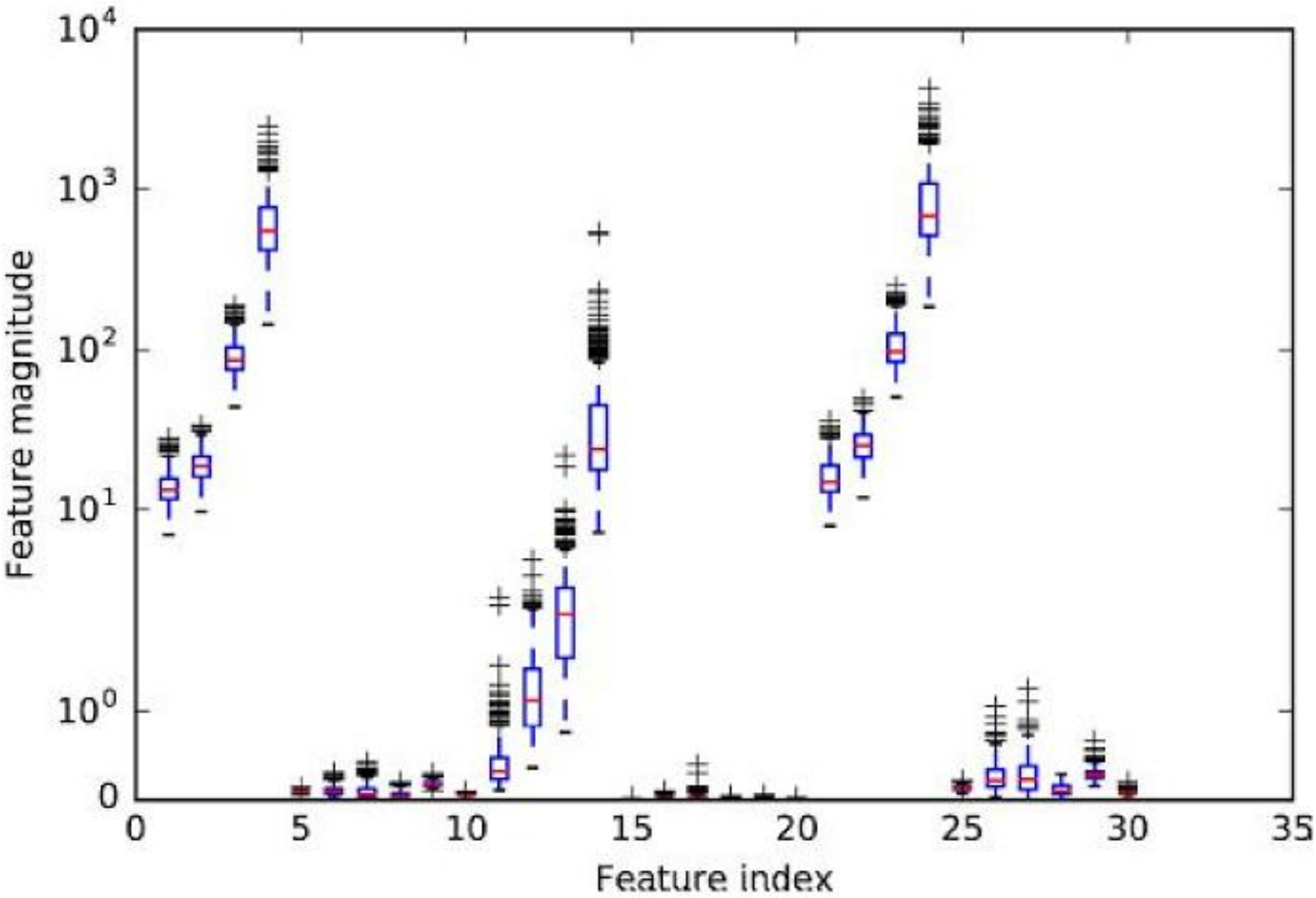


Figure 2-43. Feature ranges for the Breast Cancer dataset (note that the y axis has a logarithmic scale)

Preprocessing data for SVMs

One way to resolve this problem is by rescaling each feature so that they are all approximately on the same scale. A common rescaling method for kernel SVMs is to scale the data such that all features are between 0 and 1. We will see how to do this using the `MinMaxScaler` preprocessing method in Chapter 3, where we'll give more details. For now, let's do this "by hand":

In[83]:

```
min_on_training = X_train.min(axis=0)

range_on_training = (X_train - min_on_training).max(axis=0)

X_train_scaled = (X_train - min_on_training) / range_on_training
print("Minimum for each
feature\n{}".format(X_train_scaled.min(axis=0)))
print("Maximum for each feature\n
{}".format(X_train_scaled.max(axis=0)))
```

Out[83]:

```
Minimum for each feature
[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
0.  0.
  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
Maximum for each feature
 [ 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
1.  1.
   1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.]
```

In[84]:

```
X_test_scaled = (X_test - min_on_training) / range_on_training
```

In[85]:

```
svc = SVC()
svc.fit(X_train_scaled, y_train)

print("Accuracy on training set: {:.3f}".format(
    svc.score(X_train_scaled, y_train)))
print("Accuracy on test set:
{:.3f}".format(svc.score(X_test_scaled, y_test)))
```

Out[85]:

```
Accuracy on training set: 0.948
Accuracy on test set: 0.951
```

Scaling the data made a huge difference! Now we are actually in an underfitting regime, where training and test set performance are quite similar but less close to 100% accuracy. From here, we can try increasing either `C` or `gamma` to fit a more complex model. For example:

In[86]:

```
svc = SVC(C=1000)
svc.fit(X_train_scaled, y_train)

print("Accuracy on training set: {:.3f}".format(
    svc.score(X_train_scaled, y_train)))
```

```
print("Accuracy on test set:
{:.3f}".format(svc.score(X_test_scaled, y_test)))
```

Out[86]:

```
Accuracy on training set: 0.988
Accuracy on test set: 0.972
```

Here, increasing `C` allows us to improve the model significantly, resulting in 97.2% accuracy.

Strengths, weaknesses, and parameters

Kernelized support vector machines are powerful models and perform well on a variety of datasets. SVMs allow for complex decision boundaries, even if the data has only a few features. They work well on low-dimensional and high-dimensional data (i.e., few and many features), but don't scale very well with the number of samples. Running an SVM on data with up to 10,000 samples might work well, but working with datasets of size 100,000 or more can become challenging in terms of runtime and memory usage.

Another downside of SVMs is that they require careful preprocessing of the data and tuning of the parameters. This is why, these days, most people instead use tree-based models such as random forests or gradient boosting (which require little or no preprocessing) in many applications. Furthermore, SVM models are hard to inspect; it can be difficult to understand why a particular prediction was made, and it might be tricky to explain the model to a nonexpert.

Still, it might be worth trying SVMs, particularly if all of your features represent measurements in similar units (e.g., all are pixel intensities) and they are on similar scales.

The important parameters in kernel SVMs are the regularization parameter `C`, the choice of the kernel, and the kernel-specific parameters. Although we primarily focused on the RBF kernel, other choices are available in `scikit-learn`. The RBF kernel has only one parameter, `gamma`, which is the inverse of the width of the Gaussian kernel. `gamma` and `C` both control the complexity of the model, with large values in either resulting in a more complex model. Therefore, good settings for the two parameters are usually strongly correlated, and `C` and `gamma` should be adjusted together.

Neural Networks (Deep Learning)

A family of algorithms known as neural networks has recently seen a revival under the name "deep learning." While deep learning shows great promise in many machine learning applications, deep learning algorithms are often tailored very carefully to a specific use case. Here, we will only discuss some relatively simple methods, namely *multilayer perceptrons* for classification and regression, that can serve as a starting point for more involved deep learning methods. Multilayer perceptrons (MLPs) are also known as (vanilla) feed-forward neural networks, or sometimes just neural networks.

The neural network model

MLPs can be viewed as generalizations of linear models that perform multiple stages of processing to come to a decision.

Remember that the prediction by a linear regressor is given as:

$$= w[0] * x[0] + w[1] * x[1] + \ldots + w[p] * x[p] + b$$

In plain English, is a weighted sum of the input features $x[0]$ to $x[p]$, weighted by the learned coefficients $w[0]$ to $w[p]$. We could visualize this graphically as shown in Figure 2-44:

In[87]:

```
display(mglearn.plots.plot_logistic_regression_graph())
```

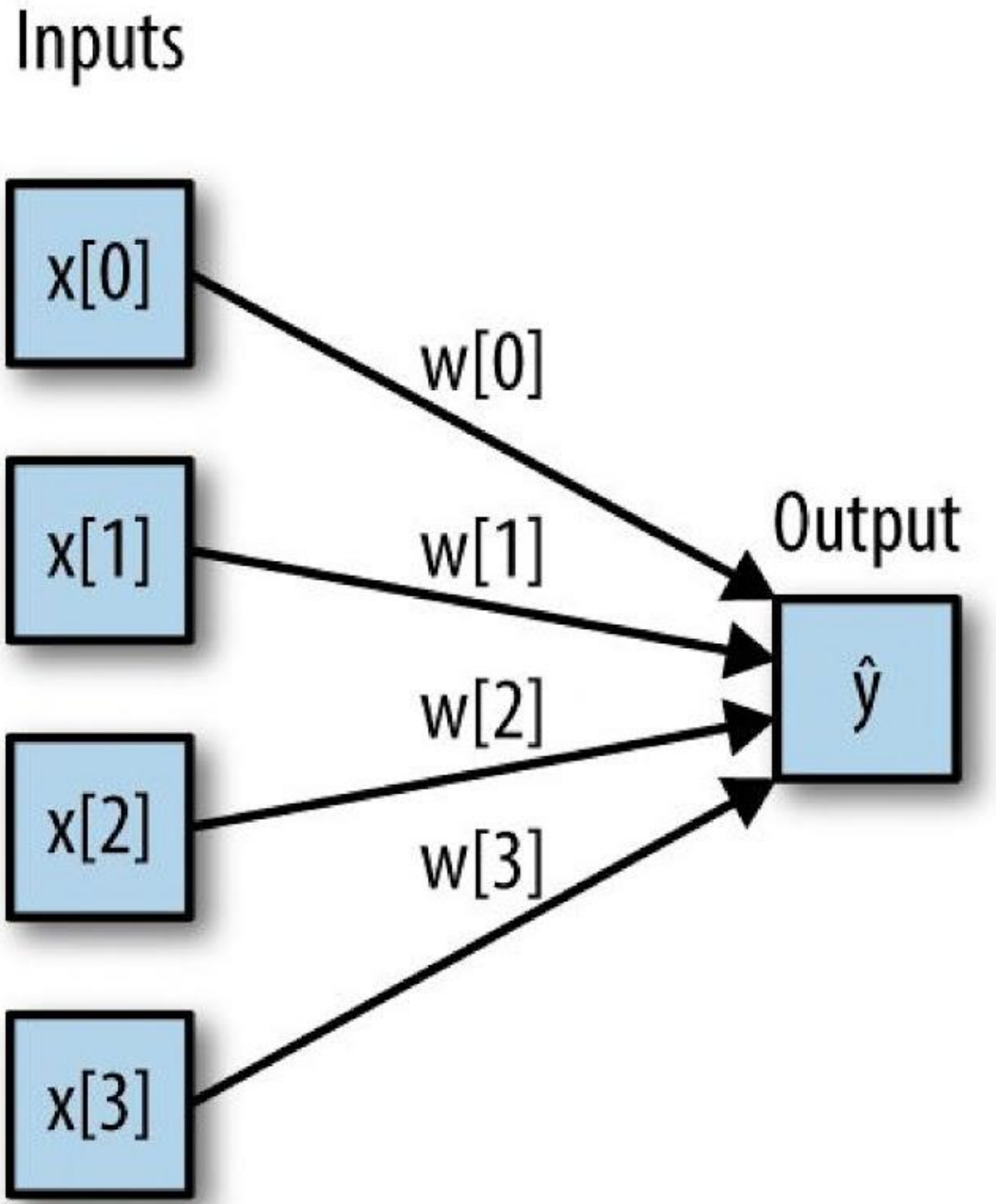


Figure 2-44. Visualization of logistic regression, where input features and predictions are shown as nodes, and the coefficients are connections between the nodes

Here, each node on the left represents an input feature, the connecting lines represent the learned coefficients, and the node on the right represents the output, which is a weighted sum of the inputs.

In an MLP this process of computing weighted sums is repeated multiple times, first computing *hidden units* that represent an intermediate processing step, which are again combined using weighted sums to yield the final result (Figure 2-45):

In[88]:

```
display(mglearn.plots.plot_single_hidden_layer_graph())
```

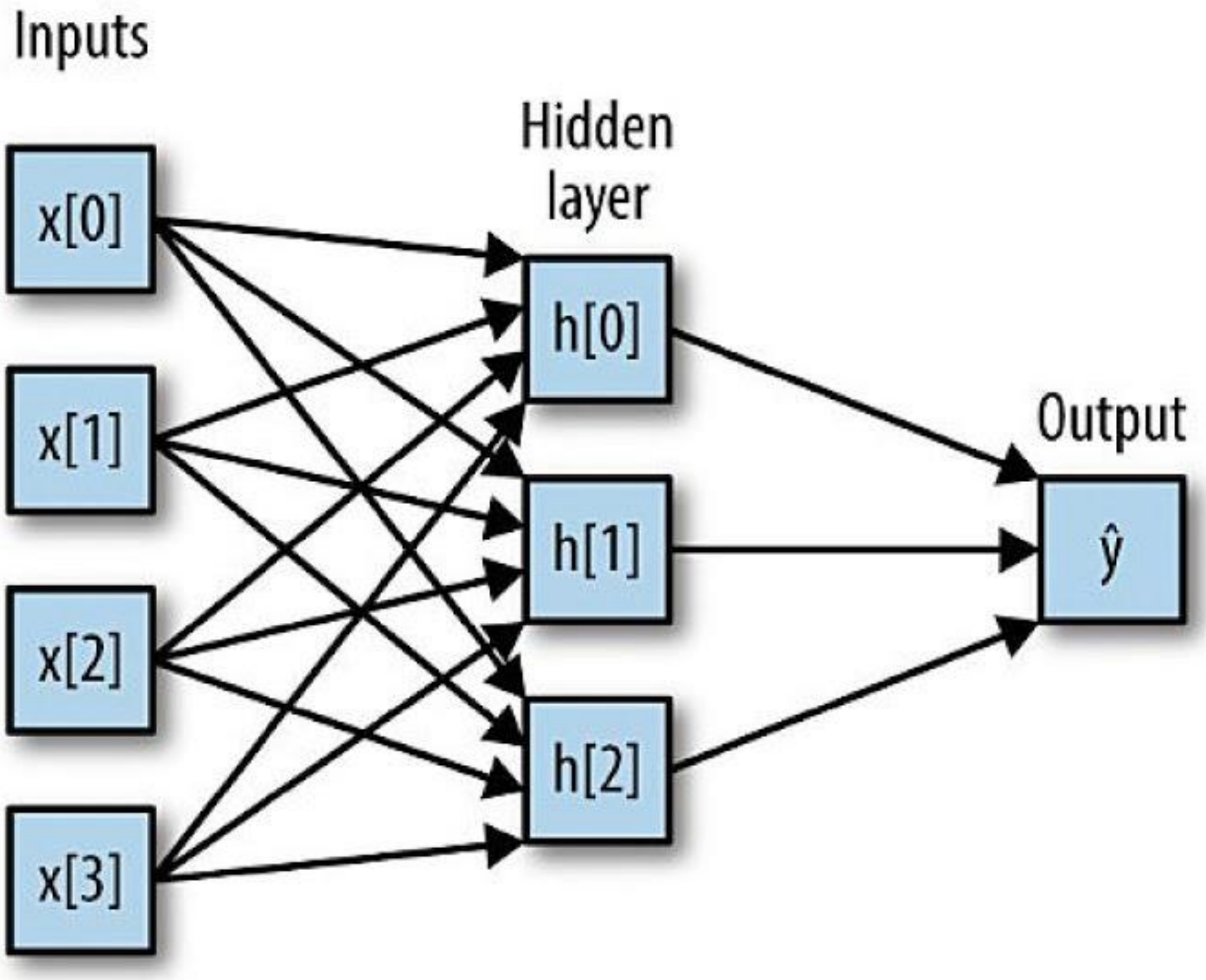


Figure 2-45. Illustration of a multilayer perceptron with a single hidden layer

This model has a lot more coefficients (also called weights) to learn: there is one between every input and every hidden unit (which make up the *hidden layer*), and one between every unit in the hidden layer and the output.

Computing a series of weighted sums is mathematically the same as computing just one weighted sum, so to make this model truly more powerful than a linear model, we need one extra trick. After computing a weighted sum for each hidden unit, a nonlinear function is applied to the result—usually the *rectifying nonlinearity* (also known as rectified linear

$x[3] + b[2])$

$= v[0] * h[0] + v[1] * h[1] + v[2] * h[2] + b$

Here, w are the weights between the input x and the hidden layer h, and v are the weights between the hidden layer h and the output . The weights v and w are learned from data, x are the input features, is the computed output, and h are intermediate computations. An important parameter that needs to be set by the user is the number of nodes in the hidden layer. This can be as small as 10 for very small or simple datasets and as big as 10,000 for very complex data. It is also possible to add additional hidden layers, as shown in Figure 2-47:

In[90]:

```
mglearn.plots.plot_two_hidden_layer_graph()
```

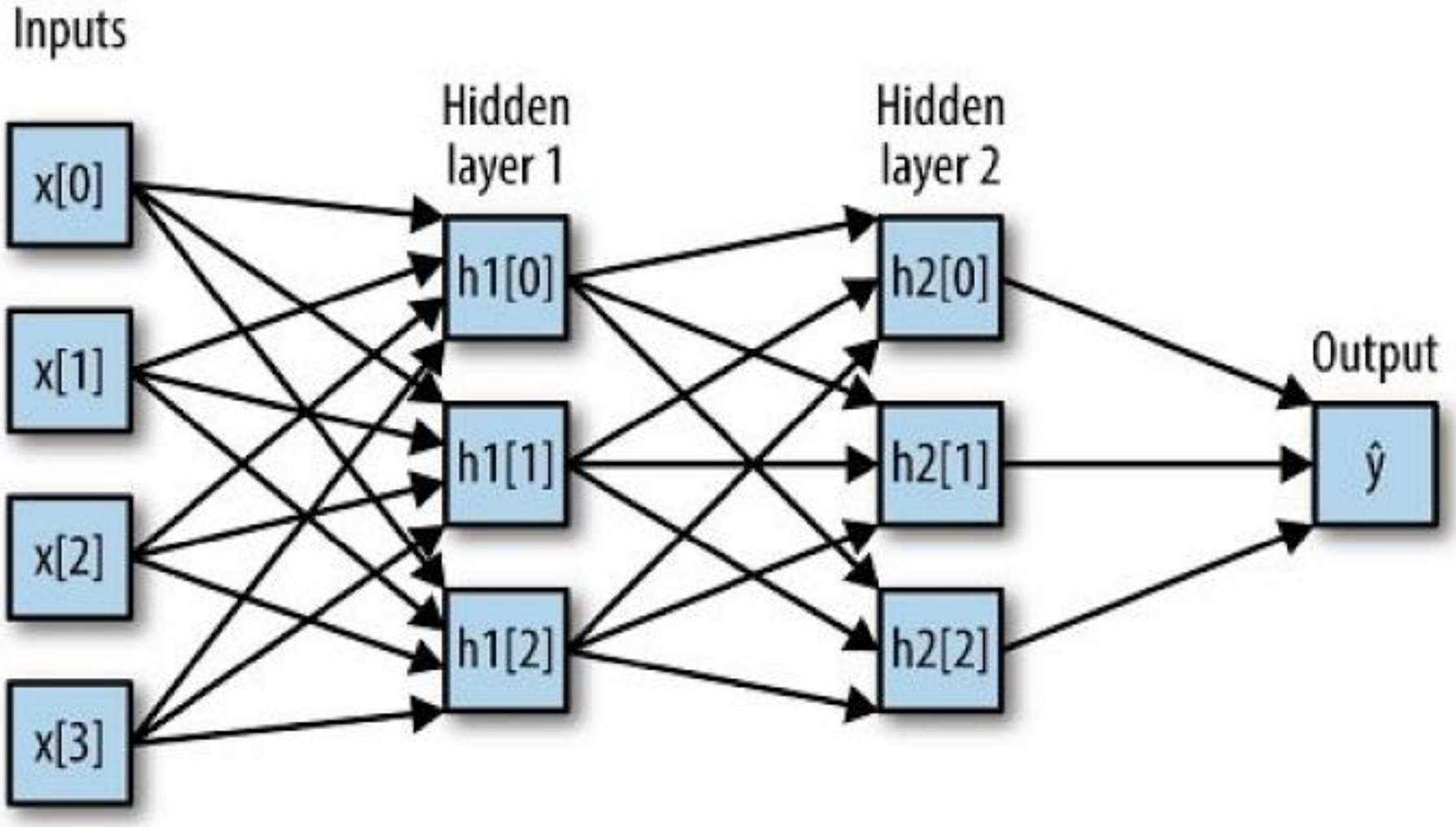


Figure 2-47. A multilayer perceptron with two hidden layers

Having large neural networks made up of many of these layers of computation is what inspired the term "deep learning."

Tuning neural networks

Let's look into the workings of the MLP by applying the `MLPClassifier` to the `two_moons` dataset we used earlier in this chapter. The results are shown in Figure 2-48:

In[91]:

```
from sklearn.neural_network import MLPClassifier
from sklearn.datasets import make_moons

X, y = make_moons(n_samples=100, noise=0.25, random_state=3)

X_train, X_test, y_train, y_test = train_test_split(X, y,
stratify=y,
```

small dataset. We can reduce the number (which reduces the complexity of the model) and still get a good result (Figure 2-49):

In[92]:

```
mlp = MLPClassifier(solver='lbfgs', random_state=0,
hidden_layer_sizes=[10])
mlp.fit(X_train, y_train)
mglearn.plots.plot_2d_separator(mlp, X_train, fill=True, alpha=.3)
mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1], y_train)
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
```

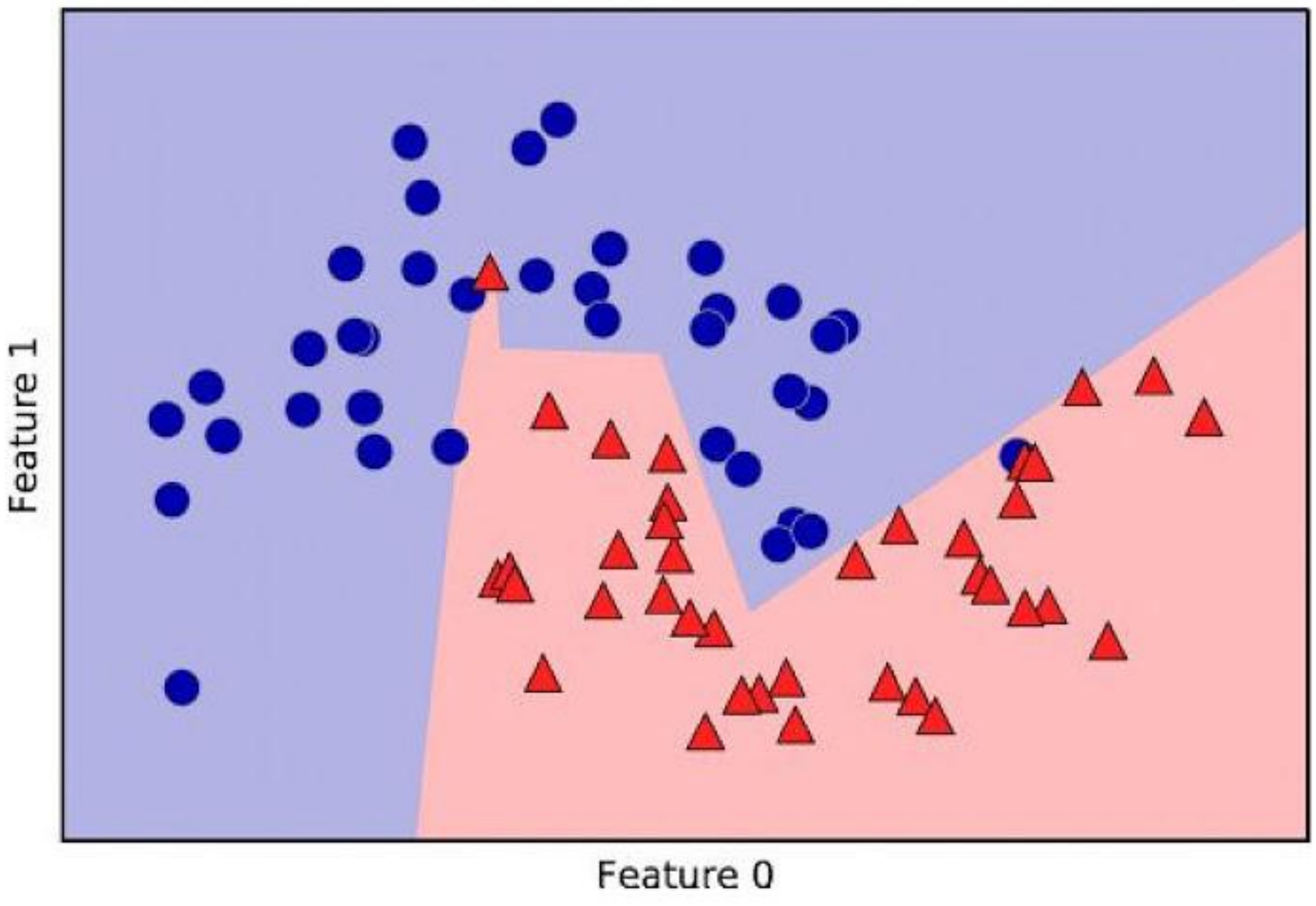


Figure 2-49. Decision boundary learned by a neural network with 10 hidden units on the two_moons dataset

With only 10 hidden units, the decision boundary looks somewhat more

ragged. The default nonlinearity is relu, shown in Figure 2-46. With a single hidden layer, this means the decision function will be made up of 10 straight line segments. If we want a smoother decision boundary, we could add more hidden units (as in Figure 2-48), add a second hidden layer (Figure 2-50), or use the tanh nonlinearity (Figure 2-51):

In[93]:

```
mlp = MLPClassifier(solver='lbfgs', random_state=0,
                    hidden_layer_sizes=[10, 10])
mlp.fit(X_train, y_train)
mglearn.plots.plot_2d_separator(mlp, X_train, fill=True, alpha=.3)
mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1], y_train)
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
```

In[94]:

```
mlp = MLPClassifier(solver='lbfgs', activation='tanh',
                    random_state=0, hidden_layer_sizes=[10, 10])
mlp.fit(X_train, y_train)
mglearn.plots.plot_2d_separator(mlp, X_train, fill=True, alpha=.3)
mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1], y_train)
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
```

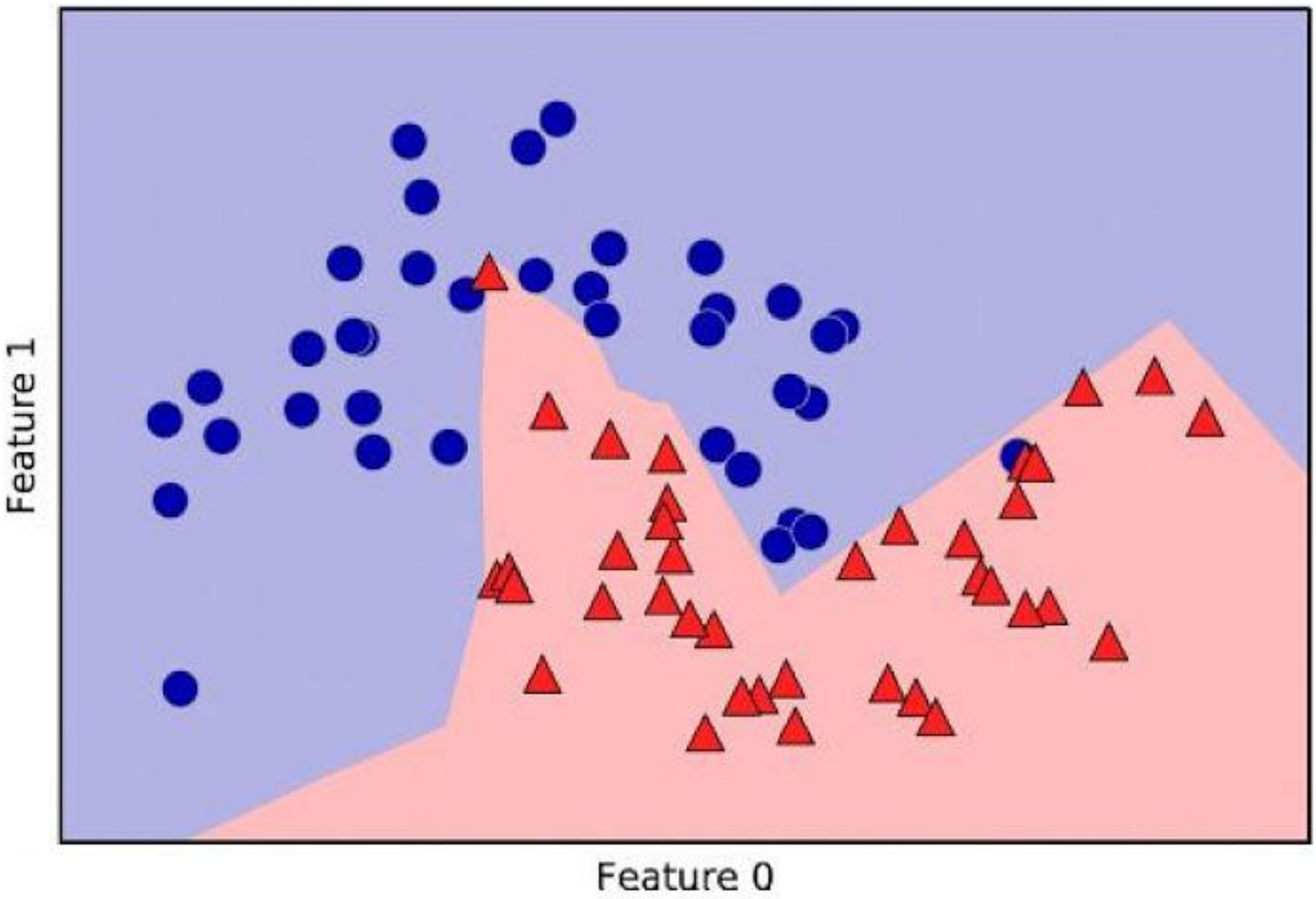


Figure 2-50. Decision boundary learned using 2 hidden layers with 10 hidden units each, with rect activation function

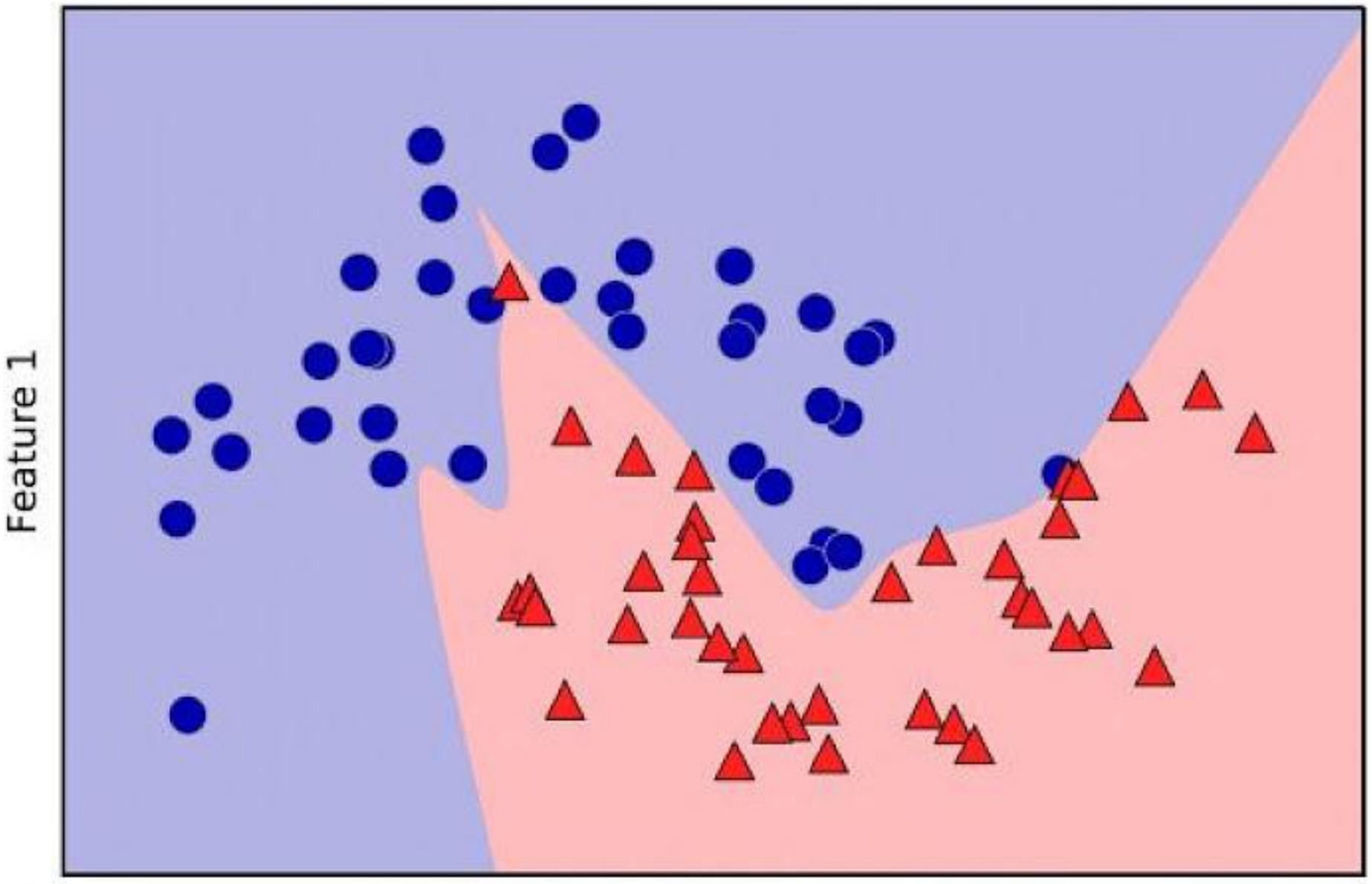


Figure 2-51. Decision boundary learned using 2 hidden layers with 10 hidden units each, with tanh activation function

Finally, we can also control the complexity of a neural network by using an `l2` penalty to shrink the weights toward zero, as we did in ridge regression and the linear classifiers. The parameter for this in the `MLPClassifier` is `alpha` (as in the linear regression models), and it's set to a very low value (little regularization) by default. Figure 2-52 shows the effect of different values of `alpha` on the `two_moons` dataset, using two hidden layers of 10 or 100 units each:

In[95]:

```
fig, axes = plt.subplots(2, 4, figsize=(20, 8))
for axx, n_hidden_nodes in zip(axes, [10, 100]):
```

```
    for ax, alpha in zip(axx, [0.0001, 0.01, 0.1, 1]):
        mlp = MLPClassifier(solver='lbfgs', random_state=0,
                            hidden_layer_sizes=[n_hidden_nodes,
n_hidden_nodes],
                            alpha=alpha)
        mlp.fit(X_train, y_train)
        mglearn.plots.plot_2d_separator(mlp, X_train, fill=True,
alpha=.3, ax=ax)
        mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1],
y_train, ax=ax)
        ax.set_title("n_hidden=[{}, {}]\nalpha={:.4f}".format(
                     n_hidden_nodes, n_hidden_nodes, alpha))
```

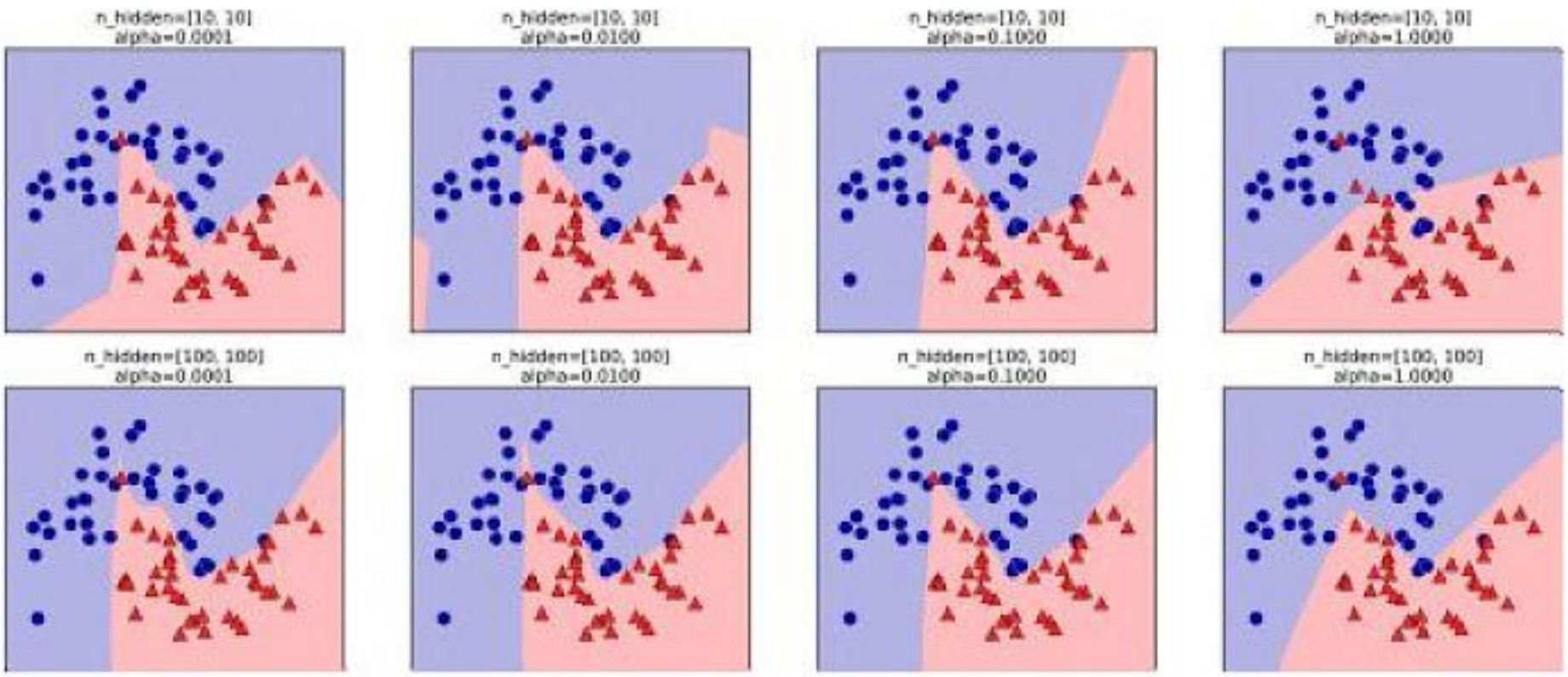


Figure 2-52. Decision functions for different numbers of hidden units and different settings of the alpha parameter

As you probably have realized by now, there are many ways to control the complexity of a neural network: the number of hidden layers, the number of units in each hidden layer, and the regularization (`alpha`). There are actually even more, which we won't go into here.

An important property of neural networks is that their weights are set randomly before learning is started, and this random initialization affects the model that is learned. That means that even when using exactly the

same parameters, we can obtain very different models when using different random seeds. If the networks are large, and their complexity is chosen properly, this should not affect accuracy too much, but it is worth keeping in mind (particularly for smaller networks). Figure 2-53 shows plots of several models, all learned with the same settings of the parameters:

In[96]:

```
fig, axes = plt.subplots(2, 4, figsize=(20, 8))
for i, ax in enumerate(axes.ravel()):
    mlp = MLPClassifier(solver='lbfgs', random_state=i,
                        hidden_layer_sizes=[100, 100])
    mlp.fit(X_train, y_train)
    mglearn.plots.plot_2d_separator(mlp, X_train, fill=True,
alpha=.3, ax=ax)
    mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1],
y_train, ax=ax)
```

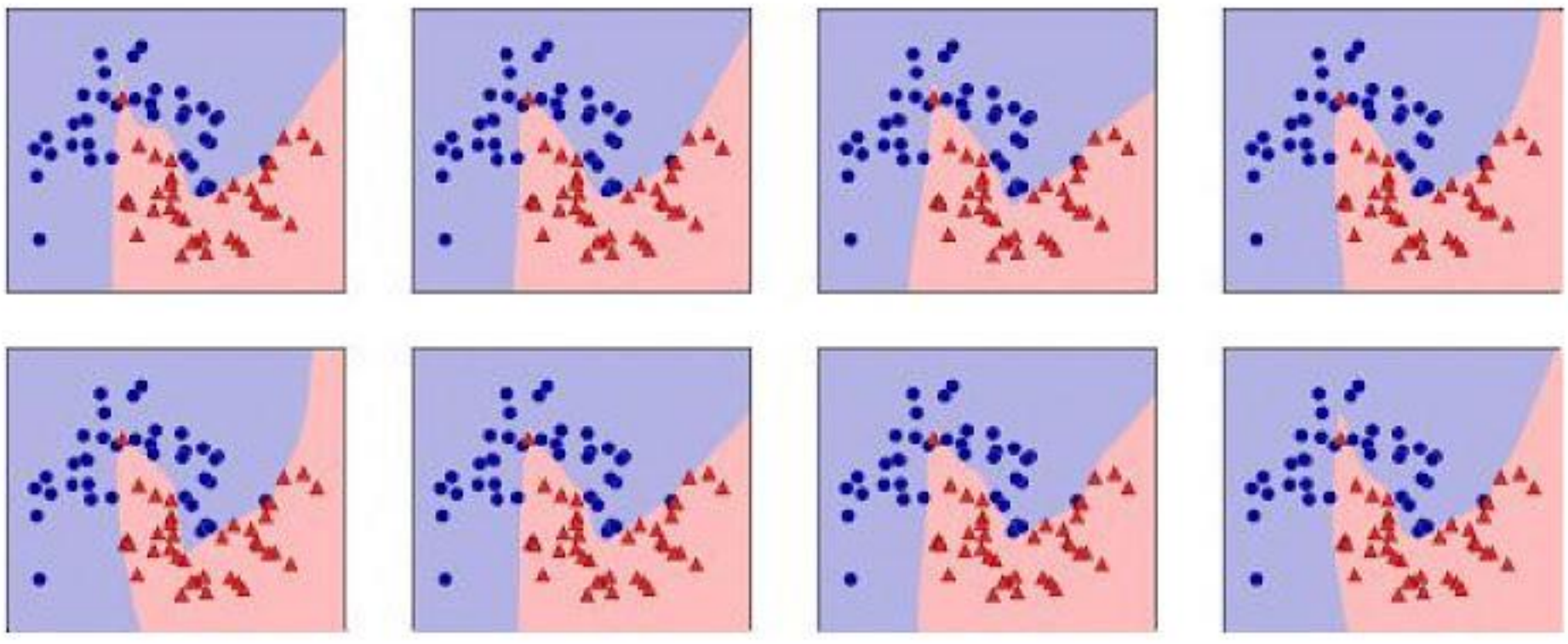


Figure 2-53. Decision functions learned with the same parameters but different random initializations

To get a better understanding of neural networks on real-world data, let's apply the `MLPClassifier` to the Breast Cancer dataset. We start with the

default parameters:

In[97]:

```
print("Cancer data per-feature
maxima:\n{}".format(cancer.data.max(axis=0)))
```

Out[97]:

```
Cancer data per-feature maxima:
[   28.110    39.280   188.500  2501.000     0.163     0.345
0.427
     0.201     0.304     0.097     2.873     4.885    21.980
542.200
     0.031     0.135     0.396     0.053     0.079     0.030
36.040
    49.540   251.200  4254.000     0.223     1.058     1.252
0.291
     0.664     0.207]
```

In[98]:

```
X_train, X_test, y_train, y_test = train_test_split(
    cancer.data, cancer.target, random_state=0)

mlp = MLPClassifier(random_state=42)
mlp.fit(X_train, y_train)

print("Accuracy on training set: {:.2f}".format(mlp.score(X_train,
y_train)))
print("Accuracy on test set: {:.2f}".format(mlp.score(X_test,
y_test)))
```

Out[98]:

```
Accuracy on training set: 0.92
Accuracy on test set: 0.90
```

The accuracy of the MLP is quite good, but not as good as the other models. As in the earlier SVC example, this is likely due to scaling of the data. Neural networks also expect all input features to vary in a similar way, and ideally to have a mean of 0, and a variance of 1. We must rescale our data so that it fulfills these requirements. Again, we will do this by hand here, but we'll introduce the `StandardScaler` to do this automatically in Chapter 3:

In[99]:

```
mean_on_train = X_train.mean(axis=0)

std_on_train = X_train.std(axis=0)


X_train_scaled = (X_train - mean_on_train) / std_on_train


X_test_scaled = (X_test - mean_on_train) / std_on_train

mlp = MLPClassifier(random_state=0)
mlp.fit(X_train_scaled, y_train)

print("Accuracy on training set: {:.3f}".format(
    mlp.score(X_train_scaled, y_train)))
print("Accuracy on test set:
{:.3f}".format(mlp.score(X_test_scaled, y_test)))
```

Out[99]:

```
Accuracy on training set: 0.991
Accuracy on test set: 0.965

ConvergenceWarning:
    Stochastic Optimizer: Maximum iterations reached and the
optimization
    hasn't converged yet.
```

The results are much better after scaling, and already quite competitive. We got a warning from the model, though, that tells us that the maximum number of iterations has been reached. This is part of the `adam` algorithm for learning the model, and tells us that we should increase the number of iterations:

In[100]:

```
mlp = MLPClassifier(max_iter=1000, random_state=0)
mlp.fit(X_train_scaled, y_train)

print("Accuracy on training set: {:.3f}".format(
    mlp.score(X_train_scaled, y_train)))
print("Accuracy on test set:
{:.3f}".format(mlp.score(X_test_scaled, y_test)))
```

Out[100]:

```
Accuracy on training set: 0.995
Accuracy on test set: 0.965
```

Increasing the number of iterations only increased the training set performance, not the generalization performance. Still, the model is performing quite well. As there is some gap between the training and the test performance, we might try to decrease the model's complexity to get better generalization performance. Here, we choose to increase the `alpha`

parameter (quite aggressively, from `0.0001` to 1) to add stronger regularization of the weights:

In[101]:

```
mlp = MLPClassifier(max_iter=1000, alpha=1, random_state=0)
mlp.fit(X_train_scaled, y_train)

print("Accuracy on training set: {:.3f}".format(
    mlp.score(X_train_scaled, y_train)))
print("Accuracy on test set:
{:.3f}".format(mlp.score(X_test_scaled, y_test)))
```

Out[101]:

```
Accuracy on training set: 0.988
Accuracy on test set: 0.972
```

This leads to a performance on par with the best models so far.[12]

While it is possible to analyze what a neural network has learned, this is usually much trickier than analyzing a linear model or a tree-based model. One way to introspect what was learned is to look at the weights in the model. You can see an example of this in the `scikit-learn` example gallery. For the Breast Cancer dataset, this might be a bit hard to understand. The following plot (Figure 2-54) shows the weights that were learned connecting the input to the first hidden layer. The rows in this plot correspond to the 30 input features, while the columns correspond to the 100 hidden units. Light colors represent large positive values, while dark colors represent negative values:

In[102]:

```
plt.figure(figsize=(20, 5))
plt.imshow(mlp.coefs_[0], interpolation='none', cmap='viridis')
plt.yticks(range(30), cancer.feature_names)
plt.xlabel("Columns in weight matrix")
plt.ylabel("Input feature")
plt.colorbar()
```

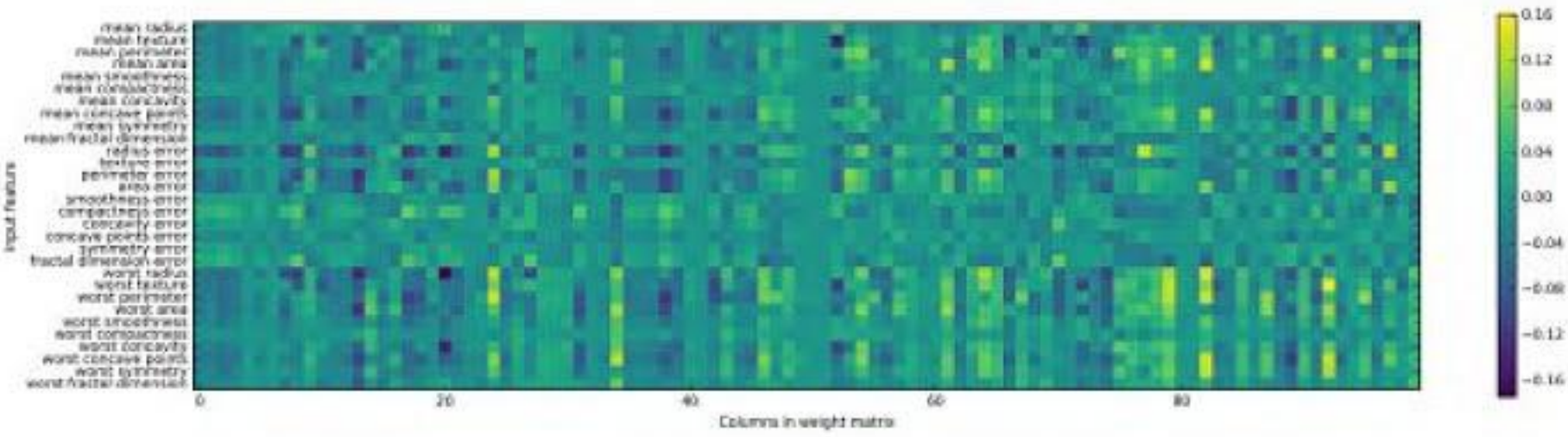


Figure 2-54. Heat map of the first layer weights in a neural network learned on the Breast Cancer dataset

One possible inference we can make is that features that have very small weights for all of the hidden units are "less important" to the model. We can see that "mean smoothness" and "mean compactness," in addition to the features found between "smoothness error" and "fractal dimension error," have relatively low weights compared to other features. This could mean that these are less important features or possibly that we didn't represent them in a way that the neural network could use.

We could also visualize the weights connecting the hidden layer to the output layer, but those are even harder to interpret.

While the `MLPClassifier` and `MLPRegressor` provide easy-to-use interfaces for the most common neural network architectures, they only capture a small subset of what is possible with neural networks. If you are interested in working with more flexible or larger models, we encourage you to look beyond `scikit-learn` into the fantastic deep learning libraries

that are out there. For Python users, the most well-established are `keras`, `lasagna`, and `tensor-flow`. `lasagna` builds on the `theano` library, while `keras` can use either `tensor-flow` or `theano`. These libraries provide a much more flexible interface to build neural networks and track the rapid progress in deep learning research. All of the popular deep learning libraries also allow the use of high-performance graphics processing units (GPUs), which `scikit-learn` does not support. Using GPUs allows us to accelerate computations by factors of 10x to 100x, and they are essential for applying deep learning methods to large-scale datasets.

Strengths, weaknesses, and parameters

Neural networks have reemerged as state-of-the-art models in many applications of machine learning. One of their main advantages is that they are able to capture information contained in large amounts of data and build incredibly complex models. Given enough computation time, data, and careful tuning of the parameters, neural networks often beat other machine learning algorithms (for classification and regression tasks).

This brings us to the downsides. Neural networks—particularly the large and powerful ones—often take a long time to train. They also require careful preprocessing of the data, as we saw here. Similarly to SVMs, they work best with "homogeneous" data, where all the features have similar meanings. For data that has very different kinds of features, tree-based models might work better. Tuning neural network parameters is also an art unto itself. In our experiments, we barely scratched the surface of possible ways to adjust neural network models and how to train them.

Estimating complexity in neural networks

The most important parameters are the number of layers and the number of hidden units per layer. You should start with one or two hidden layers, and possibly expand from there. The number of nodes per hidden layer is often similar to the number of input features, but rarely higher than in the low to

mid-thousands.

A helpful measure when thinking about the model complexity of a neural network is the number of weights or coefficients that are learned. If you have a binary classification dataset with 100 features, and you have 100 hidden units, then there are 100 * 100 = 10,000 weights between the input and the first hidden layer. There are also 100 * 1 = 100 weights between the hidden layer and the output layer, for a total of around 10,100 weights. If you add a second hidden layer with 100 hidden units, there will be another 100 * 100 = 10,000 weights from the first hidden layer to the second hidden layer, resulting in a total of 20,100 weights. If instead you use one layer with 1,000 hidden units, you are learning 100 * 1,000 = 100,000 weights from the input to the hidden layer and 1,000 * 1 weights from the hidden layer to the output layer, for a total of 101,000. If you add a second hidden layer you add 1,000 * 1,000 = 1,000,000 weights, for a whopping total of 1,101,000—50 times larger than the model with two hidden layers of size 100.

A common way to adjust parameters in a neural network is to first create a network that is large enough to overfit, making sure that the task can actually be learned by the network. Then, once you know the training data can be learned, either shrink the network or increase `alpha` to add regularization, which will improve generalization performance.

In our experiments, we focused mostly on the definition of the model: the number of layers and nodes per layer, the regularization, and the nonlinearity. These define the model we want to learn. There is also the question of *how* to learn the model, or the algorithm that is used for learning the parameters, which is set using the `algorithm` parameter. There are two easy-to-use choices for `algorithm`. The default is `'adam'`, which works well in most situations but is quite sensitive to the scaling of the data

`predict_proba`. Most (but not all) classifiers have at least one of them, and many classifiers have both. Let's look at what these two functions do on a synthetic two-dimensional dataset, when building a `GradientBoostingClassifier` classifier, which has both a `decision_function` and a `predict_proba` method:

In[103]:

```
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.datasets import make_circles
X, y = make_circles(noise=0.25, factor=0.5, random_state=1)


y_named = np.array(["blue", "red"])[y]


X_train, X_test, y_train_named, y_test_named, y_train, y_test = \
    train_test_split(X, y_named, y, random_state=0)


gbrt = GradientBoostingClassifier(random_state=0)
gbrt.fit(X_train, y_train_named)
```

The Decision Function

In the binary classification case, the return value of `decision_function` is of shape `(n_samples,)`, and it returns one floating-point number for each sample:

In[104]:

```
print("X_test.shape: {}".format(X_test.shape))
print("Decision function shape: {}".format(
    gbrt.decision_function(X_test).shape))
```

Out[104]:

```
X_test.shape: (25, 2)
Decision function shape: (25,)
```

This value encodes how strongly the model believes a data point to belong to the “positive” class, in this case class 1. Positive values indicate a preference for the positive class, and negative values indicate a preference for the “negative” (other) class:

In[105]:

```
print("Decision
function:\n{}".format(gbrt.decision_function(X_test)[:6]))
```

Out[105]:

```
Decision function:
[ 4.136 -1.683 -3.951 -3.626  4.29   3.662]
```

We can recover the prediction by looking only at the sign of the decision function:

In[106]:

```
print("Thresholded decision function:\n{}".format(
    gbrt.decision_function(X_test) > 0))
print("Predictions:\n{}".format(gbrt.predict(X_test)))
```

Out[106]:

In[108]:

```
decision_function = gbrt.decision_function(X_test)
print("Decision function minimum: {:.2f} maximum: {:.2f}".format(
    np.min(decision_function), np.max(decision_function)))
```

Out[108]:

```
Decision function minimum: -7.69 maximum: 4.29
```

This arbitrary scaling makes the output of `decision_function` often hard to interpret.

In the following example we plot the `decision_function` for all points in the 2D plane using a color coding, next to a visualization of the decision boundary, as we saw earlier. We show training points as circles and test data as triangles (Figure 2-55):

In[109]:

```
fig, axes = plt.subplots(1, 2, figsize=(13, 5))
mglearn.tools.plot_2d_separator(gbrt, X, ax=axes[0], alpha=.4,
                                fill=True, cm=mglearn.cm2)
scores_image = mglearn.tools.plot_2d_scores(gbrt, X, ax=axes[1],
                                            alpha=.4,
cm=mglearn.ReBl)

for ax in axes:

    mglearn.discrete_scatter(X_test[:, 0], X_test[:, 1], y_test,
                             markers='^', ax=ax)
    mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1],
y_train,
                             markers='o', ax=ax)
    ax.set_xlabel("Feature 0")
```

```
    ax.set_ylabel("Feature 1")
cbar = plt.colorbar(scores_image, ax=axes.tolist())
axes[0].legend(["Test class 0", "Test class 1", "Train class 0",
                "Train class 1"], ncol=4, loc=(.1, 1.1))
```

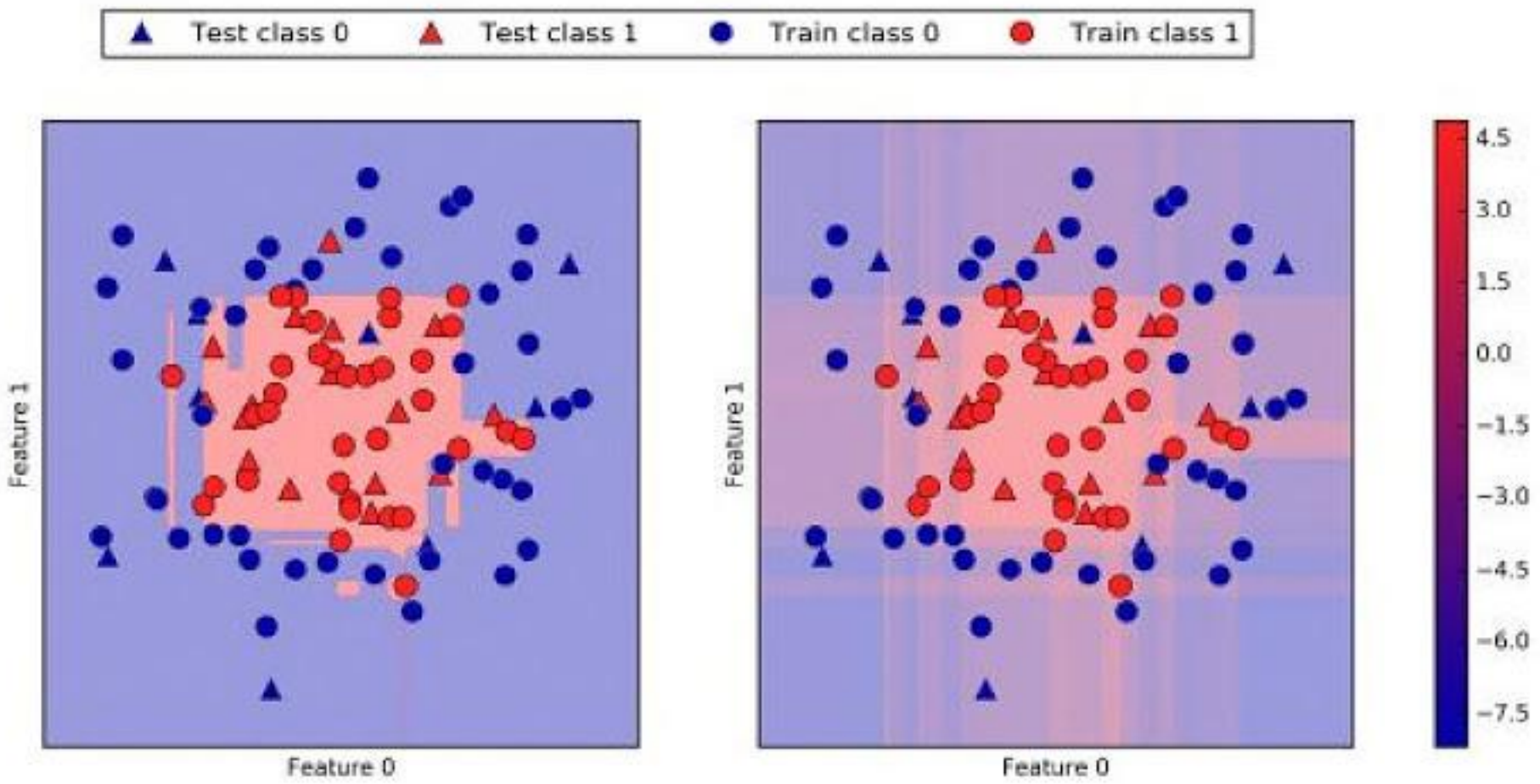


Figure 2-55. Decision boundary (left) and decision function (right) for a gradient boosting model on a two-dimensional toy dataset

Encoding not only the predicted outcome but also how certain the classifier is provides additional information. However, in this visualization, it is hard to make out the boundary between the two classes.

Predicting Probabilities

The output of `predict_proba` is a probability for each class, and is often more easily understood than the output of `decision_function`. It is always of shape `(n_samples, 2)` for binary classification:

In[110]:

```
print("Shape of probabilities:
```

```
{}".format(gbrt.predict_proba(X_test).shape))
```

Out[110]:

```
Shape of probabilities: (25, 2)
```

The first entry in each row is the estimated probability of the first class, and the second entry is the estimated probability of the second class. Because it is a probability, the output of `predict_proba` is always between 0 and 1, and the sum of the entries for both classes is always 1:

In[111]:

```
print("Predicted probabilities:\n{}".format(
    gbrt.predict_proba(X_test[:6])))
```

Out[111]:

```
Predicted probabilities:
[[ 0.016  0.984]
 [ 0.843  0.157]
 [ 0.981  0.019]
 [ 0.974  0.026]
 [ 0.014  0.986]
 [ 0.025  0.975]]
```

Because the probabilities for the two classes sum to 1, exactly one of the classes will be above 50% certainty. That class is the one that is predicted.[13]

You can see in the previous output that the classifier is relatively certain for

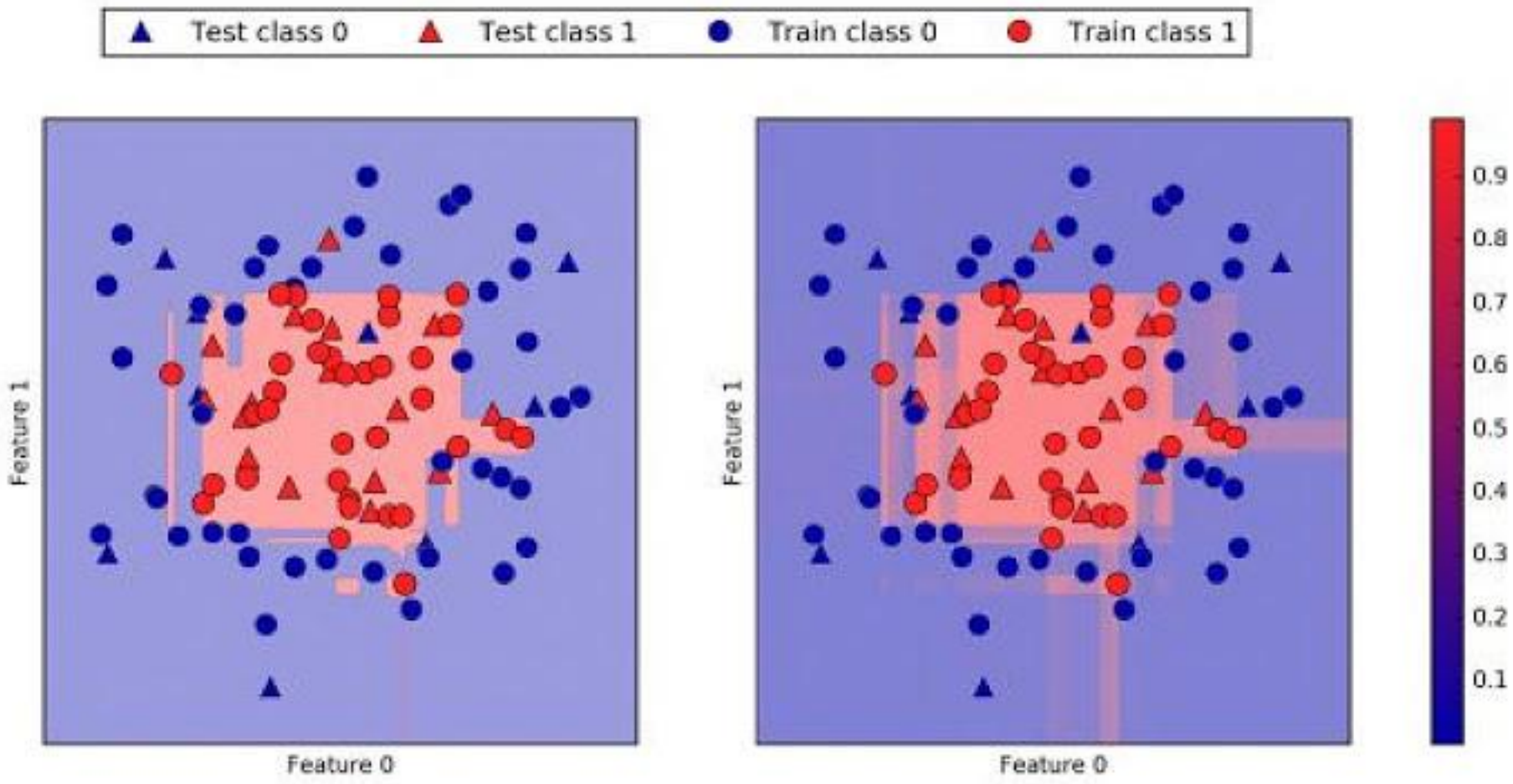


Figure 2-56. Decision boundary (left) and predicted probabilities for the gradient boosting model shown in Figure 2-55

The boundaries in this plot are much more well-defined, and the small areas of uncertainty are clearly visible.

The `scikit-learn` website has a great comparison of many models and what their uncertainty estimates look like. We've reproduced this in Figure 2-57, and we encourage you to go though the example there.

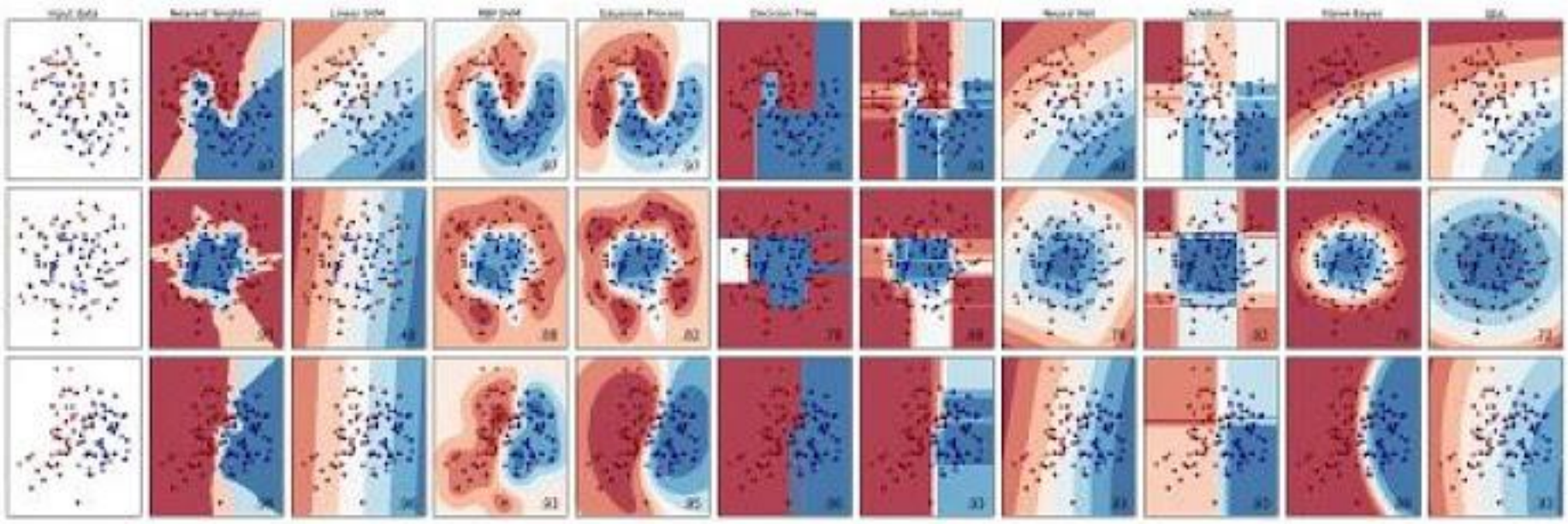


Figure 2-57. Comparison of several classifiers in scikit-learn on synthetic datasets (image courtesy http://scikit-learn.org)

Uncertainty in Multiclass Classification

So far, we've only talked about uncertainty estimates in binary classification. But the `decision_function` and `predict_proba` methods also work in the multiclass setting. Let's apply them on the Iris dataset, which is a three-class classification dataset:

In[113]:

```
from sklearn.datasets import load_iris

iris = load_iris()
X_train, X_test, y_train, y_test = train_test_split(
    iris.data, iris.target, random_state=42)

gbrt = GradientBoostingClassifier(learning_rate=0.01,
random_state=0)
gbrt.fit(X_train, y_train)
```

In[114]:

```
print("Decision function shape:
{}".format(gbrt.decision_function(X_test).shape))
```

```
print("Decision
function:\n{}".format(gbrt.decision_function(X_test)[:6, :]))
```

Out[114]:

```
Decision function shape: (38, 3)
Decision function:
[[-0.529  1.466 -0.504]
 [ 1.512 -0.496 -0.503]
 [-0.524 -0.468  1.52 ]
 [-0.529  1.466 -0.504]
 [-0.531  1.282  0.215]
 [ 1.512 -0.496 -0.503]]
```

In the multiclass case, the `decision_function` has the shape `(n_samples, n_classes)` and each column provides a "certainty score" for each class, where a large score means that a class is more likely and a small score means the class is less likely. You can recover the predictions from these scores by finding the maximum entry for each data point:

In[115]:

```
print("Argmax of decision function:\n{}".format(
      np.argmax(gbrt.decision_function(X_test), axis=1)))
print("Predictions:\n{}".format(gbrt.predict(X_test)))
```

Out[115]:

```
Argmax of decision function:
[1 0 2 1 1 0 1 2 1 1 2 0 0 0 0 1 2 1 1 2 0 2 0 2 2 2 2 2 0 0 0 0 1
0 0 2 1 0]
Predictions:
[1 0 2 1 1 0 1 2 1 1 2 0 0 0 0 1 2 1 1 2 0 2 0 2 2 2 2 2 0 0 0 0 1
```

assumptions the model makes and the meanings of the parameter settings will rarely lead to an accurate model.

This chapter contains a lot of information about the algorithms, and it is not necessary for you to remember all of these details for the following chapters. However, some knowledge of the models described here—and which to use in a specific situation—is important for successfully applying machine learning in practice. Here is a quick summary of when to use each model:

Nearest neighbors

: For small datasets, good as a baseline, easy to explain.

Linear models

: Go-to as a first algorithm to try, good for very large datasets, good for very high-dimensional data.

Naive Bayes

: Only for classification. Even faster than linear models, good for very large datasets and high-dimensional data. Often less accurate than linear models.

Decision trees

: Very fast, don't need scaling of the data, can be visualized and easily explained.

Random forests

: Nearly always perform better than a single decision tree, very robust and powerful. Don't need scaling of data. Not good for very high-dimensional sparse data.

Gradient boosted decision trees

: Often slightly more accurate than random forests. Slower to train but faster to predict than random forests, and smaller in memory. Need more parameter tuning than random forests.

Support vector machines

: Powerful for medium-sized datasets of features with similar meaning. Require scaling of data, sensitive to parameters.

Neural networks

: Can build very complex models, particularly for large datasets. Sensitive to scaling of the data and to the choice of parameters. Large models need a long time to train.

When working with a new dataset, it is in general a good idea to start with a simple model, such as a linear model or a naive Bayes or nearest neighbors classifier, and see how far you can get. After understanding more about the data, you can consider moving to an algorithm that can build more complex models, such as random forests, gradient boosted decision trees, SVMs, or neural networks.

You should now be in a position where you have some idea of how to apply, tune, and analyze the models we discussed here. In this chapter, we focused on the binary classification case, as this is usually easiest to understand. Most of the algorithms presented have classification and regression variants, however, and all of the classification algorithms support both binary and multiclass classification. Try applying any of these algorithms to the built-in datasets in `scikit-learn`, like the `boston_housing` or `diabetes` datasets for regression, or the `digits` dataset for multiclass classification. Playing around with the algorithms on

different datasets will give you a better feel for how long they need to train, how easy it is to analyze the models, and how sensitive they are to the representation of the data.

While we analyzed the consequences of different parameter settings for the algorithms we investigated, building a model that actually generalizes well to new data in production is a bit trickier than that. We will see how to properly adjust parameters and how to find good parameters automatically in Chapter 6.

First, though, we will dive in more detail into unsupervised learning and preprocessing in the next chapter.

[1] We ask linguists to excuse the simplified presentation of languages as distinct and fixed entities.

[2] In the real world, this is actually a tricky problem. While we know that the other customers haven't bought a boat from us yet, they might have bought one from someone else, or they may still be saving and plan to buy one in the future.

[3] And also provably, with the right math.

[4] Discussing all of them is beyond the scope of the book, and we refer you to the `scikit-learn` documentation for more details.

[5] This is 13 interactions for the first feature, plus 12 for the second not involving the first, plus 11 for the third and so on ($13 + 12 + 11 + \ldots + 1 = 91$).

[6] This is easy to see if you know some linear algebra.

[7] Mathematically, `Ridge` penalizes the squared L2 norm of the coefficients, or the Euclidean length of w.

[8] The lasso penalizes the L1 norm of the coefficient vector—or in other words, the sum of the absolute values of the coefficients.

[9] It is actually possible to make very good forecasts with tree-based models (for example, when trying to predict whether a price will go up or down). The point of this example was not to show that trees are a bad model for time series, but to illustrate a particular property of how trees make predictions.

[10] We picked this particular feature to add for illustration purposes. The choice is not particularly important.

[11] This follows from the Taylor expansion of the exponential map.

[12] You might have noticed at this point that many of the well-performing models achieved exactly the same accuracy of 0.972. This means that all of the models make exactly the same number of mistakes, which is four. If you compare the actual predictions, you can even see that they make exactly the same mistakes! This might be a consequence of the dataset being very small, or it may be because these points are really different from the rest.

[13] Because the probabilities are floating-point numbers, it is unlikely that they will both be exactly 0.500. However, if that happens, the prediction is made at random.

Chapter 3. Unsupervised Learning and Preprocessing

The second family of machine learning algorithms that we will discuss is unsupervised learning algorithms. Unsupervised learning subsumes all kinds of machine learning where there is no known output, no teacher to instruct the learning algorithm. In unsupervised learning, the learning algorithm is just shown the input data and asked to extract knowledge from this data.

Types of Unsupervised Learning

We will look into two kinds of unsupervised learning in this chapter: transformations of the dataset and clustering.

Unsupervised transformations of a dataset are algorithms that create a new representation of the data which might be easier for humans or other machine learning algorithms to understand compared to the original representation of the data. A common application of unsupervised transformations is dimensionality reduction, which takes a high-dimensional representation of the data, consisting of many features, and finds a new way to represent this data that summarizes the essential characteristics with fewer features. A common application for dimensionality reduction is reduction to two dimensions for visualization purposes.

Another application for unsupervised transformations is finding the parts or components that "make up" the data. An example of this is topic extraction on collections of text documents. Here, the task is to find the unknown topics that are talked about in each document, and to learn what topics appear in each document. This can be useful for tracking the discussion of themes like elections, gun control, or pop stars on social media.

Clustering algorithms, on the other hand, partition data into distinct groups of similar items. Consider the example of uploading photos to a social

media site. To allow you to organize your pictures, the site might want to group together pictures that show the same person. However, the site doesn't know which pictures show whom, and it doesn't know how many different people appear in your photo collection. A sensible approach would be to extract all the faces and divide them into groups of faces that look similar. Hopefully, these correspond to the same person, and the images can be grouped together for you.

Challenges in Unsupervised Learning

A major challenge in unsupervised learning is evaluating whether the algorithm learned something useful. Unsupervised learning algorithms are usually applied to data that does not contain any label information, so we don't know what the right output should be. Therefore, it is very hard to say whether a model "did well." For example, our hypothetical clustering algorithm could have grouped together all the pictures that show faces in profile and all the full-face pictures. This would certainly be a possible way to divide a collection of pictures of people's faces, but it's not the one we were looking for. However, there is no way for us to "tell" the algorithm what we are looking for, and often the only way to evaluate the result of an unsupervised algorithm is to inspect it manually.

As a consequence, unsupervised algorithms are used often in an exploratory setting, when a data scientist wants to understand the data better, rather than as part of a larger automatic system. Another common application for unsupervised algorithms is as a preprocessing step for supervised algorithms. Learning a new representation of the data can sometimes improve the accuracy of supervised algorithms, or can lead to reduced memory and time consumption.

Before we start with "real" unsupervised algorithms, we will briefly discuss some simple preprocessing methods that often come in handy. Even though preprocessing and scaling are often used in tandem with supervised

learning algorithms, scaling methods don't make use of the supervised information, making them unsupervised.

Preprocessing and Scaling

In the previous chapter we saw that some algorithms, like neural networks and SVMs, are very sensitive to the scaling of the data. Therefore, a common practice is to adjust the features so that the data representation is more suitable for these algorithms. Often, this is a simple per-feature rescaling and shift of the data. The following code (Figure 3-1) shows a simple example:

In[1]:

```
mglearn.plots.plot_scaling()
```

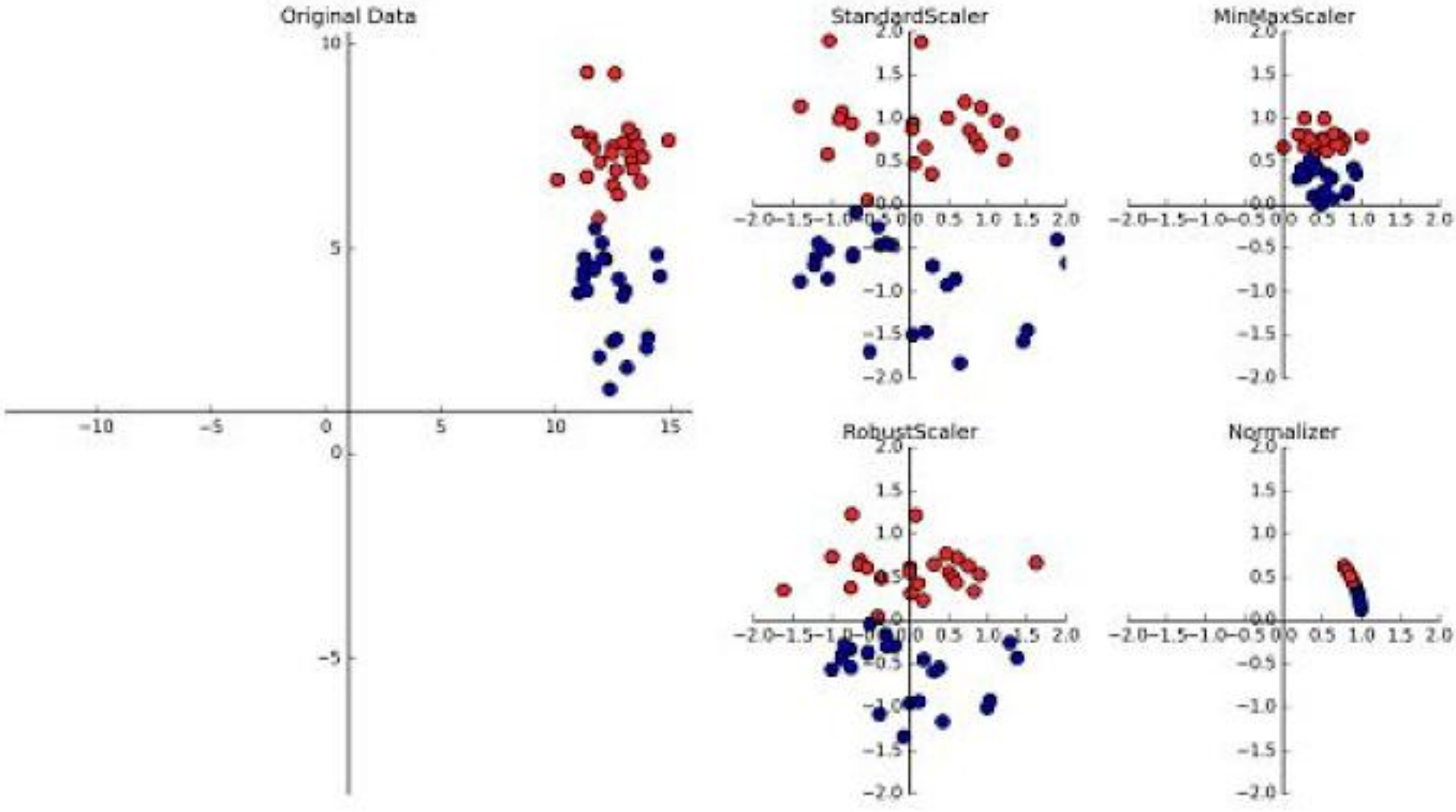


Figure 3-1. Different ways to rescale and preprocess a dataset

Different Kinds of Preprocessing

The first plot in Figure 3-1 shows a synthetic two-class classification dataset with two features. The first feature (the x-axis value) is between 10 and 15. The second feature (the y-axis value) is between around 1 and 9.

The following four plots show four different ways to transform the data that yield more standard ranges. The `StandardScaler` in `scikit-learn` ensures that for each feature the mean is 0 and the variance is 1, bringing all features to the same magnitude. However, this scaling does not ensure any particular minimum and maximum values for the features. The `RobustScaler` works similarly to the `StandardScaler` in that it ensures statistical properties for each feature that guarantee that they are on the same scale. However, the `RobustScaler` uses the median and quartiles,[1] instead of mean and variance. This makes the `RobustScaler` ignore data points that are very different from the rest (like measurement errors). These odd data points are also called *outliers*, and can lead to trouble for other scaling techniques.

The `MinMaxScaler`, on the other hand, shifts the data such that all features are exactly between 0 and 1. For the two-dimensional dataset this means all of the data is contained within the rectangle created by the x-axis between 0 and 1 and the y-axis between 0 and 1.

Finally, the `Normalizer` does a very different kind of rescaling. It scales each data point such that the feature vector has a Euclidean length of 1. In other words, it projects a data point on the circle (or sphere, in the case of higher dimensions) with a radius of 1. This means every data point is scaled by a different number (by the inverse of its length). This normalization is often used when only the direction (or angle) of the data matters, not the length of the feature vector.

Applying Data Transformations

Now that we've seen what the different kinds of transformations do, let's

apply them using `scikit-learn`. We will use the `cancer` dataset that we saw in Chapter 2. Preprocessing methods like the scalers are usually applied before applying a supervised machine learning algorithm. As an example, say we want to apply the kernel SVM (`SVC`) to the `cancer` dataset, and use `MinMaxScaler` for preprocessing the data. We start by loading our dataset and splitting it into a training set and a test set (we need separate training and test sets to evaluate the supervised model we will build after the preprocessing):

In[2]:

```
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
cancer = load_breast_cancer()

X_train, X_test, y_train, y_test = train_test_split(cancer.data,
cancer.target,
random_state=1)
print(X_train.shape)
print(X_test.shape)
```

Out[2]:

```
(426, 30)
(143, 30)
```

As a reminder, the dataset contains 569 data points, each represented by 30 measurements. We split the dataset into 426 samples for the training set and 143 samples for the test set.

As with the supervised models we built earlier, we first import the class that implements the preprocessing, and then instantiate it:

In[3]:

```
from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()
```

We then fit the scaler using the `fit` method, applied to the training data. For the `MinMaxScaler`, the `fit` method computes the minimum and maximum value of each feature on the training set. In contrast to the classifiers and regressors of Chapter 2, the scaler is only provided with the data (`X_train`) when `fit` is called, and `y_train` is not used:

In[4]:

```
scaler.fit(X_train)
```

Out[4]:

```
MinMaxScaler(copy=True, feature_range=(0, 1))
```

To apply the transformation that we just learned—that is, to actually *scale* the training data—we use the `transform` method of the scaler. The `transform` method is used in `scikit-learn` whenever a model returns a new representation of the data:

In[5]:

```
X_train_scaled = scaler.transform(X_train)

print("transformed shape: {}".format(X_train_scaled.shape))
```

```
print("per-feature minimum before scaling:\n
{}".format(X_train.min(axis=0)))
print("per-feature maximum before scaling:\n
{}".format(X_train.max(axis=0)))
print("per-feature minimum after scaling:\n {}".format(
    X_train_scaled.min(axis=0)))
print("per-feature maximum after scaling:\n {}".format(
    X_train_scaled.max(axis=0)))
```

Out[5]:

```
transformed shape: (426, 30)
per-feature minimum before scaling:
 [   6.98    9.71   43.79  143.50    0.05    0.02    0.      0.
0.11
     0.05    0.12    0.36    0.76    6.80    0.      0.      0.
0.
     0.01    0.      7.93   12.02   50.41  185.20    0.07    0.03
0.
     0.      0.16    0.06]
per-feature maximum before scaling:
 [   28.11    39.28   188.5   2501.0     0.16     0.29     0.43
0.2
     0.300    0.100    2.87     4.88    21.98   542.20     0.03
0.14
     0.400    0.050    0.06     0.03    36.04    49.54   251.20
4254.00
     0.220    0.940    1.17     0.29     0.58     0.15]
per-feature minimum after scaling:
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
0.  0.
   0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
per-feature maximum after scaling:
 [ 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
1.  1.
   1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.]
```

The transformed data has the same shape as the original data—the features are simply shifted and scaled. You can see that all of the features are now

between 0 and 1, as desired.

To apply the SVM to the scaled data, we also need to transform the test set. This is again done by calling the `transform` method, this time on `X_test`:

In[6]:

```
X_test_scaled = scaler.transform(X_test)

print("per-feature minimum after
scaling:\n{}".format(X_test_scaled.min(axis=0)))
print("per-feature maximum after
scaling:\n{}".format(X_test_scaled.max(axis=0)))
```

Out[6]:

```
per-feature minimum after scaling:
[ 0.034  0.023  0.031  0.011  0.141  0.044  0.     0.     0.154
-0.006
 -0.001  0.006  0.004  0.001  0.039  0.011  0.     0.    -0.032
0.007
  0.027  0.058  0.02   0.009  0.109  0.026  0.     0.    -0.
-0.002]
per-feature maximum after scaling:
[ 0.958  0.815  0.956  0.894  0.811  1.22   0.88   0.933  0.932
1.037
  0.427  0.498  0.441  0.284  0.487  0.739  0.767  0.629  1.337
0.391
  0.896  0.793  0.849  0.745  0.915  1.132  1.07   0.924  1.205
1.631]
```

Maybe somewhat surprisingly, you can see that for the test set, after scaling, the minimum and maximum are not 0 and 1. Some of the features are even outside the 0–1 range! The explanation is that the `MinMaxScaler` (and all the other scalers) always applies exactly the same transformation to

Shortcuts and Efficient Alternatives

Often, you want to `fit` a model on some dataset, and then `transform` it. This is a very common task, which can often be computed more efficiently than by simply calling `fit` and then `transform`. For this use case, all models that have a `transform` method also have a `fit_transform` method. Here is an example using `StandardScaler`:

In[8]:

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()


X_scaled = scaler.fit(X_train).transform(X_train)

X_scaled_d = scaler.fit_transform(X_train)
```

While `fit_transform` is not necessarily more efficient for all models, it is still good practice to use this method when trying to transform the training set.

The Effect of Preprocessing on Supervised Learning

Now let's go back to the `cancer` dataset and see the effect of using the `MinMaxScaler` on learning the SVC (this is a different way of doing the same scaling we did in Chapter 2). First, let's fit the SVC on the original data again for comparison:

In[9]:

```
from sklearn.svm import SVC
```

Learning

As we discussed earlier, transforming data using unsupervised learning can have many motivations. The most common motivations are visualization, compressing the data, and finding a representation that is more informative for further processing.

One of the simplest and most widely used algorithms for all of these is principal component analysis. We'll also look at two other algorithms: non-negative matrix factorization (NMF), which is commonly used for feature extraction, and t-SNE, which is commonly used for visualization using two-dimensional scatter plots.

Principal Component Analysis (PCA)

Principal component analysis is a method that rotates the dataset in a way such that the rotated features are statistically uncorrelated. This rotation is often followed by selecting only a subset of the new features, according to how important they are for explaining the data. The following example (Figure 3-3) illustrates the effect of PCA on a synthetic two-dimensional dataset:

In[12]:

```
mglearn.plots.plot_pca_illustration()
```

The first plot (top left) shows the original data points, colored to distinguish among them. The algorithm proceeds by first finding the direction of maximum variance, labeled "Component 1." This is the direction (or vector) in the data that contains most of the information, or in other words, the direction along which the features are most correlated with each other. Then, the algorithm finds the direction that contains the most information while being orthogonal (at a right angle) to the first direction. In two

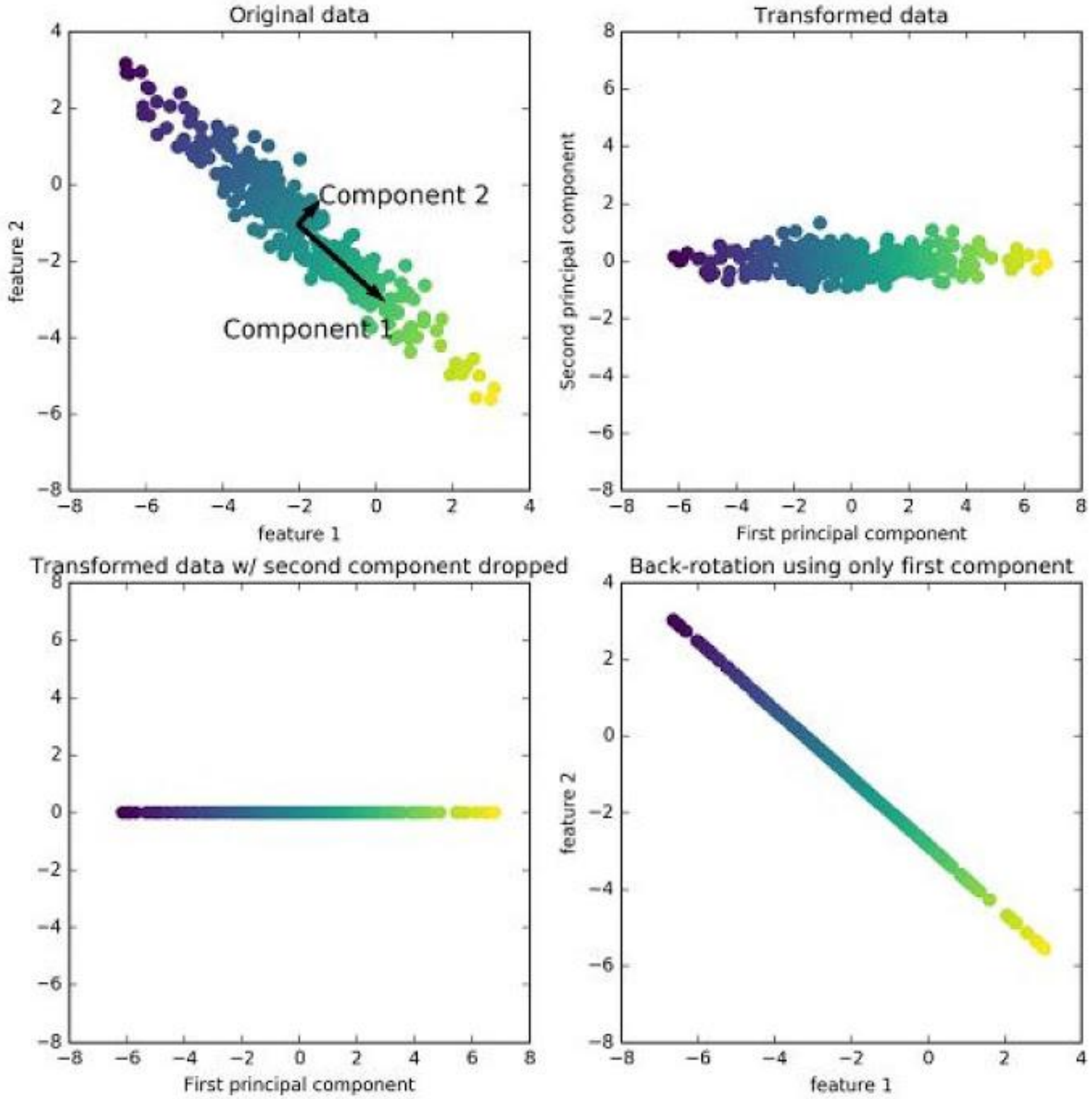


Figure 3-3. Transformation of data with PCA

The second plot (top right) shows the same data, but now rotated so that the first principal component aligns with the x-axis and the second principal component aligns with the y-axis. Before the rotation, the mean was subtracted from the data, so that the transformed data is centered around zero. In the rotated representation found by PCA, the two axes are

uncorrelated, meaning that the correlation matrix of the data in this representation is zero except for the diagonal.

We can use PCA for dimensionality reduction by retaining only some of the principal components. In this example, we might keep only the first principal component, as shown in the third panel in Figure 3-3 (bottom left). This reduces the data from a two-dimensional dataset to a one-dimensional dataset. Note, however, that instead of keeping only one of the original features, we found the most interesting direction (top left to bottom right in the first panel) and kept this direction, the first principal component.

Finally, we can undo the rotation and add the mean back to the data. This will result in the data shown in the last panel in Figure 3-3. These points are in the original feature space, but we kept only the information contained in the first principal component. This transformation is sometimes used to remove noise effects from the data or visualize what part of the information is retained using the principal components.

Applying PCA to the cancer dataset for visualization

One of the most common applications of PCA is visualizing high-dimensional datasets. As we saw in Chapter 1, it is hard to create scatter plots of data that has more than two features. For the Iris dataset, we were able to create a pair plot (Figure 1-3 in Chapter 1) that gave us a partial picture of the data by showing us all the possible combinations of two features. But if we want to look at the Breast Cancer dataset, even using a pair plot is tricky. This dataset has 30 features, which would result in 30 * 14 = 420 scatter plots! We'd never be able to look at all these plots in detail, let alone try to understand them.

There is an even simpler visualization we can use, though—computing histograms of each of the features for the two classes, benign and malignant

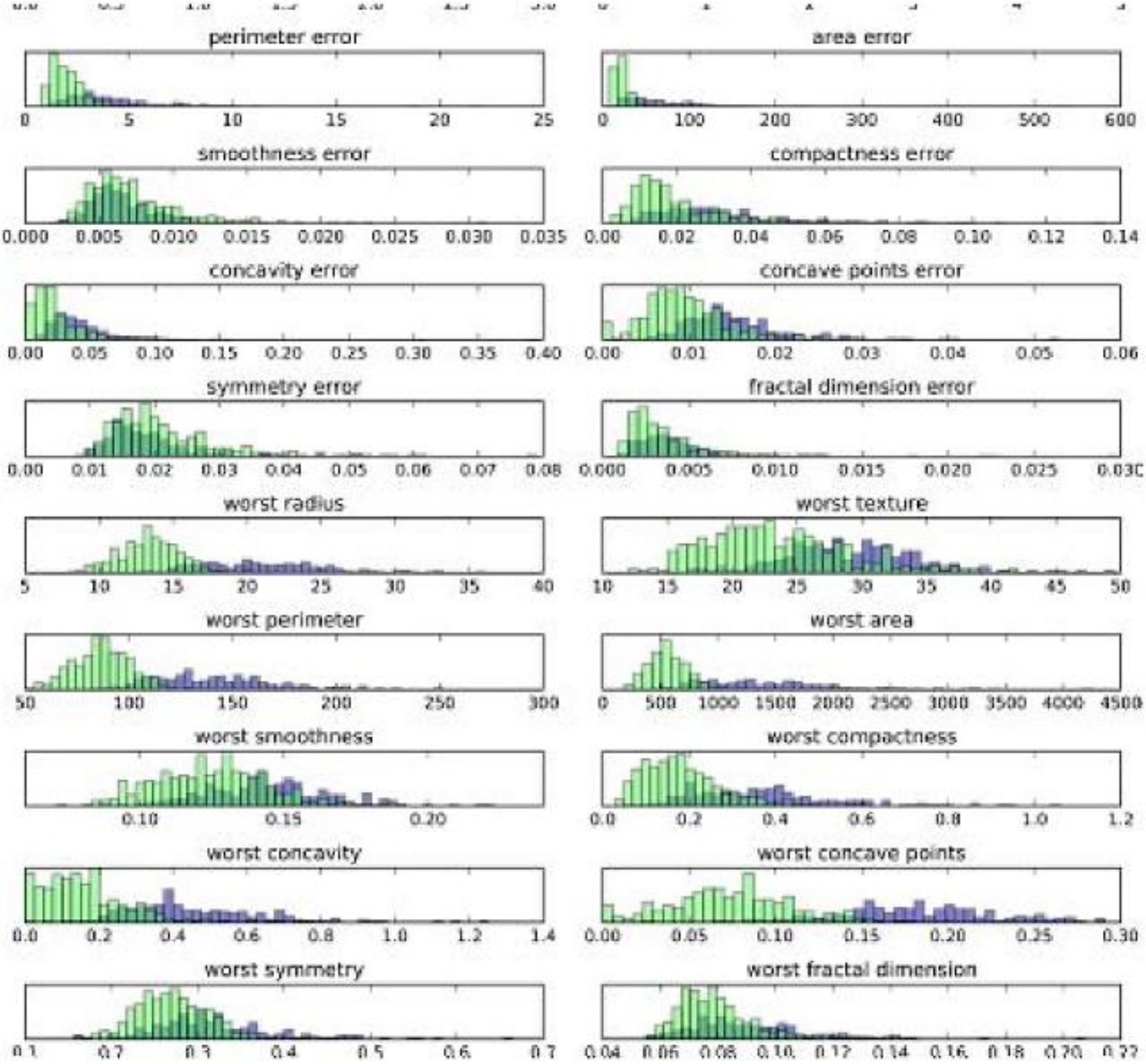


Figure 3-4. Per-class feature histograms on the Breast Cancer dataset

Here we create a histogram for each of the features, counting how often a data point appears with a feature in a certain range (called a *bin*). Each plot overlays two histograms, one for all of the points in the benign class (blue) and one for all the points in the malignant class (red). This gives us some idea of how each feature is distributed across the two classes, and allows us to venture a guess as to which features are better at distinguishing malignant and benign samples. For example, the feature "smoothness error" seems quite uninformative, because the two histograms mostly overlap,

while the feature “worst concave points” seems quite informative, because the histograms are quite disjoint.

However, this plot doesn’t show us anything about the interactions between variables and how these relate to the classes. Using PCA, we can capture the main interactions and get a slightly more complete picture. We can find the first two principal components, and visualize the data in this new two-dimensional space with a single scatter plot.

Before we apply PCA, we scale our data so that each feature has unit variance using `StandardScaler`:

In[14]:

```
from sklearn.datasets import load_breast_cancer
cancer = load_breast_cancer()

scaler = StandardScaler()
scaler.fit(cancer.data)
X_scaled = scaler.transform(cancer.data)
```

Learning the PCA transformation and applying it is as simple as applying a preprocessing transformation. We instantiate the PCA object, find the principal components by calling the `fit` method, and then apply the rotation and dimensionality reduction by calling `transform`. By default, `PCA` only rotates (and shifts) the data, but keeps all principal components. To reduce the dimensionality of the data, we need to specify how many components we want to keep when creating the `PCA` object:

In[15]:

```
from sklearn.decomposition import PCA
```

```
pca = PCA(n_components=2)

pca.fit(X_scaled)


X_pca = pca.transform(X_scaled)
print("Original shape: {}".format(str(X_scaled.shape)))
print("Reduced shape: {}".format(str(X_pca.shape)))
```

Out[15]:

```
Original shape: (569, 30)
Reduced shape: (569, 2)
```

We can now plot the first two principal components (Figure 3-5):

In[16]:

```
plt.figure(figsize=(8, 8))
mglearn.discrete_scatter(X_pca[:, 0], X_pca[:, 1], cancer.target)
plt.legend(cancer.target_names, loc="best")
plt.gca().set_aspect("equal")
plt.xlabel("First principal component")
plt.ylabel("Second principal component")
```

positive, but as we mentioned earlier, it doesn't matter which direction the arrow points in). That means that there is a general correlation between all features. As one measurement is high, the others are likely to be high as well. The second component has mixed signs, and both of the components involve all of the 30 features. This mixing of all features is what makes explaining the axes in Figure 3-6 so tricky.

Eigenfaces for feature extraction

Another application of PCA that we mentioned earlier is feature extraction. The idea behind feature extraction is that it is possible to find a representation of your data that is better suited to analysis than the raw representation you were given. A great example of an application where feature extraction is helpful is with images. Images are made up of pixels, usually stored as red, green, and blue (RGB) intensities. Objects in images are usually made up of thousands of pixels, and only together are they meaningful.

We will give a very simple application of feature extraction on images using PCA, by working with face images from the Labeled Faces in the Wild dataset. This dataset contains face images of celebrities downloaded from the Internet, and it includes faces of politicians, singers, actors, and athletes from the early 2000s. We use grayscale versions of these images, and scale them down for faster processing. You can see some of the images in Figure 3-7:

In[20]:

```
from sklearn.datasets import fetch_lfw_people
people = fetch_lfw_people(min_faces_per_person=20, resize=0.7)
image_shape = people.images[0].shape

fig, axes = plt.subplots(2, 5, figsize=(15, 8),
                         subplot_kw={'xticks': (), 'yticks': ()})
```