CHAPTER 2

Classification from Simple Training Sets

Classification is the problem of predicting a discrete class label. Classes are also called targets, labels, or categories. Classification is applied by training a classifier algorithm on training data to predict how new data is classified.

A machine learning classification data set consists of features (X) and targets (y) where input variables *X* describe known discrete output variables *y*. Feature data is typically referred to as the feature set (or feature space). Classification is considered supervised learning because we know the targets that correspond to the feature set.

Whew! That's a lot. So, let's look at a simple example to help you understand how classification works. Suppose we have a data set consisting of four categories of fruit, namely, "apple," "orange," "lemon," and "lime." Each data element (or row) describes one piece of fruit (the target) by its mass, width, height, and color (the features). So, an apple can be distinguished from an orange by different values of mass, width, height, and color.

In the example, the class label is the type of fruit. Each type of fruit is discrete. That is, an apple is easily distinguished from the other types of fruits. The goal is to predict type of fruit based on its mass, width, height, and color.

To train a data set, we split data into train-test subsets. Train data features are referred to as *X_train* and targets as *y_train*. Test data features are referred to as *X_test* and targets as *y_test*. We then build a classification model to train on X_train and y_train data. Once the model is trained, we can validate and predict from X_test and y_test data because the model has not seen the test data. By holding test data out of the training process, it effectively acts as *new* data.

Tip Never train on test data to keep it pure.

© David Paper 2020
D. Paper, *Hands-on Scikit-Learn for Machine Learning Applications*,
https://doi.org/10.1007/978-1-4842-5373-1_2

A typical train-test split is 70%/30%, but the ratio should be chosen based on the size of the data set. If the data set is small, a 30% test set may not contain all of the classes or enough information to properly validate. Also, the distribution of different classes in both train and test sets should be equal to the actual data set. The best way to ensure this distribution is to split train-test subsets randomly. Fortunately, Scikit-Learn's train_test_split package randomizes the split automatically, but its default train-test split is 75%/25%.

I recommend some general steps when tackling machine learning problems. First, always split data for training and validations purposes. Second, try scaling data to potentially improve performance. Third, experiment with training and test sizes. Fourth, always begin with a baseline model, simple algorithm or an algorithm based on prior experience with a data set. And, start with an algorithm's default hyperparameters. Fifth, experiment with more complex models since Scikit-Learn is efficient and allows easy model substitution. When working with big data sets, try drawing random samples to reduce computational expense. When working with high-dimensional data sets, try dimensionality reduction with PCA or LDA to reduce computational expense. Sixth, tune the best algorithms identified in earlier steps to get the best performance. Finally, experiment some more. Machine learning is very time intensive and rigorous, so be patient and don't give up.

Tip Always begin training with an algorithm's default hyperparameters.

Simple Data Sets

We concentrate on four simple data sets to introduce machine learning classification: wine, digits, banking, and make_moons. We didn't introduce make_moons in Chapter 1 because it is contrived. That is, Scikit-Learn provides the foundation for make_moons and we construct it as we see fit.

Classifying Wine Data

The code example shown in Listing 2-1 classifies wine data.

Listing 2-1. Classify load_wine data

```
from sklearn.datasets import load_wine
from sklearn.preprocessing import StandardScaler
from sklearn.discriminant_analysis import\
     LinearDiscriminantAnalysis as LDA
from sklearn.linear_model import SGDClassifier
from sklearn.model_selection import train_test_split
from sklearn import metrics
from random import *

if __name__ == "__main__":
    br = '\n'
    data = load_wine()
    X = data.data
    y = data.target
    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.30, random_state=0)
    lda = LDA().fit(X_train, y_train)
    print (lda, br)
    lda_name = lda.__class__.__name__
    y_pred = lda.predict(X_train)
    accuracy = metrics.accuracy_score(y_train, y_pred)
    accuracy = str(accuracy * 100) + '%'
    print (lda_name + ':')
    print ('train:', accuracy)
    y_pred_test = lda.predict(X_test)
    accuracy = metrics.accuracy_score(y_test, y_pred_test)
    accuracy = str(round(accuracy * 100, 2)) + '%'
    print ('test: ', accuracy, br)
    print('Confusion Matrix', lda_name)
    print(metrics.confusion_matrix(y_test, lda.predict(X_test)), br)
    std_scale = StandardScaler().fit(X_train)
    X_train = std_scale.transform(X_train)
    X_test = std_scale.transform(X_test)
```

```
    sgd = SGDClassifier(max_iter=5, random_state=0)
    print (sgd, br)
    sgd.fit(X_train, y_train)
    sgd_name = sgd.__class__.__name__
    y_pred = sgd.predict(X_train)
    y_pred_test = sgd.predict(X_test)
    print (sgd_name + ':')
    print('train: {:.2%}'.format(metrics.accuracy_score\(y_train, y_pred)))
    print('test:  {:.2%}\n'.format(metrics.accuracy_score\(y_test, y_pred_
    test)))
    print('Confusion Matrix', sgd_name)
    print(metrics.confusion_matrix(y_test, sgd.predict(X_test)), br)
    n, ls = 100, []
    for i, row in enumerate(range(n)):
        rs = randint(0, 100)
        sgd = SGDClassifier(max_iter=5, random_state=0)
        sgd.fit(X_train, y_train)
        y_pred = sgd.predict(X_test)
        accuracy = metrics.accuracy_score(y_test, y_pred)
        ls.append(accuracy)
    avg = sum(ls) / len(ls)
    print ('MCS (true test accuracy):', avg)
```

Go ahead and execute the code from Listing 2-1. Remember that you can find the example from the book's example download. You don't need to type the example by hand. It's easier to access the example download and copy/paste.

Your output from executing Listing 2-1 should resemble the following:

```
LinearDiscriminantAnalysis(n_components=None, priors=None, shrinkage=None,
                           solver='svd', store_covariance=False, tol=0.0001)

LinearDiscriminantAnalysis:
train: 100.0%
test:  98.15%
```

```
Confusion Matrix LinearDiscriminantAnalysis
[[19  0  0]
 [ 1 21  0]
 [ 0  0 13]]

SGDClassifier(alpha=0.0001, average=False, class_weight=None,
       early_stopping=False, epsilon=0.1, eta0=0.0,
       fit_intercept=True, l1_ratio=0.15,
       learning_rate='optimal', loss='hinge', max_iter=5,
       n_iter=None, n_iter_no_change=5, n_jobs=None,
       penalty='l2', power_t=0.5, random_state=0, shuffle=True,
       tol=None, validation_fraction=0.1, verbose=0,
       warm_start=False)

SGDClassifier:
train: 100.00%
test:  100.00%

Confusion Matrix SGDClassifier
[[19  0  0]
 [ 0 22  0]
 [ 0  0 13]]

MCS (true test accuracy): 1.0
```

The code begins by importing metrics, random, and requisite packages. The main block begins by loading data and splitting it into train-test subsets. Notice that we adjusted the test size to 30%. Next, a LinearDiscriminantAnalysis (LDA) model is created and trained on the train set. You can fiddle with test size to see if your accuracy improves. But, don't make it too big. Your model needs training data to better understand and learn.

Tip To see a model's hyperparameters, just print the variable that holds the model after creation (e.g., print (lda)).

LDA was introduced in Chapter 1 as an unsupervised learning model for dimensionality reduction. LDA is a very interesting model in that it performs unsupervised dimensionality reduction *and* supervised classification.

Data scaling doesn't improve LDA performance, so the model trains on unscaled data. Accuracy scores are then computed on both train and test subsets. Performance accuracy is typically reported only on test data. However, it is useful to get train and test accuracy to see how well the model fits the data. In this case, the model fits the data very well because train accuracy and test accuracy are very similar. If train accuracy is well above test accuracy, the model is overfitting the data.

The code continues by displaying a confusion matrix. A *confusion matrix* describes the performance of a classification model (or classifier) on a set of test data for which the true values are known. The diagonal consisting of 19, 21, and 13 is where the model correctly classified. The model only misclassified one data element from the test set, which makes perfect sense with a test accuracy of over 98%. Next, we scale data because SGDClassifier is known to perform better with scaled data. The model is trained, and train and test accuracy are displayed along with the confusion matrix. With this model, classification was perfect.

The final part of the code is optional. It employs Monte Carlo experiments to validate performance of the SGDClassifier on the wine data. *Monte Carlo experiments* use randomness to solve deterministic (or supervised) problems. With a perfect test accuracy of 100%, we should be a bit skeptical. So, we ran 100 Monte Carlo experiments to obtain the actual test performance. As you can see, we get 100%!

Monte Carlo experiments are a fantastic method for deriving accuracy, but are incredibly computationally expensive. We were safe in this cased because the data set is small and simple. With big data sets with high-dimensional data, Monte Carlo experiments are not very practical.

LinearDiscriminantAnalysis and SGDClassifier were not chosen randomly. The algorithms were identified strategically as best performers through rigorous trial-and-error experimentation and research.

Tip Each data set is different, so choose algorithms strategically through trial-and-error experimentation and of course research.

Classifying Digits

The first code example shown in Listing 2-2 loads the data and splits it into train-test subsets. Next, data is trained with classifiers GaussianNB, SGDClassifier, and svm. Algorithm svm is the best performer. The code then identifies and visualizes misclassifications. The code concludes by visualizing the first misclassification.

Listing 2-2. Classify load_digits data

```
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
from sklearn.linear_model import SGDClassifier
from sklearn.svm import SVC
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score
from sklearn.metrics import confusion_matrix
from sklearn.metrics import classification_report
import matplotlib.pyplot as plt
import seaborn as sns

def find_misses(test, pred):
    return [i for i, row in enumerate(test) if row != pred[i]]

if __name__ == "__main__":
    br = '\n'
    digits = load_digits()
    X = digits.data
    y = digits.target
    X_train, X_test, y_train, y_test = train_test_split\
                                       (X, y, random_state=0)
    gnb = GaussianNB().fit(X_train, y_train)
    gnb_name = gnb.__class__.__name__
    y_pred = gnb.predict(X_test)
    accuracy = accuracy_score(y_test, y_pred)
    print (gnb_name + ' \'test\' accuracy:', accuracy)
    scaler = StandardScaler()
```

```
X_train_std = scaler.fit_transform(X_train)
X_test_std = scaler.fit_transform(X_test)
sgd = SGDClassifier(random_state=0, max_iter=1000, tol=0.001)
sgd_name = sgd.__class__.__name__
sgd.fit(X_train_std, y_train)
y_pred = sgd.predict(X_test_std)
accuracy = accuracy_score(y_test, y_pred)
print (sgd_name + ' \'test\' accuracy:', accuracy)
svm = SVC(gamma='auto').fit(X_train_std, y_train)
svm_name = svm.__class__.__name__
y_pred = svm.predict(X_test_std)
accuracy = accuracy_score(y_test, y_pred)
print (svm_name + ' \'test\' accuracy:', accuracy, br)
indx = find_misses(y_test, y_pred)
print ('total misclassifications (' + str(svm_name) +\ '):', len(indx), br)
print ('pred', 'actual')
misses = [(y_pred[row], y_test[row], i)
          for i, row in enumerate(indx)]
[print (row[0], '  ', row[1]) for row in misses]
img_indx = misses[0][2]
img_pred = misses[0][0]
img_act = misses[0][1]
text = str(img_pred)
print(classification_report(y_test, y_pred))
cm = confusion_matrix(y_test, y_pred)
plt.figure(1)
ax = plt.axes()
sns.heatmap(cm.T, annot=True, fmt="d",
            cmap='gist_ncar_r', ax=ax)
title = svm_name + ' confusion matrix'
ax.set_title(title)
plt.xlabel('true value')
plt.ylabel('predicted value')
test_images = X_test.reshape(-1, 8, 8)
plt.figure(2)
```

```
    plt.title('1st misclassifcation')
    plt.imshow(test_images[img_indx], cmap='gray', interpolation='gaussian')
    plt.text(0, 0.05, text, color='r', bbox=dict(facecolor='white'))
    plt.show()
```

After executing code from Listing 2-2, your output should resemble the following:

```
GaussianNB 'test' accuracy: 0.8333333333333334
SGDClassifier 'test' accuracy: 0.9377777777777778
SVC 'test' accuracy: 0.9822222222222222

total misclassifications (SVC): 8

pred actual
7    2
1    8
7    9
9    5
4    7
4    3
2    8
4    1
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        37
           1       0.98      0.98      0.98        43
           2       0.98      0.98      0.98        44
           3       1.00      0.98      0.99        45
           4       0.93      1.00      0.96        38
           5       1.00      0.98      0.99        48
           6       1.00      1.00      1.00        52
           7       0.96      0.98      0.97        48
           8       1.00      0.96      0.98        48
           9       0.98      0.98      0.98        47

   micro avg       0.98      0.98      0.98       450
   macro avg       0.98      0.98      0.98       450
weighted avg       0.98      0.98      0.98       450
```

Listing 2-2 also displays Figures 2-1 and 2-2. Figure 2-1 displays the confusion matrix for the best performing algorithm, which is svm. You see *SVC* displayed because we are implementing the SVC implementation of the svm algorithm. SVC implementation utilizes C-support vector classification, which is represented as svm.SVC in Scikit-Learn. Figure 2-2 displays the first misclassification from the prediction set, which is digit 2 misclassified as digit 7. If we look at the confusion matrix, we can see this misclassification at the intersection of predicted value row for digit 7 and true value column for digit 2. So, the true value (digit 2) was incorrectly predicted (or misclassified) as digit 7.

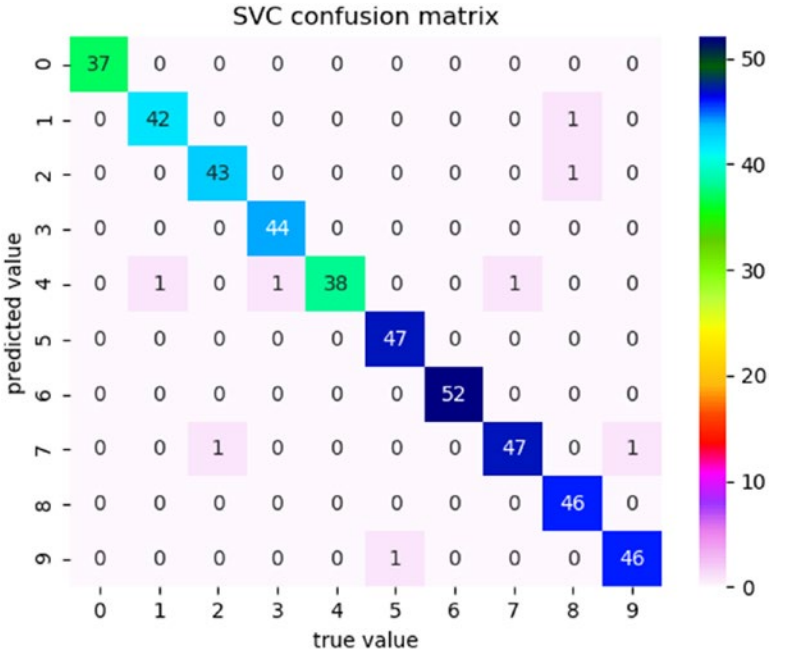

Figure 2-1. *Confusion matrix for the svm.SVC algorithm*

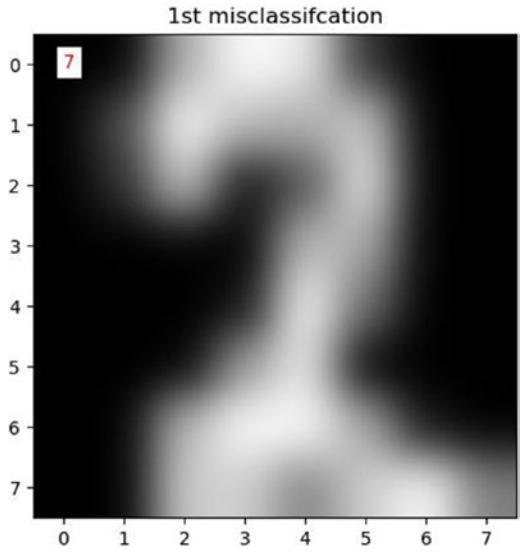

Figure 2-2. *First misclassification from the prediction set*

The code example begins by importing GaussianNB, confusion_matrix, and classification_report as well as other requisite packages. GaussianNB is an excellent baseline algorithm because it is fast, performs well with many classification problems, and has few hyperparameters to tune.

Tip If you have no experience with a classification data set, GaussianNB is a great place to start because it is simple, fast, easy to understand, and has few hyperparameters to tune.

Function find_misses returns a list of misclassified digits. The main block loads data, splits it into train-test subsets, and trains with GaussianNB, SGDClassifier, and svm.

GaussianNB is a probabilistic classifier based on applying Bayes' theorem with strong independence assumptions between features. SGDClassifier is a classifier that implements a plain stochastic gradient descent learning routine that supports different loss functions and penalties for classification. Support vector machine (svm) builds a model that assigns new examples to one category or the other.

The code then displays test accuracy for all three models. Since svm.SVC scores the highest, we use it to identify misclassifications. Total misclassifications are then displayed. The code continues by displaying how each misclassification was rendered. So, the first misclassified digit was 2 and it was misclassified as 7.

Next, the code creates a classification report (for the svm.SVC algorithm) that presents precision, recall, and f1_score scores for each digit. Accuracy is a great way to report a score, but f1_score, especially, is one to consider including because it is the most conservative.

The code concludes by displaying a svm.SVC confusion matrix and the first misclassification where 2 was misclassified as 7. The figure is also interesting because it presents the actual image of digit 2 with the way it was classified in red as digit 7.

Once a great performing algorithm is identified for a data set, I highly recommend creating a confusion matrix visualization. Not only it easy to understand how well an algorithm classified targets, it allows deeper scrutiny of where the algorithm didn't perform as expected.

Tip Creating a confusion matrix visualization is an excellent way to get a sense of how an algorithm performs.

Although we obtained a high accuracy score from svm in the previous example, Scikit-Learn allows us to substitute classifiers very easily. So, the next code example goes a bit crazy by training wine data with six additional classifiers. Keep in mind that the data set is small and simple. With larger and more complex data, substituting classifiers can be computationally expensive.

The next code example shown in Listing 2-3 classifies wine data with several Scikit-Learn algorithms to identify promising ones for improved performance.

Listing 2-3. Classifying load_digits with various algorithms

```
import humanfriendly as hf
import time
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression,\
     LogisticRegressionCV
from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import RandomForestClassifier,\
     ExtraTreesClassifier, GradientBoostingClassifier
from sklearn.preprocessing import StandardScaler
```

```
from sklearn.metrics import accuracy_score
from sklearn.metrics import f1_score

def get_scores(model, Xtest, ytest, avg):
    y_pred = model.predict(Xtest)
    accuracy = accuracy_score(ytest, y_pred)
    f1 = f1_score(ytest, y_pred, average=avg)
    return (accuracy, f1)

def get_time(time):
    return hf.format_timespan(time, detailed=True)

if __name__ == "__main__":
    br = '\n'
    digits = load_digits()
    X = digits.data
    y = digits.target
    X_train, X_test, y_train, y_test = train_test_split\
                                        (X, y, random_state=0)
    scaler = StandardScaler()
    X_train_std = scaler.fit_transform(X_train)
    X_test_std = scaler.fit_transform(X_test)

    lr = LogisticRegression(random_state=0, solver='lbfgs',
                            multi_class='auto', max_iter=4000)
    lr.fit(X_train_std, y_train)
    lr_name = lr.__class__.__name__
    acc, f1 = get_scores(lr, X_test_std, y_test, 'micro')
    print (lr_name + ' scaled \'test\':')
    print ('accuracy:', acc, ', f1_score:', f1, br)
    softmax = LogisticRegression(multi_class="multinomial",
                                 solver="lbfgs", max_iter=4000,
                                 C=10, random_state=0)
    softmax.fit(X_train_std, y_train)
    acc, f1 = get_scores(softmax, X_test_std, y_test, 'micro')
    print (lr_name + ' (softmax) scaled \'test\':')
    print ('accuracy:', acc, ', f1_score:', f1, br)
```

```
rf = RandomForestClassifier(random_state=0, n_estimators=100)
rf.fit(X_train_std, y_train)
rf_name = rf.__class__.__name__
acc, f1 = get_scores(rf, X_test_std, y_test, 'micro')
print (rf_name + ' \'test\':')
print ('accuracy:', acc, ', f1_score:', f1, br)
et = ExtraTreesClassifier(random_state=0, n_estimators=100)
et.fit(X_train, y_train)
et_name = et.__class__.__name__
acc, f1 = get_scores(et, X_test, y_test, 'micro')
print (et_name + ' \'test\':')
print ('accuracy:', acc, ', f1_score:', f1, br)
gboost_clf = GradientBoostingClassifier(random_state=0)
gb_name = gboost_clf.__class__.__name__
gboost_clf.fit(X_train, y_train)
acc, f1 = get_scores(gboost_clf, X_test, y_test, 'micro')
print (gb_name + ' \'test\':')
print ('accuracy:', acc, ', f1_score:', f1, br)
knn_clf = KNeighborsClassifier().fit(X_train, y_train)
knn_name = knn_clf.__class__.__name__
acc, f1 = get_scores(knn_clf, X_test, y_test, 'micro')
print (knn_name + ' \'test\':')
print ('accuracy:', acc, ', f1_score:', f1, br)
start = time.perf_counter()
lr_cv = LogisticRegressionCV(random_state=0, cv=5, multi_class='auto',
                             max_iter=4000)
lr_cv_name = lr_cv.__class__.__name__
lr_cv.fit(X, y)
end = time.perf_counter()
elapsed_ls = end - start
timer = get_time(elapsed_ls)
print (lr_cv_name + ' timer:', timer)
acc, f1 = get_scores(lr_cv, X_test, y_test, 'micro')
print (lr_cv_name + ' \'test\':')
print ('accuracy:', acc, ', f1_score:', f1)
```

After executing code from Listing 2-3, your output should resemble the following:

```
LogisticRegression scaled 'test':
accuracy: 0.9733333333333334 , f1_score: 0.9733333333333334

LogisticRegression (softmax) scaled 'test':
accuracy: 0.9644444444444444 , f1_score: 0.9644444444444444

RandomForestClassifier 'test':
accuracy: 0.9755555555555555 , f1_score: 0.9755555555555555

ExtraTreesClassifier 'test':
accuracy: 0.9822222222222222 , f1_score: 0.9822222222222222

GradientBoostingClassifier 'test':
accuracy: 0.9622222222222222 , f1_score: 0.9622222222222222

KNeighborsClassifier 'test':
accuracy: 0.98 , f1_score: 0.98

LogisticRegressionCV timer: 49 seconds and 38.45 milliseconds
LogisticRegressionCV 'test':
accuracy: 0.9822222222222222 , f1_score: 0.9822222222222222
```

The code begins by importing humanfriendly, time, LogisticRegression, LogisticRegressionCV, KNeighborsClassifier, GradientBoostingClassifier, and ExtraTreesClassifier as well as other requisite packages. Function get_scores returns accuracy and f1_score. Function get_time returns elapsed time. It facilitates finding how long it takes an algorithm to train a data set.

LogisticRegression is a classification algorithm traditionally limited to only two-class classification problems. Softmax (multinomial logistic regression) classification uses logistic regression for multiclass classification. RandomForestClassifier is an ensemble learning method that constructs a multitude of decision trees at training time and outputs the class that is the mode of the classes. ExtraTreesClassifier implements a meta estimator that fits a number of randomized decision trees (or extra trees) on various subsamples of the data set and uses averaging to improve predictive accuracy and control overfitting. GradientBoostingClassifier produces a prediction model in the form of weak prediction models (typically decision trees). KNeighborsClassifier implements the k-nearest neighbors' vote where input consists of the *k* closest training examples in

the feature space. *Feature space* refers to the n-dimensions where your features exist. LogisticRegression uses a logistic function to model data. LogisticRegressionCV uses logistic regression to implement cross-validation estimation.

Cross-validation (CV) divides data into *n* number of subsets and iterates *n* times. Through each iteration, one of the *n* subsets is held out as the test set while the rest are used for training. Every iteration uses a different subset. So, accuracy and error are averaged over all *n* trials. The resultant accuracy is very good, but CV can be computationally expensive.

GradientBoostingClassifier and ExtraTreesClassifier are ensemble methods similar to RandomForestClassifier in that they fit (or train) a number of decision trees on the data and average results to improve predictive accuracy.

Tip You may have to install the humanfriendly package since it isn't installed automatically by Anaconda. Open a new Anaconda prompt and install as shown in Listing 2-4.

Listing 2-4. Install a new package

```
pip install humanfriendly
```

The main block begins by loading and splitting data into train-test subsets. Each of the algorithms train the data and scores are displayed. Notice that over 46 seconds are consumed by LogisticRegressionCV. Although all of the algorithms performed admirably, we still couldn't beat 98.22% accuracy.

Classifying Bank Data

The first code example shown in Listing 2-5 loads bank data from a CSV file. Next, the *education* feature is engineered to make it more presentable.

Feature engineering is creating features (based on domain knowledge of the data) that make machine learning algorithms work. Although feature engineering is fundamental to machine learning application, it is both difficult and expensive.

The code then transforms categorical features to numerical to enable algorithm training. This transformation is typically referred to as encoding.

Tip Machine learning algorithms only operate on numerical data.

Finally, the five most important features are displayed along with data features and class counts. The feature set and targets are saved in NumPy files.

Listing 2-5. Engineering and wrangling bank data

```
import numpy as np, pandas as pd
from sklearn.ensemble import RandomForestClassifier

if __name__ == "__main__":
    br = '\n'
    f = 'data/bank.csv'
    data = pd.read_csv(f)
    print ('original "education" categories:')
    print (data.education.unique(), br)
    data['education'] = np.where(data['education'] == 'basic.9y',
                                 'basic', data['education'])
    data['education'] = np.where(data['education'] == 'basic.6y',
                                 'basic', data['education'])
    data['education'] = np.where(data['education'] == 'basic.4y',
                                 'basic', data['education'])
    data['education'] = np.where(data['education'] == 'high.school',
                                 'high_school', data.education)
    data['education'] = np.where(data['education'] == 'professional.course',
                                 'professional', data['education'])
    data['education'] = np.where(data['education'] == 'university.degree',
                                 'university', data['education'])
    print ('engineered "education" categories:')
    print (data.education.unique(), br)
    print ('target value counts:')
    print (data.y.value_counts(), br)
    data_X = data.loc[:, data.columns != 'y']
    cat_vars = ['job', 'marital', 'education', 'default', 'housing',
                'loan', 'contact', 'month', 'day_of_week', 'poutcome']
    data_new = pd.get_dummies(data_X, columns=cat_vars)
```

```
    X = data_new.values
    y = data.y.values
    attributes = list(data_X)
    rf = RandomForestClassifier(random_state=0, n_estimators=100)
    rf.fit(X, y)
    rf_name = rf.__class__.__name__
    feature_importances = rf.feature_importances_
    importance = sorted(zip(feature_importances, attributes), reverse=True)
    n = 5
    print (n, 'most important features' + ' (' + rf_name + '):')
    [print (row) for i, row in enumerate(importance) if i < n]
    print ()
    features_file = 'data/features'
    np.save(features_file, attributes)
    features = np.load('data/features.npy')
    print ('features:')
    print (features, br)
    y_file = 'data/y'
    X_file = 'data/X'
    np.save(y_file, y)
    np.save(X_file, X)
    d = {}
    dvc = data.y.value_counts()
    d['no'], d['yes'] = dvc['no'], dvc['yes']
    dvc_file = 'data/value_counts'
    np.save(dvc_file, d)
    d = np.load('data/value_counts.npy')
    print ('class counts:', d)
```

After executing code from Listing 2-5, your output should resemble the following:

```
original "education" categories:
['basic.4y' 'high.school' 'basic.6y' 'basic.9y' 'professional.course'
 'unknown' 'university.degree' 'illiterate']

engineered "education" categories:
['basic' 'high_school' 'professional' 'unknown' 'university' 'illiterate']
```

```
target value counts:
no     36548
yes     4640
Name: y, dtype: int64

5 most important features (RandomForestClassifier):
(0.28697175347986037, 'job')
(0.08761238456151103, 'month')
(0.0797624194551633, 'age')
(0.05492109153356108, 'day_of_week')
(0.04027613029914145, 'marital')

features:
['age' 'job' 'marital' 'education' 'default' 'housing' 'loan' 'contact'
'month' 'day_of_week' 'duration' 'campaign' 'pdays' 'previous' 'poutcome'
'emp.var.rate' 'cons.price.idx' 'cons.conf.idx' 'euribor3m' 'nr.employed']

class counts: {'no': 36548, 'yes': 4640}
```

The code example begins by importing requisite packages. The main block reads the data and displays the original values from the *education* feature. The code continues by feature engineering the feature and displays the new values. Notice how difficult it is to feature engineer just a single feature. Next, categorical features are encoded by the pandas *get_dummies* function to one hot encoding (OHE) vectors. Scikit-Learn expects feature data to be numeric, which is why we need to encode them.

OHE vectors are also called dummy variables. OHE is a good choice since it is one of the most common methods for dealing with categorical data in machine learning. OHE takes each category value and turns it into a binary vector of size i (where i is the number of values in category i) and makes all columns equal to zero except the category column. For example, marital status is either "married," "single," or "divorced" in our data set. If someone is married, OHE encodes a [1 0 0] vector. If single, OHE encodes a [0 1 0] vector. Finally, if divorced, OHE encodes a [0 0 1] vector. Simply, the 1 bit is hot to indicate the category that fits the data element.

The code then creates feature set X and target y from the transformed data set. Feature importance is displayed with the help of RandomForestClassifier. Next, X and y are saved in NumPy files. Finally, class counts are created, saved, and displayed. It is useful to view class counts to see the balance between targets.

Notice that we have more *no* values than *yes* values. So, the data set is a bit imbalanced. This occurrence is commonly referred to as imbalanced class distribution, which is when the number of observations belonging to one class is significantly lower than those belonging to other classes. In our case, the balance between yes and no is about 12.6%. So, we don't have a major problem. A rate (or event rate) less than 5% is a problem because machine learning algorithms can produce unsatisfactory classification when this happens.

Now that bank data is prepared, we can run experiments to identify high-performing classification algorithms as demonstrated in the next code example, which is shown in Listing 2-6. Keep in mind that many hours of experimentation led to the choice of algorithms for this example.

Listing 2-6. Classifying bank data

```
import numpy as np, pandas as pd, random
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.ensemble import RandomForestClassifier,\
     ExtraTreesClassifier
from sklearn.svm import SVC
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import f1_score
from sklearn.metrics import confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns

def get_scores(model, xtrain, ytrain, xtest, ytest, scoring):
    ypred = model.predict(xtest)
    train = model.score(xtrain, ytrain)
    test = model.score(xtest, y_test)
    f1 = f1_score(ytest, ypred, average=scoring)
    return (train, test, f1)

def prep_data(data, target):
    d = [data[i] for i, _ in enumerate(data)]
    t = [target[i] for i, _ in enumerate(target)]
    return list(zip(d, t))
```

```
def create_sample(d, n, replace='yes'):
    if replace == 'yes': s = random.sample(d, n)
    else: s = [random.choice(d) for i, _ in enumerate(d) if i < n]
    Xs = [row[0] for i, row in enumerate(s)]
    ys = [row[1] for i, row in enumerate(s)]
    return np.array(Xs), np.array(ys)

if __name__ == "__main__":
    br = '\n'
    X = np.load('data/X.npy')
    y = np.load('data/y.npy')
    print ('full data set shape for X and y:')
    print (X.shape, y.shape, br)
    X_train, X_test, y_train, y_test = train_test_split\
                                        (X, y, random_state=0)
    et = ExtraTreesClassifier(random_state=0, n_estimators=100)
    et.fit(X_train, y_train)
    et_scores = get_scores(et, X_train, y_train, X_test, y_test, 'micro')
    print (et.__class__.__name__ + '(train, test, f1_score):')
    print (et_scores, br)
    rf = RandomForestClassifier(random_state=0, n_estimators=100)
    rf.fit(X_train, y_train)
    rf_scores = get_scores(rf, X_train, y_train, X_test, y_test, 'micro')
    print (rf.__class__.__name__ + '(train, test, f1_score):')
    print (rf_scores, br)
    sample_size = 4000
    data = prep_data(X, y)
    Xs, ys = create_sample(data, sample_size, replace='no')
    print ('sample data set shape for X and y:')
    print (Xs.shape, ys.shape, br)
    X_train, X_test, y_train, y_test = train_test_split\
                                        (Xs, ys, random_state=0)
    scaler = StandardScaler().fit(X_train)
    X_train_std, X_test_std = scaler.transform(X_train),\
                              scaler.transform(X_test)
    knn = KNeighborsClassifier().fit(X_train, y_train)
```

```
knn_scores = get_scores(knn, X_train, y_train, X_test, y_test, 'micro')
print (knn.__class__.__name__ + '(train, test, f1_score):')
print (knn_scores, br)
svm = SVC(random_state=0, gamma='scale')
svm.fit(X_train_std, y_train)
svm_scores = get_scores(svm, X_train_std, y_train, X_test_std, y_test,
                        'micro')
print (svm.__class__.__name__ + '(train, test, f1_score):')
print (svm_scores, br)
knn_name, svm_name = knn.__class__.__name__,\
                     svm.__class__.__name__
y_pred_knn = knn.predict(X_test)
cm_knn = confusion_matrix(y_test, y_pred_knn)
cm_knn_T = cm_knn.T
y_pred_svm = svm.predict(X_test_std)
cm_svm = confusion_matrix(y_test, y_pred_svm)
cm_svm_T = cm_svm.T
plt.figure(knn.__class__.__name__)
ax = plt.axes()
sns.heatmap(cm_knn_T, annot=True, fmt="d", cmap='gist_ncar_r', cbar=False)
ax.set_title(str(knn_name) + ' confusion matrix')
plt.xlabel('true label')
plt.ylabel('predicted label')
plt.figure(str(svm_name) + ' confusion matrix' )
ax = plt.axes()
sns.heatmap(cm_svm_T, annot=True, fmt="d", cmap='gist_ncar_r', cbar=False)
ax.set_title(svm_name)
plt.xlabel('true label')
plt.ylabel('predicted label')
cnt_no, cnt_yes = 0, 0
for i, row in enumerate(y_test):
    if row == 'no': cnt_no += 1
    elif row == 'yes': cnt_yes += 1
cnt_no, cnt_yes = str(cnt_no), str(cnt_yes)
print ('true =>', 'no: ' + cnt_no + ', yes: ' + cnt_yes, br)
```

```
    p_no, p_nox = cm_knn_T[0][0], cm_knn_T[0][1]
    p_yes, p_yesx = cm_knn_T[1][1], cm_knn_T[1][0]
    print ('knn classification report:')
    print ('predict \'no\':', p_no, '(' +\str(p_nox) + ' misclassifed)')
    print ('predict \'yes\':', p_yes, '(' +\str(p_yesx) + ' misclassifed)', br)
    p_no, p_nox = cm_svm_T[0][0], cm_svm_T[0][1]
    p_yes, p_yesx = cm_svm_T[1][1], cm_svm_T[1][0]
    print ('svm classification report:')
    print ('predict \'no\':', p_no, '(' +\str(p_nox) + ' misclassifed)')
    print ('predict \'yes\':', p_yes, '(' +\str(p_yesx) + ' misclassifed)')
    plt.show()
```

After executing code from Listing 2-6, your output should resemble the following:

```
full data set shape for X and y:
(41188, 61) (41188,)

ExtraTreesClassifier(train, test, f1_score):
(1.0, 0.9009420219481402, 0.9009420219481401)

RandomForestClassifier(train, test, f1_score):
(0.9999676281117478, 0.9121103233951636, 0.9121103233951636)

sample data set shape for X and y:
(4000, 61) (4000,)

KNeighborsClassifier(train, test, f1_score):
(0.9323333333333333, 0.916, 0.916)

SVC(train, test, f1_score):
(0.9376666666666666, 0.92, 0.92)

true => no: 902, yes: 98

knn classification report:
predict 'no': 869 (51 misclassifed)
predict 'yes': 47 (33 misclassifed)

svm classification report:
predict 'no': 883 (61 misclassifed)
predict 'yes': 37 (19 misclassifed)
```

Listing 2-6 also displays Figures 2-3 and 2-4. Figure 2-3 displays the confusion matrix for KNeighborsClassifier and Figure 2-4 displays the confusion matrix for svm.SVC.

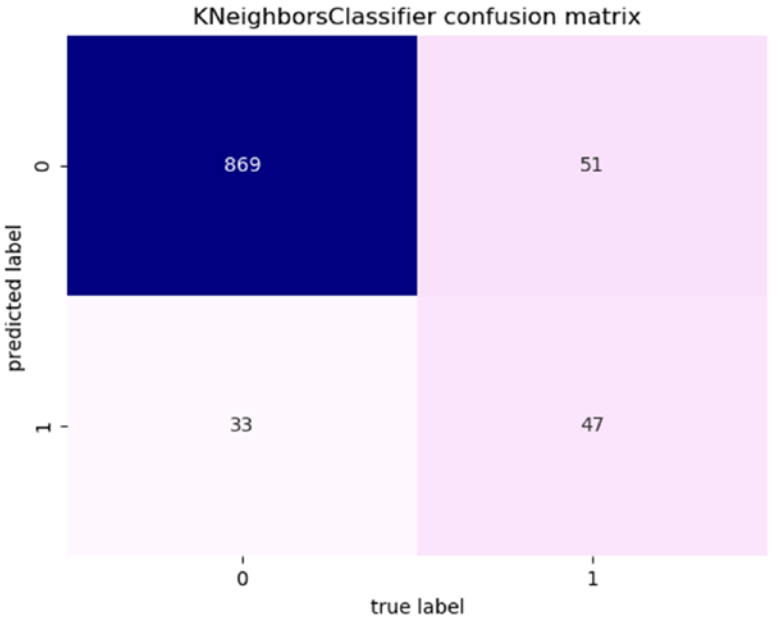

Figure 2-3. *KNeighborsClassifier confusion matrix*

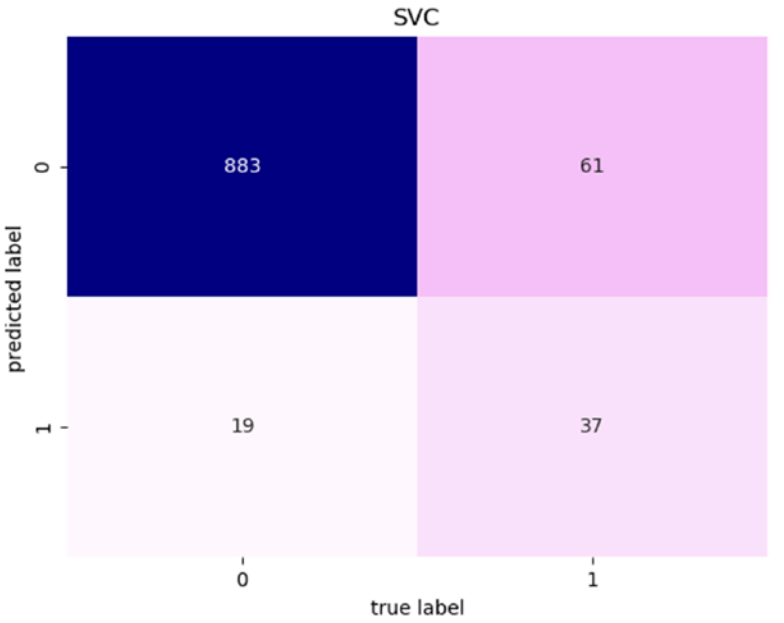

Figure 2-4. *svm.SVC confusion matrix*

The code begins by importing requisite packages. Function get_scores returns train and test accuracy scores. Function prep_data converts NumPy matrices to lists of vectors for easier manipulation of data elements for sampling. Function create_sample builds a random sample and returns it as X and y NumPy matrices.

Scikit-Learn algorithms can only train data represented as NumPy. The main block loads X and y from NumPy files created in the previous example. X and y are split into train-test subsets. The code then trains data with ExtraTreesClassifier and RandomForestClassifier. A sample of 4000 is drawn so that we can efficiently train with KNeighborsClassifier and svm.SVC. These two algorithms are excellent classifiers, but are computationally expensive with large data sets.

Confusion matrices for KNeighborsClassifier and svm.SVC are then displayed because they fit the data better. That is, accuracy was better and there was less overfitting with these models. The code concludes by calculating the balance of target values for the data and misclassifications by KNeighborsClassifier and svm.

Of note is that KNeighborsClassifier and svm.SVC performed better than the other algorithms based on a sample *less than 10%* of the original data. This is actually very impressive!

The UCI Machine Learning Repository includes a randomly selected sample from the bank data with 10% of the examples. For completeness, the next example shown in Listing 2-7 tests accuracy on this sample.

Listing 2-7. Classifying UCI Irvine sample bank data

```
import pandas as pd, numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.ensemble import RandomForestClassifier,\
     ExtraTreesClassifier
from sklearn.svm import SVC
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import f1_score

def get_scores(model, xtrain, ytrain, xtest, ytest, scoring):
    ypred = model.predict(xtest)
    train = model.score(xtrain, ytrain)
    test = model.score(xtest, y_test)
    f1 = f1_score(ytest, ypred, average=scoring)
    return (train, test, f1)
```

```
if __name__ == "__main__":
    br = '\n'
    f = 'data/bank_sample.csv'
    data = pd.read_csv(f)
    print ('data shape:', data.shape, br)
    data['education'] =\
                      np.where(data['education'] == 'basic.9y',
                               'basic', data['education'])
    data['education'] = np.where(data['education'] == 'basic.6y',
                                 'basic', data['education'])
    data['education'] = np.where(data['education'] == 'basic.4y',
                                 'basic', data['education'])
    data['education'] = np.where(data['education'] == 'high.school',
                                 'high_school', data.education)
    data['education'] = np.where(data['education'] == 'professional.course',
                                 'professional', data['education'])
    data['education'] = np.where(data['education'] == 'university.degree',
                                 'university', data['education'])
    data_X = data.loc[:, data.columns != 'y']
    cat_vars = ['job', 'marital', 'education', 'default', 'housing',
                'loan', 'contact', 'month', 'day_of_week', 'poutcome']
    data_new = pd.get_dummies(data_X, columns=cat_vars)
    attributes = list(data_X)
    y = data.y.values
    X = data_new.values
    X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
    rf = RandomForestClassifier(random_state=0, n_estimators=100)
    rf.fit(X_train, y_train)
    rf_name = rf.__class__.__name__
    rf_scores = get_scores(rf, X_train, y_train, X_test, y_test, 'micro')
    print (rf.__class__.__name__ + '(train, test, f1_score):')
    print (rf_scores, br)
    et = ExtraTreesClassifier(random_state=0, n_estimators=100)
    et.fit(X_train, y_train)
```

```
    et_name = et.__class__.__name__
    et_scores = get_scores(et, X_train, y_train, X_test, y_test, 'micro')
    print (et.__class__.__name__ + '(train, test, f1_score):')
    print (et_scores, br)
    scaler = StandardScaler().fit(X_train)
    X_train_std, X_test_std = scaler.transform(X_train),\
                              scaler.transform(X_test)
    knn = KNeighborsClassifier().fit(X_train, y_train)
    knn_scores = get_scores(knn, X_train, y_train, X_test, y_test, 'micro')
    print (knn.__class__.__name__ + '(train, test, f1_score):')
    print (knn_scores, br)
    svm = SVC(random_state=0, gamma='scale')
    svm.fit(X_train_std, y_train)
    svm_scores = get_scores(svm, X_train_std, y_train, X_test_std, y_test,
                            'micro')
    print (svm.__class__.__name__ + '(train, test, f1_score):')
    print (svm_scores)
```

After executing code from Listing 2-7, your output should resemble the following:

```
data shape: (4119, 21)

RandomForestClassifier(train, test, f1_score):
(1.0, 0.9058252427184466, 0.9058252427184466)

ExtraTreesClassifier(train, test, f1_score):
(1.0, 0.8990291262135922, 0.8990291262135922)

KNeighborsClassifier(train, test, f1_score):
(0.9323405632890903, 0.8883495145631068, 0.8883495145631068)

SVC(train, test, f1_score):
(0.9494982194885077, 0.9, 0.9)
```

The code begins by importing requisite packages. Function get_scores returns accuracy scores. The main block loads the sample, engineers the *education* feature, and encodes categorical features to OHE form. We had to feature engineer *education* for this example because we didn't draw the sample from the full data set upon which we had already engineered the feature.

The code continues by loading NumPy data into X and y, splitting it into train-test subsets, and training with RandomForestClassifier. Accuracy is then displayed. The remainder of the code trains with ExtraTreesClassifier, KNeighborsClassifier, and svm.SVC and displays accuracy scores.

The sample we created performed at least as well as the one from the UCI repository. Our sample was even a bit smaller, which means that our sampling technique is more than adequate.

Classifying make_moons

Scikit-Learn make_moons data is used primarily to visualize clustering and classification algorithms. However, it is also a great data set to get a sense of how classification algorithms attempt to separate binary target labels (or binary classification). Deployment of make_moons describes two interleaving circles with associated data points in 2D space.

Through visualization, we can easily see the separation between the two (or binary) labels. If human eyes can easily differentiate such separation in 2D space, classification algorithms should be able to do the same. We can test this with an example.

The first code example shown in Listing 2-8 creates a data set with 1000 elements, places feature data and its associated target into a Pandas DataFrame, and plots the result. Each feature element represents an x and y coordinate for plotting in 2D space. Each target represents the feature's label, which is a binary value of either 0 or 1.

Listing 2-8. Plot make_moons

```
import matplotlib.pyplot as plt, pandas as pd
from sklearn import datasets

if __name__ == "__main__":
    br = '\n'
    X, y = datasets.make_moons(n_samples=1000, shuffle=True, noise=0.2,
                               random_state=0)
    df = pd.DataFrame(dict(x=X[:,0], y=X[:,1], label=y))
    colors = {0:'magenta', 1:'cyan'}
    fig, ax = plt.subplots()
    data = df.groupby('label')
    for key, label in data:
```

```
        label.plot(ax=ax, kind='scatter', x='x', y='y', label=key,
                   color=colors[key])
    plt.show()
```

After executing code from Listing 2-8, your output should resemble the following visualization shown in Figure 2-5:

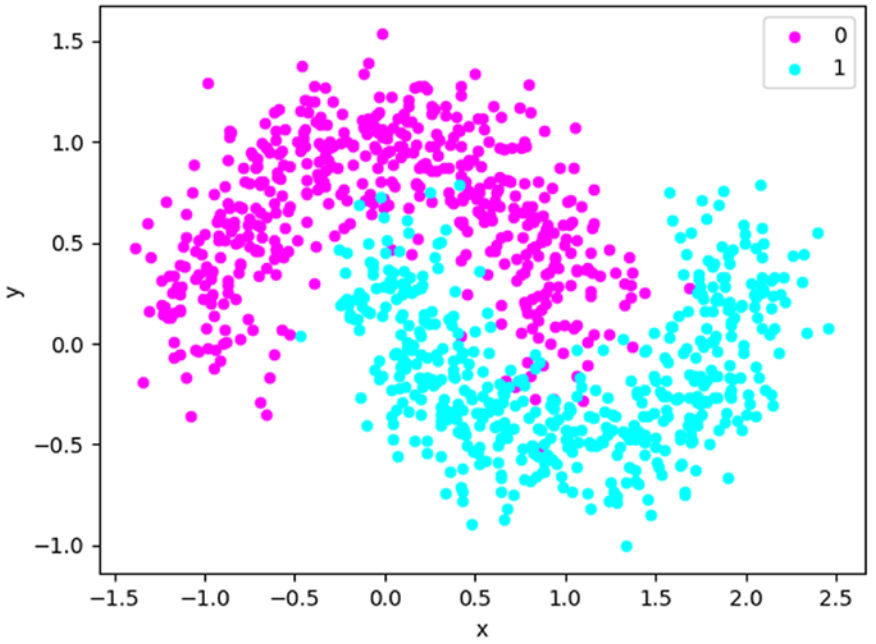

Figure 2-5. *Visualization of randomly generated make_moons data*

The next code example shown in Listing 2-9 creates a make_moons data set of 1000 elements, splits it into train-test subsets, and trains with svm.SVC and KNeighborsClassifier. I intentionally picked these two algorithms because I knew they would do a great job of binary classification since they look at every data point.

Listing 2-9. Classify make_moons

```
from sklearn import datasets
from sklearn.neighbors import KNeighborsClassifier
from sklearn import svm
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
```

```
def get_scores(model, Xtrain, Xtest, ytrain, ytest):
    y_ptrain = model.predict(Xtrain)
    y_ptest = model.predict(Xtest)
    acc_train = accuracy_score(ytrain, y_ptrain)
    acc_test = accuracy_score(ytest, y_ptest)
    name = model.__class__.__name__
    return (name, acc_train, acc_test)

if __name__ == "__main__":
    br = '\n'
    X, y = datasets.make_moons(n_samples=1000, shuffle=True, noise=0.2,
                               random_state=0)
    X_train, X_test, y_train, y_test = train_test_split(X, y, random_
                                                        state=0)
    knn = KNeighborsClassifier().fit(X_train, y_train)
    accuracy = get_scores(knn, X_train, X_test, y_train, y_test)
    print ('<<' + str(accuracy[0]) + '>>')
    print ('train:', accuracy[1], 'test:', accuracy[2], br)
    svm = svm.SVC(gamma='scale', random_state=0)
    svm.fit(X_train, y_train)
    accuracy = get_scores(svm, X_train, X_test, y_train, y_test)
    print ('<<' + str(accuracy[0]) + '>>')
    print ('train:', accuracy[1], 'test:', accuracy[2])
```

After executing code from Listing 2-9, your output should resemble the following:

```
<<KNeighborsClassifier>>
train: 0.9666666666666667 test: 0.964

<<SVC>>
train: 0.9653333333333334 test: 0.96
```

The code example begins by importing requisite packages. Function get_scores returns model name and train and test accuracy scores. The main block begins by loading sample data and splitting it into train-test subsets. It continues by training data with KNeighborsClassifier and svm.SVC and reporting accuracy scores. As expected, both algorithms recognized the labels very accurately with essentially no overfitting.

The final code example shown in Listing 2-10 extends our knowledge by splitting data into train, test, and validate subsets. KNeighborsClassifier is used to train and enable reporting.

Listing 2-10. Classify make_moons on train, validate, and test subsets

```
from sklearn.datasets import make_moons
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

def get_scores(model, Xtrain, ytrain, Xtest, ytest, Xvalid, yvalid):
    y_ptrain = model.predict(Xtrain)
    y_ptest = model.predict(Xtest)
    y_pvalid = model.predict(Xvalid)
    acc_train = accuracy_score(ytrain, y_ptrain)
    acc_test = accuracy_score(ytest, y_ptest)
    acc_valid = accuracy_score(yvalid, y_pvalid)
    name = model.__class__.__name__
    return (name, acc_train, acc_test, acc_valid)

if __name__ == "__main__":
    br = '\n'
    X_train, y_train = make_moons(n_samples=1000, shuffle=True, noise=0.2,
                                  random_state=0)
    X_test, y_test = make_moons(n_samples=1000, shuffle=True, noise=0.2,
                                random_state=0)
    X_valid, y_valid = make_moons(n_samples=10000, shuffle=True, noise=0.2,
                                  random_state=0)
    knn = KNeighborsClassifier().fit(X_train, y_train)
    accuracy = get_scores(knn, X_train, y_train, X_test, y_test, X_valid,
                          y_valid)
    print ('train test valid split (technique 1):')
    print ('<<' + str(accuracy[0]) + '>>')
    print ('train:', accuracy[1], 'test:', accuracy[2], 'valid:', accuracy[3])
    print ('sample split:', X_train.shape, X_test.shape, X_valid.shape)
    print ()
```

```
    X, y = make_moons(n_samples=1000, shuffle=True, noise=0.2, random_state=0)
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
                                                        random_state=0)
    X_train, X_val, y_train, y_val = train_test_split(X_train, y_train,
                                                      test_size=0.25,
                                                      random_state=0)
    knn = KNeighborsClassifier().fit(X_train, y_train)
    accuracy = get_scores(knn, X_train, y_train, X_test, y_test, X_valid,
                          y_valid)
    print ('train test valid split (technique 2):')
    print ('<<' + str(accuracy[0]) + '>>')
    print ('train:', accuracy[1], 'test:', accuracy[2], 'valid:', accuracy[3])
    print ('sample split:', X_train.shape, X_test.shape, X_val.shape)
```

After executing code from Listing 2-10, your output should resemble the following:

```
train test valid split (technique 1):
<<KNeighborsClassifier>>
train: 0.969 test: 0.969 valid: 0.9688
sample split: (1000, 2) (1000, 2) (10000, 2)

train test valid split (technique 2):
<<KNeighborsClassifier>>
train: 0.9616666666666667 test: 0.975 valid: 0.9694
sample split: (600, 2) (200, 2) (200, 2)
```

The code begins importing requisite packages. Function get_scores is expanded to account for validation scores. The main block begins by creating three separate test, train, and validation subsets. With this technique, we create three data sets of the same size. Although this technique produces excellent results, it is much more computationally expensive as data sets become larger and larger. Actually, this technique is three times more expensive because three data sets are created and trained. KNeighborsClassifier is used to train, validate, and test. The second technique is very common because it splits one data set into train, validate, and test. Again, KNeighborsClassifier is used. Results from both techniques are comparable and excellent as expected.

Tip Test data should only be used once a model is completely trained from training and validation phases so it can provide an unbiased evaluation of a final model fit on training data.

In industry, machine learning engineers experiment with data problems by splitting it into train, test, and validate subsets prior to training. Training data is used to fit (or train) the model. The model sees and learns from training data.

Validation data is used to evaluate a model. Machine learning engineers use validation data to fine-tune the model's hyperparameters. Test data provides an unbiased evaluation of a final model fit based on what was learned from fitting training data and tuning hyperparameters with validation data.