Boston Housing Prices

Udacity Machine Learning Engineer
Nanodegree Program: Project 1

Anderson Daniel Trimm

January 30, 2016

Abstract

In this report, we present two analyses using the Boston housing dataset: First, we perform a statistical analysis of the dataset using NumPy. Following this analysis, we optimize a decision tree regression algorithm and use it to predict the value of a house using scikit-learn.

1 Introduction

2 Statistical Analysis of the Boston Housing Dataset

In this section we compute basic statistics of the Boston Housing dataset using NumPy. To begin, we need to import the Boston Housing dataset as well as Numpy:

```
import numpy as np
from sklearn import datasets
```

We can now define the following function to load the Boston dataset:

```
def load_data():
    """Load the Boston dataset."""

    boston = datasets.load_boston()
    return boston
```

After loading the Boston dataset, we can look at its attributes `boston.data` and `boston.target` to access the features and housing prices. The attribute `boston.data` gives a two-dimensional ndarray, where each row is the list of features for a given house. The attribute `boston.target` gives a one-dimensional ndarray of the housing prices. The total number of houses is therefore just the length of the ndarray boston.target:

```
>>> np.shape(boston.target)
(506,)
```

which we see is 506. The number of features per house then is the row length of the ndarray boston.data, which is the one-eth entry of

```
>>> np.shape(boston.data)
(506, 13)
```

which is 13. We can encapsulate these in functions as

```
def size_of_data(city_data):
    number_of_houses = np.shape(city_data.data)[0]
    return number_of_houses

def number_of_features(city_data):
    number_of_features = np.shape(city_data.data)[1]
    return number_of_features
```

To compute the minimum, maximum, mean, and meadian price and the standard deviation, we simply use the methods `np.min(boston.target)`,`np.max(boston.target)`, `np.mean(boston.tar` `np.meadian(boston.target)`, and `np.std(boston.target)`, respectively. We find

```
>>> np.min(boston.target)
5.0
>>> np.max(boston.target)
50.0
>>> np.mean(boston.target)
22.532806324110677
>>> np.median(boston.target)
21.199999999999999
>>> np.std(boston.target)
9.1880115452782025
```

As before, we can encapsulate these in functions as

```
def get_min_price(city_data):
    min_price = np.min(city_data.target)
    return min_price

def get_max_price(city_data):
    max_price = np.max(city_data.target)
    return max_price

def get_mean_price(city_data):
    mean_price = np.mean(city_data.target)
    return mean_price
```

```
def get_median_price(city_data):
    median_price = np.median(city_data.target)
    return median_price

def get_standard_deviation(city_data):
    standard_deviation = np.std(city_data.target)
    return standard_deviation
```

3 Predicting Housing Prices

In this section we... ♣ FILL IN SOME INFO ABOUT THE DECISION TREE REGRESSOR

3.1 Evaluating Model Performance

Since we would like to be able to evaluate our model's ability to predict housing prices given new, unseen data, we begin by splitting the Boston dataset into a training and testing set. By holding out the testing data and allowing our model to learn only on the training set, we leave ourselves an independent set of data we can use to verify that our model can generalize well to unseen data. If we trained on the *entire* dataset, we would have no way to evaluate how well our model can predict the housing price of new data points.

To split the dataset, we first import `train_test_split` from the `sklearn.cross_validation` module:

```
from sklearn.cross_validation import train_test_split
```

and define the function ♣ FIX TEXT RUNOFF

```
def split_data(city_data):
    # Get the features and labels from the Boston housing data
    X, y = city_data.data, city_data.target
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.30, random_sta
    return X_train, y_train, X_test, y_test
```

Choosing `test_size=0.30` splits the data into 30 % testing data and 70 % training data, while `random_state=42` is the (pseudo-)random number generator state used for random sampling (which is set arbitrarily to 42).

Next, we will train our model on the training set and compare with its performance on the testing set. To do so, we first need to choose an appropriate performance metric. Looking at the list of regression performance metrics on sklearn, we find four options:

```
mean_absolute_error
mean_squared_error
median_absolute_error
r2
```

Of these, one could argue that the `mean_squared_error` is the most appropriate performance metric for our algorithm, as the sklearn decision tree regressor already by default uses mean squared error to measure the quality of each split (♣ IS THIS REALLY A GOOD REASON?). Additionally, it has the desired properties that larger deviations from the true labels are penalized more heavily as well as being everywhere differentiable, so that one can compute the maximum and minimum errors using calculus, while none of the other choices share all of these properties. In the following, we will therefore use mean squared error as the performance metric for our model (♣ BE MORE SPECIFIC ABOUT WHAT PERFORMANCE METRIC IS ANALYZING).

♣ INSERT DESCRIPTION OF LEARNING CURVES AND MODEL COMPLEXITY GRAPH.

However, we still have a parameter in our model - namely the depth of the decision tree, which we need to optimize. Depending on the value of the parameter, the model may perform better or worse. If we set the parameter too low, the model will suffer from high bias. If it is too high, the model will have high variance. We should therefore cross-validate our model by testing different values of our parameter and comparing the results. We will do this using `GridSearch` over a range of possible values for this parameter. In general, `GridSearch` systematically tries all combinations of parameters, cross-validating as it goes to determine which parameter combination gives the best model performance.

We implement `GridSearch` with the following function (♣ FIX TEXT RUNOFF)

```
from sklearn import grid_search
from sklearn.metrics import mean_squared_error, make_scorer
from sklearn.tree import DecisionTreeRegressor

def fit_predict_model(city_data):
    """Find and tune the optimal model. Make a prediction on housing data."""

    # Get the features and labels from the Boston housing data
    X, y = city_data.data, city_data.target

    # Setup a Decision Tree Regressor
    regressor = DecisionTreeRegressor()

# Parameter set for GridSearch to test
    parameters = {'max_depth':(1,2,3,4,5,6,7,8,9,10)}

    # Score models by mean squared error
    mse_scorer = make_scorer(mean_squared_error, greater_is_better=False)

    # Implement GridSearch
    clf = grid_search.GridSearchCV(regressor, parameters, scoring=mse_scorer)
```

```
    # Fit the learner to the training data to obtain the best parameter set

    clf.fit(city_data.data, city_data.target)

    # Print the result from GridSearch
    print "Final Model: "
    print clf.best_estimator_

    # Use the model to predict the output of a particular sample
    x = [11.95, 0.00, 18.100, 0, 0.6590, 5.6090, 90.00, 1.385, 24, 680.0, 20.20, 332.09,
    y = clf.best_estimator_.predict(x)
    print "House: " + str(x)
    print "Prediction: " + str(y)
```

3.2 Analyzing Model Performance

We can analyze our model's performance visually by plotting a *learning curve*, which plots the error of the training and testing data over the number of training instances. Setting our performance metric as the mean squared error

```
def performance_metric(label, prediction):
    """Calculate and return the appropriate error performance metric."""
    return mean_squared_error(label, prediction)
```

we plot the learning curve for each max depth with the following code:

```
def learning_curve(depth, X_train, y_train, X_test, y_test):
    """Calculate the performance of the model after a set of training data."""

    # We will vary the training set size so that we have 50 different sizes
    sizes = np.round(np.linspace(1, len(X_train), 50))
    train_err = np.zeros(len(sizes))
    test_err = np.zeros(len(sizes))

    print "Decision Tree with Max Depth: "
    print depth

    for i, s in enumerate(sizes):

        # Create and fit the decision tree regressor model
        regressor = DecisionTreeRegressor(max_depth=depth)
        regressor.fit(X_train[:s], y_train[:s])
```

```
        # Find the performance on the training and testing set
        train_err[i] = performance_metric(y_train[:s], regressor.predict(X_train[:s]))
        test_err[i] = performance_metric(y_test, regressor.predict(X_test))


    # Plot learning curve graph
    learning_curve_graph(sizes, train_err, test_err)


def learning_curve_graph(sizes, train_err, test_err):
    """Plot training and test error as a function of the training size."""

    pl.figure()
    pl.title('Decision Trees: Performance vs Training Size')
    pl.plot(sizes, test_err, lw=2, label = 'test error')
    pl.plot(sizes, train_err, lw=2, label = 'training error')
    pl.legend()
    pl.xlabel('Training Size')
    pl.ylabel('Error')
    pl.show()
```

Looking at the resulting learning curves for our example, we see that the training and testing error initially converge at a high value. As we increase the complexity of the model, the training and testing error start to converge at lower values. However, as we keep increasing the complexity, the testing error stays roughly the same (or even becomes slightly worse) while the training error continues to decrease. This is the sign of overfitting. Comparing the first and last learning curves (the learning curves for the decision tree regressor with max depth 1 and 10, respectively) we see that, when fully trained, the first model suffers from high bias (the error is high as the model is not complex enough to accurately model the data) while the last model suffers from high variance (it is too complex - by overfitting the training data, it is not predictive of the testing data).

We can visualize this by plotting a *model complexity graph*, which plots the training and testing error of each fully trained model against the max depth of the decision tree regressor. This is implemented by the following code:

```
def model_complexity(X_train, y_train, X_test, y_test):
    """Calculate the performance of the model as model complexity increases."""

    print "Model Complexity: "

    # We will vary the depth of decision trees from 2 to 25
    max_depth = np.arange(1, 25)
    train_err = np.zeros(len(max_depth))
```

```
    test_err = np.zeros(len(max_depth))

    for i, d in enumerate(max_depth):
        # Setup a Decision Tree Regressor so that it learns a tree with depth d
        regressor = DecisionTreeRegressor(max_depth=d)

        # Fit the learner to the training data
        regressor.fit(X_train, y_train)

        # Find the performance on the training set
        train_err[i] = performance_metric(y_train, regressor.predict(X_train))

        # Find the performance on the testing set
        test_err[i] = performance_metric(y_test, regressor.predict(X_test))

    # Plot the model complexity graph
    model_complexity_graph(max_depth, train_err, test_err)


def model_complexity_graph(max_depth, train_err, test_err):
    """Plot training and test error as a function of the depth of the decision tree lear

    pl.figure()
    pl.title('Decision Trees: Performance vs Max Depth')
    pl.plot(max_depth, test_err, lw=2, label = 'test error')
    pl.plot(max_depth, train_err, lw=2, label = 'training error')
    pl.legend()
    pl.xlabel('Max Depth')
    pl.ylabel('Error')
    pl.show()
```

Looking at the graph, we see confirm that training and testing errors initially decrease together as we increase model complexity, until the testing error levels off and the training error continues to decrease. The region where these two errors diverge is the overfitting region, so we can read off from the graph that a max depth of approximately 4 best generalizes the dataset.