CHAPTER 6

Scikit-Learn Classifier Tuning from Complex Training Sets

Now that we have practiced tuning low-dimensional (or simple) data, we are ready to experiment tuning high-dimensional (or complex) data sets. *Low-dimensional* data consists of a limited number of features, whereas *high-dimensional* data consists of a very high number of features.

The term most commonly used to describe the dimensionality of a data set in machine learning literature is feature space. *Feature space* refers to the collection of features used to characterize the data set. That is, feature space refers to the n-dimensions where your variables live (not including a target variable if it is present).

Consistent with tuning low-dimensional data, we follow a structured process when tuning high-dimensional data:

a) Always begin with default hyperparameters using baseline algorithms.

b) Experiment with training and test sizes.

c) Use dimensionality reduction when working with high-dimensional data.

d) Draw random samples when working with large data sets.

e) Scale data (where appropriate) to potentially increase performance.

f) Use GridSearchCV or RandomizedSearchCV to tune.

g) Once tuned with baseline algorithms, experiment with complex algorithms.

D. Paper, *Hands-on Scikit-Learn for Machine Learning Applications*,
https://doi.org/10.1007/978-1-4842-5373-1_6

Tuning Data Sets

We concentrate on three data sets: fetch_1fw_people, MNIST, and fetch_20newsgroups. The fetch_1fw_people data set contains 1288 face images and seven targets. Each face image is represented by a 50 × 37 matrix of pixels. The MNIST data set contains 70000 examples of handwritten digit images labeled from 0 to 9. Each digit is represented by a 28 × 28 matrix. The fetch_20newsgroups data set consists of approximately 18000 posts on 20 topics. Data is split into a training and testing sets. The split is based on messages posted before and after a specific date.

Tuning fetch_1fw_people

Face recognition is a *very* complex topic in machine learning. But, Scikit-Learn provides fetch_1fw_people that is a wonderful data set upon which to experiment and learn. Through experience and experimentation, I identified two Scikit-Learn algorithms – SGDClassifier and svm.SVC – that work relatively well with the data set.

The first code example shown in Listing 6-1 tunes data with SGDClassifier.

Listing 6-1. Tuning fetch_1fw_people with SGDClassifier

```
import numpy as np, humanfriendly as hf, warnings
import time
from sklearn.decomposition import PCA
from sklearn.model_selection import train_test_split,\
     GridSearchCV, cross_val_score
from sklearn.linear_model import SGDClassifier
from sklearn.metrics import classification_report

def see_time(note):
    end = time.perf_counter()
    elapsed = end - start
    print (note, hf.format_timespan(elapsed, detailed=True))

def get_cross(model, data, target, groups=10):
    return cross_val_score(model, data, target, cv=groups)
```

```
if __name__ == "__main__":
    br = '\n'
    warnings.filterwarnings("ignore", category=DeprecationWarning)
    X = np.load('data/X_faces.npy')
    y = np.load('data/y_faces.npy')
    X_train, X_test, y_train, y_test = train_test_split(
        X, y, random_state=0)
    pca = PCA(n_components=0.95, whiten=True, random_state=1)
    pca.fit(X_train)
    X_train_pca = pca.transform(X_train)
    X_test_pca = pca.transform(X_test)
    pca_name = pca.__class__.__name__
    print ('<<' + pca_name + '>>')
    print ('features (before PCA):', X.shape[1])
    print ('features (after PCA):', pca.n_components_, br)
    sgd = SGDClassifier(max_iter=1000, tol=.001, random_state=0)
    sgd.fit(X_train_pca, y_train)
    y_pred = sgd.predict(X_test_pca)
    cr = classification_report(y_test, y_pred)
    print (cr)
    sgd_name = sgd.__class__.__name__
    param_grid = {'alpha': [1e-3, 1e-2, 1e-1, 1e0], 'max_iter': [1000],
                  'loss': ['log', 'perceptron'], 'penalty': ['l1'],
                  'tol': [.001]}
    grid = GridSearchCV(sgd, param_grid, cv=5)
    start = time.perf_counter()
    grid.fit(X_train_pca, y_train)
    see_time('training time:')
    print ()
    bp = grid.best_params_
    print ('best parameters:')
    print (bp, br)
    sgd = SGDClassifier(**bp, random_state=1)
```

```
    sgd.fit(X_train_pca, y_train)
    y_pred = sgd.predict(X_test_pca)
    cr = classification_report(y_test, y_pred)
    print (cr)
    print ('cross-validation:')
    scores = get_cross(sgd, X_train_pca, y_train)
    print (np.mean(scores))
```

Go ahead and execute the code from Listing 6-1. 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 6-1 should resemble the following:

```
<<PCA>>
features (before PCA): 1850
features (after PCA): 135

              precision    recall  f1-score   support

           0       0.89      0.57      0.70        28
           1       0.80      0.78      0.79        63
           2       0.83      0.62      0.71        24
           3       0.73      0.89      0.80       132
           4       0.55      0.55      0.55        20
           5       0.88      0.32      0.47        22
           6       0.67      0.73      0.70        33

   micro avg       0.74      0.74      0.74       322
   macro avg       0.76      0.64      0.67       322
weighted avg       0.76      0.74      0.73       322

training time: 7 seconds and 745.7 milliseconds

best parameters:
{'alpha': 0.001, 'loss': 'log', 'max_iter': 1000, 'penalty': 'l1', 'tol': 0.001}

              precision    recall  f1-score   support

           0       0.91      0.71      0.80        28
           1       0.79      0.79      0.79        63
```

```
              2       0.71      0.71      0.71        24
              3       0.84      0.86      0.85       132
              4       0.48      0.75      0.59        20
              5       0.83      0.45      0.59        22
              6       0.72      0.79      0.75        33

   micro avg          0.78      0.78      0.78       322
   macro avg          0.76      0.72      0.73       322
weighted avg          0.79      0.78      0.78       322

cross-validation:
0.7808966616425951
```

The first code example begins by importing requisite packages. Function see_time returns elapased time. The main block loads data into X and y, splits it into train-test subsets, and conducts PCA to reduce feature space dimensionality.

PCA is critical when tuning high-dimensional data because it *drastically* reduces computational expense with minimal information loss. The code then trains data with SGDClassifier (to obtain a baseline performance measure) and displays results. Next, tuning commences with GridSearchCV.

Tip PCA is a critical tuning tool because it reduces dimensionality on high-dimensional data sets with minimal information loss, which results in drastically lower tuning time (or less computational expense).

We tune *alpha*, *max_iter*, *loss*, *penalty*, and *tol* hyperparameters. Hyperparameter *alpha* is the constant that multiplies the regularization term. Hyperparameter *max_iter* sets the maximum number of passes (or epochs) over training data. An *epoch* is one complete presentation of the data set to be learned by a machine.

Hyperparameter *loss* refers to the loss function used for the experiment. Machines learn by means of a *loss function*, which is a method for evaluating how well an algorithm models a given set of data. Hyperparameter *penalty* refers to the regularization term that is used by the model. Hyperparameter *tol* is the stopping criteria.

The two *most important hyperparameters* are alpha and penalty as they are directly related to the type and amount of regularization employed by the model.

The parameter grid is constructed next. Notice that alpha is the critical hyperparameter adjusted in this experiment. Through trial-and-error experiments, I determined that *l1* penalty was the best option, so I hard-coded it into the grid to reduce tuning time. Once tuned, SGDClassifier trains on the data with the best parameters and displays results. Finally, cross-validation is conducted to ensure that the model is performing at its best (which it is).

Tip It is much easier (and faster) to conduct tuning experiments by varying one or two hyperparameters at a time and keeping the others constant by hard-coding their values.

The second code example shown in Listing 6-2 tunes with svm.SVC. From experience, I knew that svm.SVC outperformed SGDClassifier, but I wanted to demonstrate at least some of the rigor inherent in the experimental process of tuning by including the first code example in the chapter.

Listing 6-2. Tuning fetch_1fw_people with svm.SVC

```
import numpy as np, humanfriendly as hf
import time
from sklearn.decomposition import PCA
from sklearn.model_selection import train_test_split,\
     GridSearchCV, cross_val_score
from sklearn.svm import SVC
from sklearn.metrics import classification_report
import matplotlib.pyplot as plt

def see_time(note):
    end = time.perf_counter()
    elapsed = end - start
    print (note, hf.format_timespan(elapsed, detailed=True))

def get_cross(model, data, target, groups=10):
    return cross_val_score(model, data, target, cv=groups)
```

```
if __name__ == "__main__":
    br = '\n'
    X = np.load('data/X_faces.npy')
    y = np.load('data/y_faces.npy')
    images = np.load('data/faces_images.npy')
    targets = np.load('data/faces_targets.npy')
    _, h, w = images.shape
    n_images, n_features, n_classes = X.shape[0], X.shape[1],\
                                      len(targets)
    X_train, X_test, y_train, y_test = train_test_split(
        X, y, random_state=0)
    pca = PCA(n_components=0.95, whiten=True, random_state=0)
    pca.fit(X_train)
    components = pca.n_components_
    eigenfaces = pca.components_.reshape((components, h, w))
    X_train_pca = pca.transform(X_train)
    pca_name = pca.__class__.__name__
    print ('<<' + pca_name + '>>')
    print ('features (before PCA):', n_features)
    print ('features (after PCA):', components, br)
    X_i = np.array(eigenfaces[0].reshape(h, w))
    fig = plt.figure('eigenface')
    ax = fig.subplots()
    image = ax.imshow(X_i, cmap='bone')
    svm = SVC(random_state=0, gamma='scale')
    print (svm, br)
    svm.fit(X_train_pca, y_train)
    X_test_pca = pca.transform(X_test)
    y_pred = svm.predict(X_test_pca)
    cr = classification_report(y_test, y_pred)
    print (cr)
    svm_name = svm.__class__.__name__
    param_grid = {'C': [1e2, 1e3, 5e3], 'gamma': [0.001, 0.005, 0.01, 0.1],
                  'kernel': ['rbf'], 'class_weight': ['balanced']}
    grid = GridSearchCV(svm, param_grid, cv=5)
```

```
    start = time.perf_counter()
    grid.fit(X_train_pca, y_train)
    see_time('training time:')
    print ()
    bp = grid.best_params_
    print ('best parameters:')
    print (bp, br)
    svm = SVC(**bp)
    svm.fit(X_train_pca, y_train)
    y_pred = svm.predict(X_test_pca)
    print ()
    cr = classification_report(y_test, y_pred)
    print (cr, br)
    print ('cross-validation:')
    scores = get_cross(svm, X_train_pca, y_train)
    print (np.mean(scores), br)
    file = 'data/bp_face'
    np.save(file, bp)
    bp = np.load('data/bp_face.npy')
    bp = bp.tolist()
    print ('best parameters:')
    print (bp)
    plt.show()
```

Your output from executing Listing 6-2 should resemble the following:

```
<<PCA>>
features (before PCA): 1850
features (after PCA): 135

SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape='ovr', degree=3, gamma='scale',
  kernel='rbf', max_iter=-1, probability=False, random_state=0,
  shrinking=True, tol=0.001, verbose=False)
```

```
              precision    recall  f1-score   support

           0       1.00      0.43      0.60        28
           1       0.83      0.87      0.85        63
           2       0.94      0.62      0.75        24
           3       0.71      0.97      0.82       132
           4       1.00      0.70      0.82        20
           5       1.00      0.36      0.53        22
           6       0.96      0.73      0.83        33

   micro avg       0.80      0.80      0.80       322
   macro avg       0.92      0.67      0.74       322
weighted avg       0.84      0.80      0.78       322

training time: 18 seconds and 143.89 milliseconds

best parameters:
{'C': 100.0, 'class_weight': 'balanced', 'gamma': 0.005, 'kernel': 'rbf'}

              precision    recall  f1-score   support

           0       1.00      0.64      0.78        28
           1       0.76      0.92      0.83        63
           2       0.91      0.88      0.89        24
           3       0.88      0.92      0.90       132
           4       0.74      0.85      0.79        20
           5       1.00      0.64      0.78        22
           6       0.90      0.85      0.88        33

   micro avg       0.86      0.86      0.86       322
   macro avg       0.89      0.81      0.84       322
weighted avg       0.87      0.86      0.86       322

cross-validation:
0.8393624737627647

best parameters:
{'C': 100.0, 'class_weight': 'balanced', 'gamma': 0.005, 'kernel': 'rbf'}
```

Listing 6-2 also displays Figure 6-1, which is the first eigenface created by PCA.

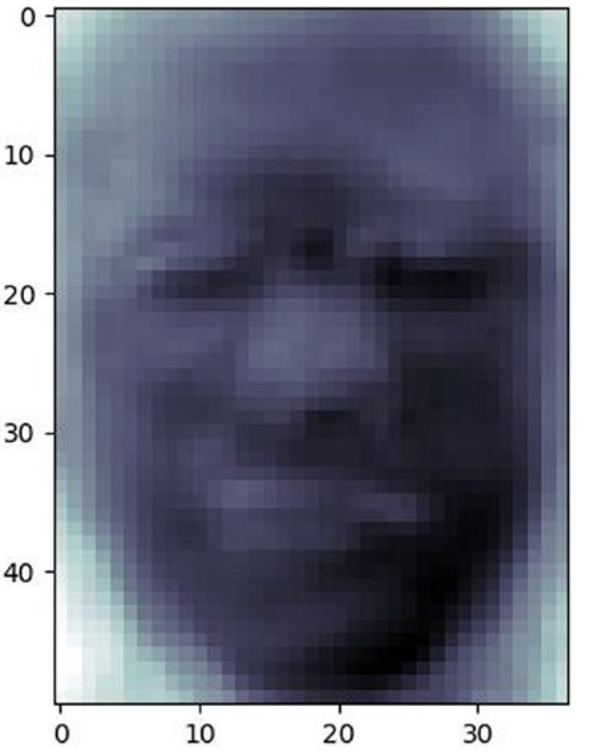

Figure 6-1. *First eigenface created by PCA*

The code begins by importing requisite packages. Function see_time returns elapsed time. The main block loads data into X and y, splits it into train-test subsets, and conducts PCA for dimensionality reduction. Baseline performance for svm.SVC is displayed for later comparison to the tuned svm.SVC score.

Tuning commences by constructing a grid with *C*, *gamma*, *kernel*, and *class_weight* hyperparameters. Hyperparameter *C* is the penalty parameter of the error term, so it is very important for tuning. Hyperparameter *gamma* is the kernel coefficient. Hyperparameter *kernel* specifies the kernel type to be used by the algorithm (e.g., linear). Hyperparameter *class_weight* is used to set the weight (or emphasis) of each class. Through experimentation, I found that the *rbf* kernel and *balanced* class weight were the best, so I hard-coded them into the grid.

My process of discovery is as follows: First, I kept all other hyperparameters constant and changed kernel to see the setting that yielded the best performance. Second, I kept kernel constant and changed class weight.

As you can tell by the grid, we vary C and gamma to improve performance. Once best parameters are determined, svm.SVC trains the data with them. Results are displayed along with cross-validation measures. We have done well with svm.SVC since we performed significantly better than the cross-validation score. We display the first eigenface from dimensionality reduction for completeness. Finally, best parameters are saved (and displayed).

Tuning MNIST

MNIST is not a large data set with 70000 examples, but it has a high-dimensional feature space consisting of 784 features. Such feature space complexity increases computational expense, so we must take this into account when running experiments with computationally expensive algorithms like svm.SVC.

The first code example in Listing 6-3 tunes MNIST with RandomForestClassifier and ExtraTreesClassifier. These algorithms have numerous hyperparameters, but we only adjust a few. I was able to greatly simplify tuning from my experience with these algorithms. You can experiment further, but computational expense increases greatly as you adjust additional hyperparameters.

Listing 6-3. Tuning with RandomForestClassifier and ExtraTreesClassifier

```
import numpy as np, humanfriendly as hf, random
import time
from sklearn.model_selection import train_test_split
from sklearn.model_selection import RandomizedSearchCV,\
     cross_val_score
from sklearn.ensemble import RandomForestClassifier,\
     ExtraTreesClassifier

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

```
    name = model.__class__.__name__
    return (name, train, test)

def get_cross(model, data, target, groups=10):
    return cross_val_score(model, data, target, cv=groups)

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)

def see_time(note):
    end = time.perf_counter()
    elapsed = end - start
    print (note, hf.format_timespan(elapsed, detailed=True))

if __name__ == "__main__":
    br = '\n'
    X_file = 'data/X_mnist'
    y_file = 'data/y_mnist'
    X = np.load('data/X_mnist.npy')
    y = np.load('data/y_mnist.npy')
    X = X.astype(np.float32)
    data = prep_data(X, y)
    sample_size = 7000
    Xs, ys = create_sample(data, sample_size)
    rf = RandomForestClassifier(random_state=0, n_estimators=100)
    print (rf, br)
    params = {'class_weight': ['balanced'], 'max_depth': [10, 30]}
    random = RandomizedSearchCV(rf, param_distributions = params,
                                cv=3, n_iter=2, random_state=0)
```

```
start = time.perf_counter()
random.fit(Xs, ys)
see_time('RandomizedSearchCV total tuning time:')
bp = random.best_params_
print (bp, br)
X_train, X_test, y_train, y_test = train_test_split(
    X, y, random_state=0)
rf = RandomForestClassifier(**bp, random_state=0, n_estimators=100)
start = time.perf_counter()
rf.fit(X_train, y_train)
rf_scores = get_scores(rf, X_train, y_train, X_test, y_test)
see_time('total time:')
print (rf_scores[0] + ' (train, test):')
print (rf_scores[1], rf_scores[2], br)
et = ExtraTreesClassifier(random_state=0, n_estimators=200)
print (et, br)
params = {'class_weight': ['balanced'], 'max_depth': [10, 30]}
random = RandomizedSearchCV(et, param_distributions = params,
                            cv=3, n_iter=2, random_state=0)
start = time.perf_counter()
random.fit(Xs, ys)
see_time('RandomizedSearchCV total tuning time:')
bp = random.best_params_
print (bp, br)
X_train, X_test, y_train, y_test = train_test_split(
    X, y, random_state=0)
et = ExtraTreesClassifier(**bp, random_state=0, n_estimators=200)
start = time.perf_counter()
et.fit(X_train, y_train)
et_scores = get_scores(et, X_train, y_train, X_test, y_test)
see_time('total time:')
print (et_scores[0] + ' (train, test):')
print (et_scores[1], et_scores[2], br)
print ('cross-validation (et):')
start = time.perf_counter()
```

```
    scores = get_cross(rf, X, y)
    see_time('total time:')
    print (np.mean(scores), br)
    file = 'data/bp_mnist_et'
    np.save(file, bp)
    bp = np.load('data/bp_mnist_et.npy')
    bp = bp.tolist()
    print ('best parameters:')
    print (bp)
```

Your output from executing Listing 6-3 should resemble the following:

```
RandomForestClassifier(bootstrap=True, class_weight=None,
            criterion='gini', max_depth=None,
            max_features='auto', max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf='deprecated', min_samples_split=2,
            min_weight_fraction_leaf='deprecated',
            n_estimators=100, n_jobs=None, oob_score=False,
            random_state=0, verbose=0, warm_start=False)

RandomizedSearchCV total tuning time: 13 seconds and 398.73 milliseconds
{'max_depth': 30, 'class_weight': 'balanced'}

total time: 32 seconds and 589.23 milliseconds
RandomForestClassifier (train, test):
0.9999809523809524 0.9701142857142857

ExtraTreesClassifier(bootstrap=False, class_weight=None,
           criterion='gini', max_depth=None, max_features='auto',
           max_leaf_nodes=None, min_impurity_decrease=0.0,
           min_impurity_split=None,
           min_samples_leaf='deprecated', min_samples_split=2,
           min_weight_fraction_leaf='deprecated',
           n_estimators=200, n_jobs=None, oob_score=False,
           random_state=0, verbose=0, warm_start=False)

RandomizedSearchCV total tuning time: 23 seconds and 342.93 milliseconds
{'max_depth': 30, 'class_weight': 'balanced'}
```

```
total time: 1 minute, 8 seconds and 270.59 milliseconds
ExtraTreesClassifier (train, test):
1.0 0.9732

cross-validation (et):
total time: 5 minutes, 40 seconds and 788.07 milliseconds
0.9692001937716965

best parameters:
{'max_depth': 30, 'class_weight': 'balanced'}
```

The code begins by importing requisite packages. Function get_scores returns accuracy scores and model name. Function get_cross returns cross-validation score. Function prep_data prepares data for function create_sample. Function create sample creates a random sample with or without replacement. Function see_time returns elapsed time. The main block loads data, creates a random sample, and instantiates algorithm RandomForestClassifier.

Tuning commences by constructing a grid with *class_weight* and *max_depth* hyperparameters. Hyperparameter *class_weight* is used to set the weight (or emphasis) of each class. Hyperparameter *max_depth* is used to establish the maximum depth of the tree. Through many hours of experimentation, I found that these two parameters were key to increasing performance. Tuning continues by leveraging RandomizedSearchCV to obtain the best parameters. Notice that tuning time is only a bit over thirteen seconds because the grid is very simple.

Now we can test RandomForestClassifier with best parameters. Notice that we include hyperparameter *n_estimators* in the algorithm along with best parameters. Hyperparameter *n_estimators* represents the number of trees in the forest and may be the most important hyperparameter for improving performance.

We include n_estimators in the algorithm (instead of putting it in the grid) for two reasons. First, it is such an important hyperparameter that we can save time by adjusting it outside a tuning experiment. That is, we can adjust it very easily without adding computational expense to the tuning experiment. However, increasing its value does add computational expense to processing the algorithm. Second, it must be included with this algorithm to avoid an annoying warning.

Tuning ExtraTreesClassifier follows the exact same logic with only one difference. We increase n_estimators to 200 trees. Notice that this increase causes processing time to more than double, but performance is better.

Finally, we run cross-validation (on ExtraTreesClassifier) and save the best parameters from ExtraTreesClassifier for future processing. From the cross-validation score, we know that our accuracy scores are solid. However, cross-validation consumes over 5 minutes of processing time! *You can comment out the cross-validation part of the code if you don't want to wait.*

On a positive note, cross-validation only needs to be executed once on an algorithm. I suggest that you run cross-validation before commencing a tuning experiment. You can then run trial-and-error experiments until you meet or exceed the cross-validation score.

Tip Cross-validation need only be run once because it cannot be tuned.

Overall performance was good with accuracy over 97% with not too much overfitting. But, don't be lulled into a false sense of security by working through my tuning experiments. Tuning consumes a lot of time and patience. I can only give you examples and hints to help you become a more accomplished data scientist.

I highly recommend timing tuning experiments, especially ones that are computationally expensive (such as tuning with numerous hyperparameters over various ranges of values). Otherwise, it is very difficult to get a sense of how well your experiment is proceeding. When I first began tuning machine learning algorithms, I didn't time experiments. My progress was slow because I became very frustrated when I couldn't differentiate tuning experiments by elapsed time.

Tip Always time tuning experiments to gauge progress.

The next code example shown in Listing 6-4 tunes MNIST with svm.SVC.

Listing 6-4. Tuning MNIST with svm.SVC

```
import numpy as np, humanfriendly as hf, random
import time
from sklearn.model_selection import train_test_split
from sklearn.model_selection import RandomizedSearchCV
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC
```

```
def get_scores(model, xtrain, ytrain, xtest, ytest):
    ypred = model.predict(xtest)
    train = model.score(xtrain, ytrain)
    test = model.score(xtest, y_test)
    name = model.__class__.__name__
    return (name, train, test)

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)

def see_time(note):
    end = time.perf_counter()
    elapsed = end - start
    print (note, hf.format_timespan(elapsed, detailed=True))

if __name__ == "__main__":
    br = '\n'
    X_file = 'data/X_mnist'
    y_file = 'data/y_mnist'
    X = np.load('data/X_mnist.npy')
    y = np.load('data/y_mnist.npy')
    X = X.astype(np.float32)
    data = prep_data(X, y)
    sample_size = 7000
    Xs, ys = create_sample(data, sample_size)
    pca = PCA(n_components=0.95, random_state=0)
    Xs = StandardScaler().fit_transform(Xs)
    Xs_reduced = pca.fit_transform(Xs)
```

```
    X_train, X_test, y_train, y_test = train_test_split(
        Xs_reduced, ys, random_state=0)
    svm = SVC(gamma='scale', random_state=0)
    print (svm, br)
    start = time.perf_counter()
    svm.fit(X_train, y_train)
    svm_scores = get_scores(svm, X_train, y_train, X_test, y_test)
    print (svm_scores[0] + ' (train, test):')
    print (svm_scores[1], svm_scores[2])
    see_time('time:')
    print ()
    param_grid = {'C': [30, 35, 40], 'kernel': ['poly'],
                  'gamma': ['scale'], 'degree': [3], 'coef0': [0.1]}
    start = time.perf_counter()
    rand = RandomizedSearchCV(svm, param_grid, cv=3, n_jobs = -1,
                              random_state=0, n_iter=3, verbose=2)
    rand.fit(X_train, y_train)
    see_time('RandomizedSearchCV total tuning time:')
    bp = rand.best_params_
    print (bp, br)
    svm = SVC(**bp, random_state=0)
    start = time.perf_counter()
    svm.fit(X_train, y_train)
    svm_scores = get_scores(svm, X_train, y_train, X_test, y_test)
    print (svm_scores[0] + ' (train, test):')
    print (svm_scores[1], svm_scores[2])
    see_time('total time:')
```

Your output from executing Listing 6-4 should resemble the following:

```
SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape='ovr', degree=3, gamma='scale',
  kernel='rbf', max_iter=-1, probability=False, random_state=0,
  shrinking=True, tol=0.001, verbose=False)
```

```
SVC (train, test):
0.9845714285714285 0.9228571428571428
time: 13 seconds and 129.03 milliseconds

Fitting 3 folds for each of 3 candidates, totalling 9 fits
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done   4 out of   9 | elapsed:   14.0s
remaining:   17.6s
[Parallel(n_jobs=-1)]: Done   9 out of   9 | elapsed:   19.3s remaining:
0.0s
[Parallel(n_jobs=-1)]: Done   9 out of   9 | elapsed:   19.3s finished
RandomizedSearchCV total tuning time: 23 seconds and 824.72 milliseconds
{'kernel': 'poly', 'gamma': 'scale', 'degree': 3, 'coef0': 0.1, 'C': 30}

SVC (train, test):
1.0 0.9542857142857143
total time: 10 seconds and 810.06 milliseconds
```

Like the first MNIST tuning code example, we take a random sample. But, we also use PCA for dimensionality reduction because of the immense computational expense inherent with svm.SVC.

Tip For computationally expensive algorithms, we recommend drawing a random sample *and* using PCA for dimensionality reduction to speed processing.

The code begins by importing requisite packages. We already talked about the functions in the last example, so we don't need to discuss it here.

The main block loads data and draws a random sample of 7000. PCA is used for dimensionality reduction with 5% information loss. Next, we scale training data because svm.SVC responds well to scaling. The code continues by splitting data into train-test subsets. Next, svm.SVC is trained with default parameters to gauge performance.

The code continues using RandomizedSearchCV to tune. We create a grid with hyperparameters *C*, *kernel*, *gamma*, *degree*, and *coef0*. We've already discussed hyperparameters C, kernel, and gamma, so we don't need to do it again here. Hyperparameter *degree* represents the degree of the polynomial kernel function. We include it because we chose *poly* for the kernel. Hyperparameter *coef0* is used in conjunction with *degree* for polynomial kernels.

Through experimentation, I found that hyperparameter C was the most important one to adjust. So, the grid only varies the values for C.

The code continues by using the best parameters from the tuning experiment with svm.SVC. We were able to increase test performance by quite a bit, but we still face overfitting.

We didn't include cross-validation for two reasons. First, svm.SVC didn't perform as well as ExtraTreeClassifier (so what's the point?). Second, it takes an extraordinary amount of time to run cross-validation on svm.SVC with MNIST.

Tuning fetch_20newsgroups

Like face recognition, text exploration is a *very* complex topic in machine learning. But, Scikit-Learn provides fetch_20newsgroups that is a wonderful data set upon which to experiment and learn.

Tuning complexity is greatly exacerbated because a pipelined model (with MultinomialNB and TfidfVectorizer) includes two sets of hyperparmeters (one from each algorithm).

Tuning MultinomialNB by itself is very easy because one need only adjust the *alpha* hyperparameter. Hyperparameter *alpha* allows us to adjust smoothing. However, tuning TfidfVectorizer is much more difficult as it includes numerous hyperparameters.

We encounter an even higher level of difficulty when tuning a pipelined model with RandomizedSearchCV because the names of the hyperparameters are different. Each hyperparameter from a pipelined model must be prefixed with the algorithm name so that RandomizedSearchCV can interpret correctly. This makes sense because algorithms can share the same hyperparameters.

The code example shown in Listing 6-5 tunes a pipelined model.

Listing 6-5. Tuning fetch_20newsgroups with a pipelined model

```
import numpy as np, humanfriendly as hf
import time
from sklearn.datasets import fetch_20newsgroups
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.pipeline import make_pipeline
from sklearn.metrics import f1_score
```

```
from sklearn.model_selection import RandomizedSearchCV,\
     cross_val_score

def get_cross(model, data, target, groups=10):
    return cross_val_score(model, data, target, cv=groups)

def see_time(note):
    end = time.perf_counter()
    elapsed = end - start
    print (note, hf.format_timespan(elapsed, detailed=True))

if __name__ == "__main__":
    br = '\n'
    train = fetch_20newsgroups(subset='train')
    test = fetch_20newsgroups(subset='test')
    categories = ['rec.autos', 'rec.motorcycles', 'sci.space', 'sci.med']
    train = fetch_20newsgroups(subset='train', categories=categories,
                               remove=('headers', 'footers', 'quotes'))
    test = fetch_20newsgroups(subset='test', categories=categories,
                              remove=('headers', 'footers', 'quotes'))
    targets = train.target_names
    mnb = MultinomialNB()
    tf = TfidfVectorizer()
    print (mnb, br)
    print (tf, br)
    pipe = make_pipeline(tf, mnb)
    pipe.fit(train.data, train.target)
    labels = pipe.predict(test.data)
    f1 = f1_score(test.target, labels, average='micro')
    print ('f1_score', f1, br)
    print (pipe.get_params().keys(), br)
    param_grid = {'tfidfvectorizer__ngram_range': [(1, 1), (1, 2)],
                  'tfidfvectorizer__use_idf': [True, False],
                  'multinomialnb__alpha': [1e-2, 1e-3],
                  'multinomialnb__fit_prior': [True, False]}
    start = time.perf_counter()
```

```
    rand = RandomizedSearchCV(pipe, param_grid, cv=3, n_jobs = -1,
                              random_state=0, n_iter=16, verbose=2)
    rand.fit(train.data, train.target)
    see_time('RandomizedSearchCV tuning time:')
    bp = rand.best_params_
    print ()
    print ('best parameters:')
    print (bp, br)
    rbs = rand.best_score_
    mnb = MultinomialNB(alpha=0.01)
    tf = TfidfVectorizer(ngram_range=(1, 1), use_idf=False)
    pipe = make_pipeline(tf, mnb)
    pipe.fit(train.data, train.target)
    labels = pipe.predict(test.data)
    f1 = f1_score(test.target, labels, average='micro')
    print ('f1_score', f1, br)
    file = 'data/bp_news'
    np.save(file, bp)
    bp = np.load('data/bp_news.npy')
    bp = bp.tolist()
    print ('best parameters:')
    print (bp, br)
    start = time.perf_counter()
    scores = get_cross(pipe, train.data, train.target)
    see_time('cross-validation:')
    print (np.mean(scores))
```

Your output from executing Listing 6-5 should resemble the following:

```
MultinomialNB(alpha=1.0, class_prior=None, fit_prior=True)

TfidfVectorizer(analyzer='word', binary=False,
        decode_error='strict', dtype=<class 'numpy.float64'>,
        encoding='utf-8', input='content', lowercase=True,
        max_df=1.0, max_features=None, min_df=1,
        ngram_range=(1, 1), norm='l2', preprocessor=None,
```

```
        smooth_idf=True, stop_words=None, strip_accents=None,
        sublinear_tf=False, token_pattern='(?u)\\b\\w\\w+\\b',
        tokenizer=None, use_idf=True, vocabulary=None)

f1_score 0.8440656565656567

dict_keys(['memory', 'steps', 'tfidfvectorizer', 'multinomialnb',
'tfidfvectorizer__analyzer', 'tfidfvectorizer__binary',
'tfidfvectorizer__decode_error', 'tfidfvectorizer__dtype',
'tfidfvectorizer__encoding', 'tfidfvectorizer__input',
'tfidfvectorizer__lowercase', 'tfidfvectorizer__max_df',
'tfidfvectorizer__max_features', 'tfidfvectorizer__min_df',
'tfidfvectorizer__ngram_range', 'tfidfvectorizer__norm',
'tfidfvectorizer__preprocessor', 'tfidfvectorizer__smooth_idf',
'tfidfvectorizer__stop_words', 'tfidfvectorizer__strip_accents',
'tfidfvectorizer__sublinear_tf', 'tfidfvectorizer__token_pattern',
'tfidfvectorizer__tokenizer', 'tfidfvectorizer__use_idf',
'tfidfvectorizer__vocabulary', 'multinomialnb__alpha',
'multinomialnb__class_prior', 'multinomialnb__fit_prior'])

Fitting 3 folds for each of 16 candidates, totalling 48 fits
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  25 tasks      | elapsed:    7.6s
[Parallel(n_jobs=-1)]: Done  48 out of  48 | elapsed:   12.4s finished
RandomizedSearchCV tuning time: 12 seconds and 747.04 milliseconds

best parameters:
{'tfidfvectorizer__use_idf': False, 'tfidfvectorizer__ngram_range': (1, 1),
'multinomialnb__fit_prior': False, 'multinomialnb__alpha': 0.01}

f1_score 0.8611111111111112

best parameters:
{'tfidfvectorizer__use_idf': False, 'tfidfvectorizer__ngram_range': (1, 1),
'multinomialnb__fit_prior': False, 'multinomialnb__alpha': 0.01}

cross-validation: 2 seconds and 750.36 milliseconds
0.8735201157292913
```

The code begins by importing requisite packages. Functions get_cross and see_time are next. The main block begins by creating train and test sets from the fetch_20newsgroups data set. Next, we create subcategories and split data into train-test subsets. The code continues by creating a baseline pipeline model and displaying f1_score for later comparison to the tuned model.

Possible hyperparameters of the pipelined model can be displayed with *pipe.get_params().keys()*. This is an important step because we must include the exact names for RandomizedSearchCV tuning.

Tip You can (and should) display hyperparameters of a pipelined model with *model_name.get_params().keys()*.

The parameter grid is created with *tfidfvectorizer__ngram_range*, *tfidfvectorizer__use_idf*, *multinomialnb__alpha*, and *multinomialnb__fit_prior*.

Hyperparameter *multinomialnb__alpha* is exactly the same as alpha from MultinomialNB. The only difference is that prefix *multinomialnb* is included to inform RandomizedSearchCV the algorithm upon which it belongs. Hyperparameter *multinomialnb__fit_prior* indicates whether or not to learn class prior probabilities.

Hyperparameters *tfidfvectorizer__ngram_range* and *tfidfvectorizer__use_idf* belong to algorithm TfidfVectorizer as indicated by their prefixes. *ngram_range* indicates the upper and lower boundary of the range of n-values for different n-grams to be extracted from the document. *use_idf* enables or disables inverse-document-frequency reweighting.

Tuning commences with RandomizedSearchCV based on the parameter grid values. With tuning, we are able to increase performance to over 86%. However, cross-validation indicates that we can squeeze out a bit more performance from our model.