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 nearest neighbor, which is the closest training data point to the point we want to make a prediction for. The prediction is then simply the known output for this training point. Figure 2-4 illustrates this for the case of classification on the `forge` dataset:

In[10]:

```
mglearn.plots.plot_knn_classification(n_neighbors=1)
```

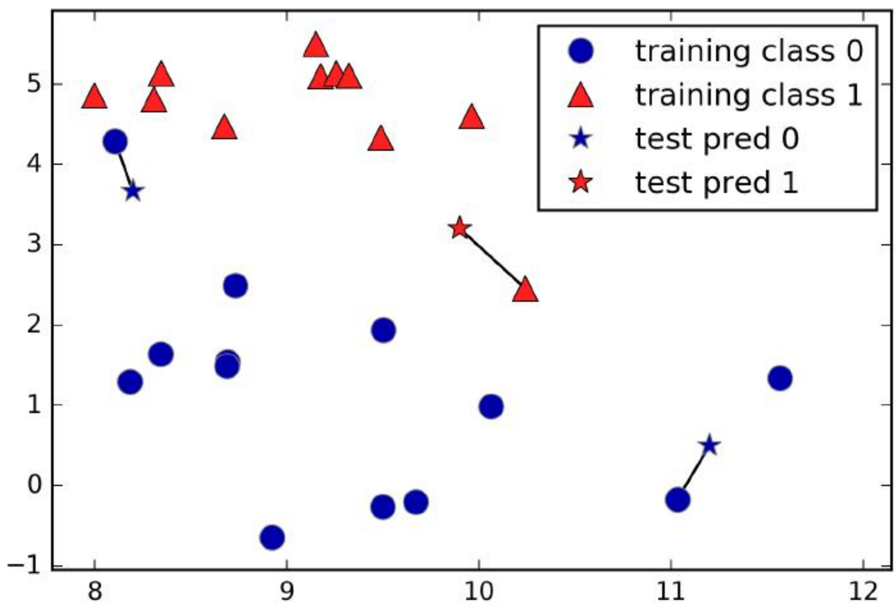

Figure 2-4. Predictions made by the one-nearest-neighbor model on the forge dataset

Here, we added three new data points, shown as stars. For each of them, we marked the closest point in the training set. The prediction of the one-nearest-neighbor algorithm is the label of that point (shown by the color of the cross).

Instead of considering only the closest neighbor, we can also consider an arbitrary number, *k*, of neighbors. This is where the name of the *k*-nearest neighbors algorithm comes from. When considering more than one neighbor, we use *voting* to assign a label. This means that for each test point, we count how many neighbors belong to class 0 and how many neighbors belong to class 1. We then assign the class that is more frequent: in other words, the majority class among the *k*-nearest neighbors. The following example (Figure 2-5) uses the three closest neighbors:

In[11]:

```
mglearn.plots.plot_knn_classification(n_neighbors=3)
```

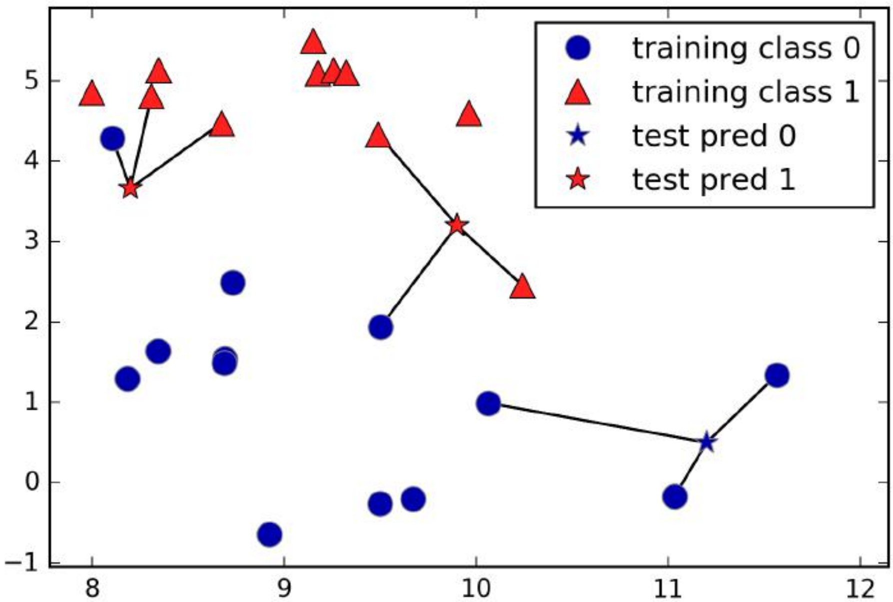

Figure 2-5. Predictions made by the three-nearest-neighbors model on the forge dataset

Again, the prediction is shown as the color of the cross. You can see that the prediction for the new data point at the top left is not the same as the prediction when we used only one neighbor.

While this illustration is for a binary classification problem, this method can be applied to datasets with any number of classes. For more classes, we count how many neighbors belong to each class and again predict the most common class.

Now let's look at how we can apply the *k*-nearest neighbors algorithm using `scikit-learn`. First, we split our data into a training and a test set so we can evaluate generalization performance, as discussed in Chapter 1:

In[12]:

```
from sklearn.model_selection import train_test_split
X, y = mglearn.datasets.make_forge()

X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
```

Next, we import and instantiate the class. This is when we can set parameters, like the number of neighbors to use. Here, we set it to 3:

In[13]:

```
from sklearn.neighbors import KNeighborsClassifier
clf = KNeighborsClassifier(n_neighbors=3)
```

Now, we fit the classifier using the training set. For `KNeighborsClassifier` this means storing the dataset, so we can compute neighbors during prediction:

In[14]:

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

To make predictions on the test data, we call the `predict` method. For each data point in the test set, this computes its nearest neighbors in the training set and finds the most common class among these:

In[15]:

```
print("Test set predictions: {}".format(clf.predict(X_test)))
```

Out[15]:

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

To evaluate how well our model generalizes, we can call the `score` method with the test data together with the test labels:

In[16]:

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

Out[16]:

```
Test set accuracy: 0.86
```

We see that our model is about 86% accurate, meaning the model predicted the class correctly for 86% of the samples in the test dataset.

Analyzing KNeighborsClassifier

For two-dimensional datasets, we can also illustrate the prediction for all possible test points in the xy-plane. We color the plane according to the class that would be assigned to a point in this region. This lets us view the *decision boundary*, which is the divide between where the algorithm assigns class 0 versus where it assigns class 1.

The following code produces the visualizations of the decision boundaries for one, three, and nine neighbors shown in Figure 2-6:

In[17]:

```
fig, axes = plt.subplots(1, 3, figsize=(10, 3))

for n_neighbors, ax in zip([1, 3, 9], axes):
    # the fit method returns the object self, so we can instantiate
    # and fit in one line
    clf = KNeighborsClassifier(n_neighbors=n_neighbors).fit(X, y)
    mglearn.plots.plot_2d_separator(clf, X, fill=True, eps=0.5, ax=ax, alpha=.4)
    mglearn.discrete_scatter(X[:, 0], X[:, 1], y, ax=ax)
    ax.set_title("{} neighbor(s)".format(n_neighbors))
    ax.set_xlabel("feature 0")
    ax.set_ylabel("feature 1")
axes[0].legend(loc=3)
```

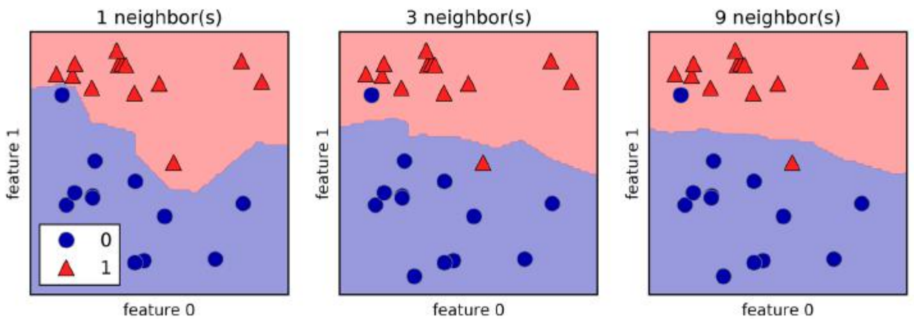

Figure 2-6. Decision boundaries created by the nearest neighbors model for different values of n_neighbors

As you can see on the left in the figure, using a single neighbor results in a decision boundary that follows the training data closely. Considering more and more neighbors leads to a smoother decision boundary. A smoother boundary corresponds to a simpler model. In other words, using few neighbors corresponds to high model complexity (as shown on the right side of Figure 2-1), and using many neighbors corresponds to low model complexity (as shown on the left side of Figure 2-1). If you consider the extreme case where the number of neighbors is the number of all data points in the training set, each test point would have exactly the same neighbors (all training points) and all predictions would be the same: the class that is most frequent in the training set.

Let's investigate whether we can confirm the connection between model complexity and generalization that we discussed earlier. We will do this on the real-world Breast Cancer dataset. We begin by splitting the dataset into a training and a test set. Then we evaluate training and test set performance with different numbers of neighbors. The results are shown in Figure 2-7:

In[18]:

```
from sklearn.datasets import load_breast_cancer

cancer = load_breast_cancer()
X_train, X_test, y_train, y_test = train_test_split(
    cancer.data, cancer.target, stratify=cancer.target, random_state=66)

training_accuracy = []
test_accuracy = []
# try n_neighbors from 1 to 10
neighbors_settings = range(1, 11)

for n_neighbors in neighbors_settings:
    # build the model
    clf = KNeighborsClassifier(n_neighbors=n_neighbors)
    clf.fit(X_train, y_train)
    # record training set accuracy
    training_accuracy.append(clf.score(X_train, y_train))
    # record generalization accuracy
    test_accuracy.append(clf.score(X_test, y_test))

plt.plot(neighbors_settings, training_accuracy, label="training accuracy")
plt.plot(neighbors_settings, test_accuracy, label="test accuracy")
plt.ylabel("Accuracy")
plt.xlabel("n_neighbors")
plt.legend()
```

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.

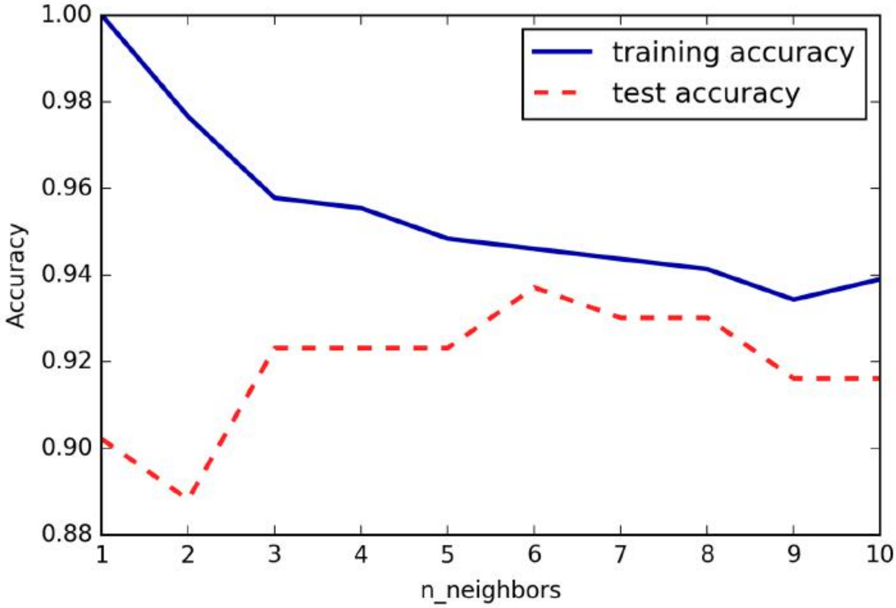

Figure 2-7. Comparison of training and test accuracy as a function of n_neighbors

~~k-neighbors regression~~

~~There is also a regression variant of the *k* nearest neighbors algorithm. Again, let's start by using the single nearest neighbor, this time using the `wave` dataset. We've added three test data points as green stars on the x-axis. The prediction using a single neighbor is just the target value of the nearest neighbor. These are shown as blue stars in Figure 2-8:~~

~~**In[19]:**~~

```
mglearn.plots.plot_knn_regression(n_neighbors=1)
```