CHAPTER 7

Scikit-Learn Regression Tuning

Regression predictive modeling (or just *regression*) is the problem of learning the strength of association between independent variables (or features) and *continuous* dependent variables (or outcomes). Tuning regression algorithms is similar to tuning classification algorithms. That is, we adjust a model's hyperparameters until we arrive at an optimal solution.

The difference is that the goal of regression tuning is to reduce root mean squared error (RMSE), while the goal of classification tuning is to maximize accuracy. A benefit of *RMSE* is that units of the error score are the same as the predicted value. While regression predictions can be evaluated using RMSE, classification predictions cannot.

Tip The goal of regression tuning is to minimize RMSE.

Machine learning algorithms chosen for our tuning examples are not a coincidence. I chose them based on many hours of experimentation, reading, and insight. Algorithms that performed best for a given data set were included, and those that performed poorly were not.

For regression experiments in this chapter, we leverage GridSearchCV for tuning.

Tip Tuning with GridSearchCV is suitable for an exhaustive search for the best performing hyperparameters given adequate computing resources. Tuning with RandomizedSearchCV is suitable for a good search or if tuning high-dimensional data.

© David Paper 2020
D. Paper, *Hands-on Scikit-Learn for Machine Learning Applications*,
https://doi.org/10.1007/978-1-4842-5373-1_7

Learning to tune regression algorithms can be accelerated by working through examples with a variety of data sets and regressors. But, I also suggest following a structured process:

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.

Tip Begin tuning with a baseline algorithm (with its default hyperparameters) to establish baseline performance.

Tuning Data Sets

We concentrate on four data sets: tips, boston, and wine (red and white). tips data is composed of food server tips in restaurants and related factors including tip, price of meal, and time of day. boston data is composed of housing prices from various Boston locations. wine data is composed two data sets (red and white) that consist of variants of Portuguese Vinho Verde wine.

Tuning tips

The code example shown in Listing 7-1 calculates RMSE for a variety of regression algorithms based on unscaled and scaled data. Since tips is such a small data set, it is computationally inexpensive to run this type of experiment.

Listing 7-1. Calculating RMSE for tips data with regression algorithms

```
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
from sklearn.ensemble import RandomForestRegressor as rfr,\
     AdaBoostRegressor as ada, GradientBoostingRegressor as gbr
from sklearn.linear_model import LinearRegression as lr,\
     BayesianRidge as bay, Ridge as rr, Lasso as l,\
     LassoLars as ll, ElasticNet as en,\
     ARDRegression as ard, RidgeCV as rcv
from sklearn.svm import SVR
from sklearn.tree import DecisionTreeRegressor as dtr
from sklearn.neighbors import KNeighborsRegressor as knn
from sklearn.preprocessing import StandardScaler

def get_error(model, Xtest, ytest):
    y_pred = model.predict(Xtest)
    return np.sqrt(mean_squared_error(ytest, y_pred)),\
           model.__class__.__name__

if __name__ == "__main__":
    br = '\n'
    X = np.load('data/X_tips.npy')
    y = np.load('data/y_tips.npy')
    X_train, X_test, y_train, y_test = train_test_split(
        X, y, random_state=0)
    regressors = [lr(), bay(), rr(alpha=.5, random_state=0),
                  l(alpha=0.1, random_state=0), ll(), knn(),
                  ard(), rfr(random_state=0, n_estimators=100),
                  SVR(gamma='scale', kernel='rbf'),
                  rcv(fit_intercept=False), en(random_state=0),
                  dtr(random_state=0), ada(random_state=0),
                  gbr(random_state=0)]
    print ('unscaled:', br)
    for reg in regressors:
        reg.fit(X_train, y_train)
        rmse, name = get_error(reg, X_test, y_test)
        name = reg.__class__.__name__
        print (name + '(rmse):', end=' ')
        print (rmse)
    print ()
    print ('scaled:', br)
    scaler = StandardScaler()
    X_train_std = scaler.fit_transform(X_train)
    X_test_std = scaler.fit_transform(X_test)
    for reg in regressors:
        reg.fit(X_train_std, y_train)
        rmse, name = get_error(reg, X_test_std, y_test)
        name = reg.__class__.__name__
        print (name + '(rmse):', end=' ')
        print (rmse)
```

Go ahead and execute the code from Listing 7-1. 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 7-1 should resemble the following:

```
unscaled:

LinearRegression(rmse): 0.9474705746817206
BayesianRidge(rmse): 0.9245282337469829
Ridge(rmse): 0.9471900902779103
Lasso(rmse): 0.9158574785712037
LassoLars(rmse): 1.333812899498391
KNeighborsRegressor(rmse): 1.086204460049883
ARDRegression(rmse): 0.9264801346401996
RandomForestRegressor(rmse): 0.8850975551298138
SVR(rmse): 0.9441992099702836
RidgeCV(rmse): 0.9426372075893412
ElasticNet(rmse): 0.9307377813721578
DecisionTreeRegressor(rmse): 1.2994272932036561
AdaBoostRegressor(rmse): 0.932681302158466
GradientBoostingRegressor(rmse): 0.9112440690311495
```

```
scaled:

LinearRegression(rmse): 0.9007751177881488
BayesianRidge(rmse): 0.9096801291989541
Ridge(rmse): 0.9010890080377257
Lasso(rmse): 0.8785977911833892
LassoLars(rmse): 1.333812899498391
KNeighborsRegressor(rmse): 0.9613578099280607
ARDRegression(rmse): 0.8745960871430548
RandomForestRegressor(rmse): 0.893772251516372
SVR(rmse): 0.9749204385201592
RidgeCV(rmse): 3.1960055364135638
ElasticNet(rmse): 1.1310151423347359
DecisionTreeRegressor(rmse): 1.1835900827021861
AdaBoostRegressor(rmse): 0.986987944835978
GradientBoostingRegressor(rmse): 0.8908489427010696
```

The code begins by importing requisite packages and a variety of regression algorithms. Function get_error returns model name and RMSE. The main block begins by loading preprocessed tips data from NumPy files. Remember that we encoded tips data and saved it for future processing in Chapter 4.

Tip Scikit-Learn allows you to experiment with a variety of algorithms to test performance without requiring contextual knowledge of them.

The code continues by splitting data into train-test subsets. Next, we create a list of regression algorithms. The code continues by training each algorithm on unscaled data and displaying results. The code then scales data, trains each algorithm on scaled data, and displays results.

Scaling data is a very important part of this experiment because many of the algorithms reported lower RMSE results than their unscaled brethren. The best performing algorithms with scaled data are Lasso and ARDRegression.

Tip Scaling can be a very important technique during the tuning process.

So, the experiment was a success! It guided us to two algorithms upon which we can concentrate our tuning efforts.

The next code example shown in Listing 7-2 tunes tips with Lasso.

Listing 7-2. Tuning tips with Lasso

```
import numpy as np, humanfriendly as hf
import time
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.linear_model import Lasso
from sklearn.model_selection import GridSearchCV,\
     cross_val_score
from sklearn.metrics import mean_squared_error

def get_error(model, Xtest, ytest):
    y_pred = model.predict(Xtest)
    return np.sqrt(mean_squared_error(ytest, y_pred)),\
           model.__class__.__name__

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,
                           scoring='neg_mean_squared_error')

if __name__ == "__main__":
    br = '\n'
    X = np.load('data/X_tips.npy')
    y = np.load('data/y_tips.npy')
    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)
    lasso = Lasso(random_state=0, alpha=0.1)
    print (lasso, br)
    lasso.fit(X_train_std, y_train)
    rmse, name = get_error(lasso, X_test_std, y_test)
    print (name + '(rmse):', end=' ')
    print (rmse, br)
    alpha_lasso = [1e-1]
    params = {'alpha': alpha_lasso, 'positive': [True, False],
              'max_iter': [10, 50, 100]}
    grid = GridSearchCV(lasso, params, cv=5, n_jobs=-1, verbose=1)
    start = time.perf_counter()
    grid.fit(X_train, y_train)
    see_time('training time:')
    bp = grid.best_params_
    print (bp, br)
    lasso = Lasso(**bp, random_state=0).fit(X_train_std, y_train)
    rmse, name = get_error(lasso, X_test_std, y_test)
    print (name + '(rmse):', end=' ')
    print (rmse, br)
    start = time.perf_counter()
    scores = get_cross(lasso, X, y)
    see_time('cross-validation rmse:')
    rmse = np.sqrt(np.mean(scores) * -1)
    print (rmse)
```

Your output from executing Listing 7-2 should resemble the following:

```
Lasso(alpha=0.1, copy_X=True, fit_intercept=True, max_iter=1000,
   normalize=False, positive=False, precompute=False,
   random_state=0, selection='cyclic', tol=0.0001,
   warm_start=False)

Lasso(rmse): 0.8785977911833892
```

```
Fitting 5 folds for each of 6 candidates, totalling 30 fits
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  30 out of  30 | elapsed:    2.1s finished
training time: 2 seconds and 246.86 milliseconds
{'alpha': 0.1, 'max_iter': 10, 'positive': True}

Lasso(rmse): 0.8781319871042923

cross-validation rmse: 8.58 milliseconds
1.0379804468729155
```

The code begins by importing requisite packages. Function get_error returns RMSE. Function see_time returns elapsed time. Function get_cross returns cross_validation RMSE.

The main block begins by loading preprocessed tips data. The code continues by splitting data into train-test subsets. Next, we scale data. We then train data with Lasso and display results for baseline comparison with the tuned RMSE.

Lasso is an algorithm that uses L1 penalty for regularization. We tune *alpha*, *positive*, and *max_iter* hyperparameters based on prior experimentation.

Hyperparameter *alpha* is the constant that multiplies the L1 penalty term. It is also the most important hyperparameter to tune with Lasso. Hyperparameter *positive* forces the coefficient to be positive. Hyperparameter *max_iter* represents the maximum number of iterations.

Tuning commences using GridSearchCV with grid *params*. With tuning, we were able to lower RMSE by a very small amount. Cross-validation reveals that we are doing very well.

Tip Keep in mind that function get_error returns negative mean squared error, so we have to make the result positive by multiplying it by -1 and taking the square root of the result to get RMSE.

The next code example shown in Listing 7-3 tunes tips with ARDRegression.

Listing 7-3. Tuning tips with ARDRegression

```
import numpy as np, humanfriendly as hf
import time
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.linear_model import ARDRegression
from sklearn.model_selection import GridSearchCV,\
     cross_val_score
from sklearn.metrics import mean_squared_error

def get_error(model, Xtest, ytest):
    y_pred = model.predict(Xtest)
    return np.sqrt(mean_squared_error(ytest, y_pred)),\
           model.__class__.__name__

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,
                           scoring='neg_mean_squared_error')

if __name__ == "__main__":
    br = '\n'
    X = np.load('data/X_tips.npy')
    y = np.load('data/y_tips.npy')
    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)
    ard = ARDRegression().fit(X_train_std, y_train)
    print (ard, br)
    rmse, name = get_error(ard, X_test_std, y_test)
    print (name + '(rmse):', end=' ')
    print (rmse, br)
    iters = [50]
    a1 = [1e5, 1e4]
    a2 = [1e5, 1e4]
    params = {'n_iter': iters, 'alpha_1': a1, 'alpha_2': a2}
    grid = GridSearchCV(ard, params, cv=5, n_jobs=-1, verbose=1)
    start = time.perf_counter()
    grid.fit(X_train, y_train)
    see_time('training time:')
    bp = grid.best_params_
    print (bp, br)
    ard = ARDRegression(**bp).fit(X_train_std, y_train)
    rmse, name = get_error(ard, X_test_std, y_test)
    print (name + '(rmse):', end=' ')
    print (rmse, br)
    start = time.perf_counter()
    scores = get_cross(ard, X, y)
    see_time('cross-validation rmse:')
    rmse = np.sqrt(np.mean(scores) * -1)
    print (rmse)
```

Your output from executing Listing 7-3 should resemble the following:

```
ARDRegression(alpha_1=1e-06, alpha_2=1e-06, compute_score=False,
       copy_X=True, fit_intercept=True, lambda_1=1e-06,
       lambda_2=1e-06, n_iter=300, normalize=False,
       threshold_lambda=10000.0, tol=0.001, verbose=False)

ARDRegression(rmse): 0.8745960871430548

Fitting 5 folds for each of 4 candidates, totalling 20 fits
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  20 out of  20 | elapsed:    3.5s finished
training time: 4 seconds and 286.03 milliseconds
{'alpha_1': 10000.0, 'alpha_2': 100000.0, 'n_iter': 50}
```

```
ARDRegression(rmse): 0.8645625277607758

cross-validation rmse: 4 seconds and 10.17 milliseconds
1.0376527153700184
```

The code begins by importing requisite packages. Function get_error returns RMSE. Function see_time returns elapsed time. Function get_cross returns cross_validation RMSE.

The main block begins by loading preprocessed tips data. The code continues by splitting data into train-test subsets. Next, we scale data. We then train data with ARDRegression and display results for baseline comparison with the tuned RMSE.

ARDRegression (Automatic Relevance Determination Regression) fits a regression model with Bayesian Ridge Regression. Estimation of the model is accomplished by iteratively maximizing the marginal log-likelihood of the observations.

We tune with *n_iter*, *alpha_1*, and *alpha_2*. Hyperparameter *n_iter* is the maximum number of iterations. Hyperparameter *alpha_1* is the shape parameter for the gamma distribution prior over the alpha parameter. Hyperparameter *alpha_2* is the inverse scale parameter (or rate parameter) for the gamma distribution prior over the alpha parameter.

We are able to reduce RMSE with tuning. Also, cross-validation reveals that we are doing very well.

Tuning boston

The code example shown in Listing 7-4 calculates RMSE for a variety of regression algorithms based on unscaled and scaled data. Since boston is a relatively small data set, it is computationally inexpensive to run this type of experiment.

Listing 7-4. Calculating RMSE for boston data with regression algorithms

```
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
from sklearn.ensemble import RandomForestRegressor as rfr,\
    AdaBoostRegressor as ada, GradientBoostingRegressor as gbr
from sklearn.linear_model import LinearRegression as lr,\
    BayesianRidge as bay, Ridge as rr, Lasso as l,\
    LassoLars as ll, ElasticNet as en,\
     ARDRegression as ard, RidgeCV as rcv
from sklearn.svm import SVR
from sklearn.tree import DecisionTreeRegressor as dtr
from sklearn.neighbors import KNeighborsRegressor as knn
from sklearn.preprocessing import StandardScaler

def get_error(model, Xtest, ytest):
    y_pred = model.predict(Xtest)
    return np.sqrt(mean_squared_error(ytest, y_pred)),\
           model.__class__.__name__

if __name__ == "__main__":
    br = '\n'
    X = np.load('data/X_boston.npy')
    y = np.load('data/y_boston.npy')
    X_train, X_test, y_train, y_test = train_test_split(
        X, y, random_state=0)
    regressors = [lr(), bay(), rr(alpha=.5, random_state=0),
                  l(alpha=0.1, random_state=0), ll(), knn(),
                  ard(), rfr(random_state=0, n_estimators=100),
                  SVR(gamma='scale', kernel='rbf'),
                  rcv(fit_intercept=False), en(random_state=0),
                  dtr(random_state=0), ada(random_state=0),
                  gbr(random_state=0)]
    print ('unscaled:', br)
    for reg in regressors:
        reg.fit(X_train, y_train)
        rmse, name = get_error(reg, X_test, y_test)
        name = reg.__class__.__name__
        print (name + '(rmse):', end=' ')
        print (rmse)
    print ()
    print ('scaled:', br)
    scaler = StandardScaler()
    X_train_std = scaler.fit_transform(X_train)
    X_test_std = scaler.fit_transform(X_test)
    for reg in regressors:
        reg.fit(X_train_std, y_train)
        rmse, name = get_error(reg, X_test_std, y_test)
        name = reg.__class__.__name__
        print (name + '(rmse):', end=' ')
        print (rmse)
```

Your output from executing Listing 7-4 should resemble the following:

```
unscaled:

LinearRegression(rmse): 4.236710574387242
BayesianRidge(rmse): 4.317939916221959
Ridge(rmse): 4.243658717030716
Lasso(rmse): 4.300740333025026
LassoLars(rmse): 8.754893348840868
KNeighborsRegressor(rmse): 5.9934937623789
ARDRegression(rmse): 4.28415048500826
RandomForestRegressor(rmse): 3.37169151536684
SVR(rmse): 7.100029068343849
RidgeCV(rmse): 4.392246392993031
ElasticNet(rmse): 4.88844846745213
DecisionTreeRegressor(rmse): 4.346328232622458
AdaBoostRegressor(rmse): 3.652816906059683
GradientBoostingRegressor(rmse): 3.1941117128039194

scaled:

LinearRegression(rmse): 4.398269524691269
BayesianRidge(rmse): 4.419543929268475
Ridge(rmse): 4.400075160458176
Lasso(rmse): 4.489952156682322
LassoLars(rmse): 8.754893348840868
KNeighborsRegressor(rmse): 4.757936288305807
ARDRegression(rmse): 4.383622227159
RandomForestRegressor(rmse): 4.053037237125816
SVR(rmse): 5.083294658978756
RidgeCV(rmse): 22.34757636411328
ElasticNet(rmse): 5.277752330669967
```

```
DecisionTreeRegressor(rmse): 5.2796587719252726
AdaBoostRegressor(rmse): 4.100148076529094
GradientBoostingRegressor(rmse): 3.7490071027496015
```

The code begins by importing requisite packages and a variety of regression algorithms. Function get_error returns model name and RMSE. The main block begins by loading cleansed boston data from NumPy files. Remember that we cleansed boston data and saved it for future processing in Chapter 4.

The code continues by splitting data into train-test subsets. Next, we create a list of regression algorithms. The code continues by training each algorithm on unscaled data and displaying results. The code then scales data, trains each algorithm on scaled data, and displays results.

The best performing algorithms in this experiment are GradientBoostingRegressor and RandomForestRegressor (both with unscaled data). So, scaling data did not add value with this data set.

The next code example shown in Listing 7-5 tunes the boston data set with GradientBoostingRegressor.

Listing 7-5. Tuning boston data with GradientBoostingRegressor

```
import numpy as np, humanfriendly as hf, warnings, sys
import time
from sklearn.model_selection import train_test_split
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.model_selection import GridSearchCV,\
     cross_val_score
from sklearn.metrics import mean_squared_error

def get_error(model, Xtest, ytest):
    y_pred = model.predict(Xtest)
    return np.sqrt(mean_squared_error(ytest, y_pred)),\
           model.__class__.__name__

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,
                           scoring='neg_mean_squared_error')

if __name__ == "__main__":
    br = '\n'
    if not sys.warnoptions:
        warnings.simplefilter('ignore')
    X = np.load('data/X_boston.npy')
    y = np.load('data/y_boston.npy')
    X_train, X_test, y_train, y_test = train_test_split(
        X, y, random_state=0)
    gbr = GradientBoostingRegressor(random_state=0)
    print (gbr, br)
    gbr.fit(X_train, y_train)
    rmse, name = get_error(gbr, X_test, y_test)
    print (name + '(rmse):', end=' ')
    print (rmse, br)
    loss = ['ls', 'lad', 'huber']
    lr = [1e-2, 1e-1, 1e-0]
    n_est = [150, 200, 300, 500]
    alpha = [0.9]
    params = {'loss': loss, 'learning_rate': lr,
              'n_estimators': n_est, 'alpha': alpha}
    grid = GridSearchCV(gbr, params, cv=5, n_jobs=-1,
                        verbose=1, refit=False)
    start = time.perf_counter()
    grid.fit(X_train, y_train)
    see_time('training time:')
    bp = grid.best_params_
    print (bp, br)
    gbr = GradientBoostingRegressor(**bp, random_state=0)
    gbr.fit(X_train, y_train)
    rmse, name = get_error(gbr, X_test, y_test)
    print (name + '(rmse):', end=' ')
    print (rmse, br)
    start = time.perf_counter()
    scores = get_cross(gbr, X, y)
    see_time('cross-validation rmse:')
    rmse = np.sqrt(np.mean(scores) * -1)
    print (rmse)
```

Your output from executing Listing 7-5 should resemble the following:

```
GradientBoostingRegressor(alpha=0.9, criterion='friedman_mse',
             init=None, learning_rate=0.1, loss='ls',
             max_depth=3, max_features=None,
             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_iter_no_change=None,
             presort='auto', random_state=0, subsample=1.0,
             tol=0.0001, validation_fraction=0.1, verbose=0,
             warm_start=False)

GradientBoostingRegressor(rmse): 3.1941117128039194

Fitting 5 folds for each of 36 candidates, totalling 180 fits
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  34 tasks      | elapsed:    3.1s
[Parallel(n_jobs=-1)]: Done 180 out of 180 | elapsed:    9.1s finished
training time: 9 seconds and 170.11 milliseconds
{'alpha': 0.9, 'learning_rate': 0.1, 'loss': 'huber', 'n_estimators': 300}

GradientBoostingRegressor(rmse): 3.0839764165411934

cross-validation rmse: 3 seconds and 258.29 milliseconds
3.7929403445012064
```

The code begins by importing GradientBoostingRegressor as well as other requisite packages. *GradientBoostingRegressor* performs gradient boosting for regression by building an additive model in a forward-stage fashion.

Function get_error returns the RMSE and model name for a given algorithm. Function see_time returns elapsed time. Function get_cross returns the negative mean squared error.

The main block loads boston data, splits it into train-test subsets, and trains data with GradientBoostingRegressor. The code continues by displaying RMSE with default parameters to provide a baseline score for comparison to the tuned RMSE. Next, the model is tuned with hyperparameters *loss*, *learning_rate*, *n_estimators*, and *alpha*.

Hyperparameter *loss* is the loss function to be optimized. Hyperparameter *learning_rate* controls how much we adjust model learning with respect to the loss gradient. Hyperparameter *n_estimators* is the number of boosting stages to perform. Hypeparameter *alpha* is the alpha-quantile of the huber loss function.

Tuning enabled a reduction in RMSE. We end by running cross-validation. Since our tuned RMSE is lower than the cross-validation RMSE, we are in good shape.

Tip You may have to occasionally reboot your computer as tuning requires an enormous amount of computing resources.

The final code example in this section (shown in Listing 7-6) tunes the boston data set with RandomForestRegressor.

Listing 7-6. Tuning boston data with RandomForestRegressor

```
import numpy as np, humanfriendly as hf, warnings, sys
import time
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import GridSearchCV,\
     cross_val_score
from sklearn.metrics import mean_squared_error

def get_error(model, Xtest, ytest):
    y_pred = model.predict(Xtest)
    return np.sqrt(mean_squared_error(ytest, y_pred)),\
           model.__class__.__name__
```

```
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,
                           scoring='neg_mean_squared_error')

if __name__ == "__main__":
    br = '\n'
    if not sys.warnoptions:
        warnings.simplefilter('ignore')
    X = np.load('data/X_boston.npy')
    y = np.load('data/y_boston.npy')
    X_train, X_test, y_train, y_test = train_test_split(
        X, y, random_state=0)
    rfr = RandomForestRegressor(random_state=0)
    print (rfr, br)
    rfr.fit(X_train, y_train)
    rmse, name = get_error(rfr, X_test, y_test)
    print (name + '(rmse):', end=' ')
    print (rmse, br)
    n_est = [100, 500, 1000]
    boot = [True, False]
    params = {'n_estimators': n_est, 'bootstrap': boot}
    grid = GridSearchCV(rfr, params, cv=5, n_jobs=-1,
                        verbose=1, refit=False)
    start = time.perf_counter()
    grid.fit(X_train, y_train)
    see_time('training time:')
    bp = grid.best_params_
    print (bp, br)
    rfr = RandomForestRegressor(**bp, random_state=0)
    rfr.fit(X_train, y_train)
    rmse, name = get_error(rfr, X_test, y_test)
    print (name + '(rmse):', end=' ')
    print (rmse, br)
    start = time.perf_counter()
    scores = get_cross(rfr, X, y)
    see_time('cross-validation rmse:')
    rmse = np.sqrt(np.mean(scores) * -1)
    print (rmse)
```

Your output from executing Listing 7-6 should resemble the following:

```
RandomForestRegressor(bootstrap=True, criterion='mse',
           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='warn', n_jobs=None, oob_score=False,
           random_state=0, verbose=0, warm_start=False)

RandomForestRegressor(rmse): 3.5587794792757004

Fitting 5 folds for each of 6 candidates, totalling 30 fits
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  30 out of  30 | elapsed:    8.3s finished
training time: 8 seconds and 453.84 milliseconds
{'bootstrap': True, 'n_estimators': 100}

RandomForestRegressor(rmse): 3.37169151536684

cross-validation rmse: 1 second and 845.76 milliseconds
3.6815463792891623
```

The code begins by importing RandomForestRegressor as well as other requisite packages. *RandomForestRegressor* fits a number of classifying decision trees on various subsamples of the data set and uses averaging to improve predictive accuracy and control overfitting.

Function get_error returns the RMSE and model name for a given algorithm. Function see_time returns elapsed time. Function get_cross returns the negative mean squared error.

The main block loads boston data, splits it into train-test subsets, and trains data with RandomForestRegressor. The code continues by displaying RMSE with default parameters to provide a baseline score for comparison to the tuned RMSE. Next, the model is tuned with hyperparameters *n_estimators* and *bootstrap*.

Hyperparameter *n_estimators* is the number of trees in the forest. Hyperparameter *bootstrap* determines whether bootstrap samples are used when building trees.

Tuning enabled a reduction in RMSE. We end by running cross-validation. Since our tuned RMSE is lower than the cross-validation RMSE, we are in good shape.

Tuning wine

By running an experiment similar to those shown in Listings 7-1 and 7-4, we found that RandomForestRegressor (with unscaled data) delivered the lowest RMSE for both red and white wine data. Go ahead and create your own experiments to verify our results if you wish.

The code example shown in Listing 7-7 tunes the red wine data set with RandomForestRegressor.

Listing 7-7. Tuning red wine data with RandomForestRegressor

```
import numpy as np, humanfriendly as hf
import time
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import GridSearchCV,\
     cross_val_score
from sklearn.metrics import mean_squared_error

def get_error(model, Xtest, ytest):
    y_pred = model.predict(Xtest)
    return np.sqrt(mean_squared_error(ytest, y_pred)),\
           model.__class__.__name__

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,
                           scoring='neg_mean_squared_error')

if __name__ == "__main__":
    br = '\n'
    X = np.load('data/X_red.npy')
    y = np.load('data/y_red.npy')
    X_train, X_test, y_train, y_test = train_test_split(
        X, y, random_state=0)
    rfr = RandomForestRegressor(random_state=0, n_estimators=10)
    print (rfr, br)
    rfr.fit(X_train, y_train)
    rmse, name = get_error(rfr, X_test, y_test)
    print (name + '(rmse):', end=' ')
    print (rmse, br)
    n_est = [100, 500]
    boot = [True, False]
    params = {'n_estimators': n_est, 'bootstrap': boot}
    grid = GridSearchCV(rfr, params, cv=5, n_jobs=-1, verbose=1)
    start = time.perf_counter()
    grid.fit(X_train, y_train)
    see_time('training time:')
    bp = grid.best_params_
    print (bp, br)
    rfr = RandomForestRegressor(**bp, random_state=0)
    rfr.fit(X_train, y_train)
    rmse, name = get_error(rfr, X_test, y_test)
    print (name + '(rmse):', end=' ')
    print (rmse, br)
    start = time.perf_counter()
    scores = get_cross(rfr, X, y)
    see_time('cross-validation rmse:')
    rmse = np.sqrt(np.mean(scores) * -1)
    print (rmse)
```

Your output from executing Listing 7-7 should resemble the following:

```
RandomForestRegressor(bootstrap=True, criterion='mse',
                      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=10, n_jobs=None,
                      oob_score=False, random_state=0, verbose=0,
                      warm_start=False)

RandomForestRegressor(rmse): 0.626079068488957

Fitting 5 folds for each of 4 candidates, totalling 20 fits
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  20 out of  20 | elapsed:    7.1s finished
training time: 7 seconds and 629.56 milliseconds
{'bootstrap': True, 'n_estimators': 100}

RandomForestRegressor(rmse): 0.5847897057917487

cross-validation rmse: 4 seconds and 804.96 milliseconds
0.6498982966515346
```

The code begins by importing requisite packages. Function get_error returns the RMSE and model name for a given algorithm. Function see_time returns elapsed time. Function get_cross returns the negative mean squared error.

The main block loads red wine data, splits it into train-test subsets, and trains data with RandomForestRegressor. The code continues by displaying RMSE with default parameters to provide a baseline score for comparison to the tuned RMSE. Next, the model is tuned with hyperparameters *n_estimators* and *bootstrap*.

Tuning enabled a reduction in RMSE. We end by running cross-validation. Since our tuned RMSE is lower than the cross-validation RMSE, we are in good shape.

The final code example shown in Listing 7-8 tunes the white wine data set with RandomForestRegressor.

Listing 7-8. Tuning white wine data with RandomForestRegressor

```
import numpy as np, humanfriendly as hf
import time
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import GridSearchCV,\
     cross_val_score
from sklearn.metrics import mean_squared_error

def get_error(model, Xtest, ytest):
    y_pred = model.predict(Xtest)
    return np.sqrt(mean_squared_error(ytest, y_pred)),\
           model.__class__.__name__

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,
                           scoring='neg_mean_squared_error')

if __name__ == "__main__":
    br = '\n'
    X = np.load('data/X_white.npy')
    y = np.load('data/y_white.npy')
    X_train, X_test, y_train, y_test = train_test_split(
        X, y, random_state=0)
    rfr = RandomForestRegressor(random_state=0, n_estimators=10)
    print (rfr, br)
    rfr.fit(X_train, y_train)
    rmse, name = get_error(rfr, X_test, y_test)
    print (name + '(rmse):', end=' ')
    print (rmse, br)
    n_est = [100, 500]
    boot = [True, False]
    params = {'n_estimators': n_est, 'bootstrap': boot}
    grid = GridSearchCV(rfr, params, cv=5, n_jobs=-1, verbose=1)
    start = time.perf_counter()
    grid.fit(X_train, y_train)
    see_time('training time:')
    bp = grid.best_params_
    print (bp, br)
    rfr = RandomForestRegressor(**bp, random_state=0)
    rfr.fit(X_train, y_train)
    rmse, name = get_error(rfr, X_test, y_test)
    print (name + '(rmse):', end=' ')
    print (rmse, br)
    start = time.perf_counter()
    scores = get_cross(rfr, X, y)
    see_time('cross-validation rmse:')
    rmse = np.sqrt(np.mean(scores) * -1)
    print (rmse)
```

Your output from executing Listing 7-8 should resemble the following:

```
RandomForestRegressor(bootstrap=True, criterion='mse',
           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=10,
           n_jobs=None, oob_score=False, random_state=0,
           verbose=0, warm_start=False)

RandomForestRegressor(rmse): 0.6966098665124181

Fitting 5 folds for each of 4 candidates, totalling 20 fits
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  20 out of  20 | elapsed:   18.7s finished
training time: 25 seconds and 709.64 milliseconds
{'bootstrap': True, 'n_estimators': 500}
```

```
RandomForestRegressor(rmse): 0.6728175517621279

cross-validation rmse: 1 minute, 24 seconds and 70.99 milliseconds
0.7183073387927801
```

The code begins by importing requisite packages. Function get_error returns the RMSE and model name for a given algorithm. Function see_time returns elapsed time. Function get_cross returns the negative mean squared error.

The main block loads white wine data, splits it into train-test subsets, and trains data with RandomForestRegressor. The code continues by displaying RMSE with default parameters to provide a baseline score for comparison to the tuned RMSE. Next, the model is tuned with hyperparameters *n_estimators* and *bootstrap*.

Tuning enabled a reduction in RMSE. We end by running cross-validation. Since our tuned RMSE is lower than the cross-validation RMSE, we are in good shape.