boston_housing

May 11, 2016

1 Machine Learning Engineer Nanodegree

1.1 Model Evaluation & Validation

1.2 Project 1: Predicting Boston Housing Prices

Welcome to the first project of the Machine Learning Engineer Nanodegree! In this notebook, some template code has already been written. You will need to implement additional functionality to successfully answer all of the questions for this project. Unless it is requested, do not modify any of the code that has already been included. In this template code, there are four sections which you must complete to successfully produce a prediction with your model. Each section where you will write code is preceded by a **STEP X** header with comments describing what must be done. Please read the instructions carefully!

In addition to implementing code, there will be questions that you must answer that relate to the project and your implementation. Each section where you will answer a question is preceded by a **QUESTION X** header. Be sure that you have carefully read each question and provide thorough answers in the text boxes that begin with "**Answer:**". Your project submission will be evaluated based on your answers to each of the questions.

A description of the dataset can be found here, which is provided by the **UCI Machine Learning Repository**.

2 Getting Started

To familiarize yourself with an iPython Notebook, **try double clicking on this cell**. You will notice that the text changes so that all the formatting is removed. This allows you to make edits to the block of text you see here. This block of text (and mostly anything that's not code) is written using Markdown, which is a way to format text using headers, links, italics, and many other options! Whether you're editing a Markdown text block or a code block (like the one below), you can use the keyboard shortcut **Shift + Enter** or **Shift + Return** to execute the code or text block. In this case, it will show the formatted text.

Let's start by setting up some code we will need to get the rest of the project up and running. Use the keyboard shortcut mentioned above on the following code block to execute it. Alternatively, depending on your iPython Notebook program, you can press the **Play** button in the hotbar. You'll know the code block executes successfully if the message *"Boston Housing dataset loaded successfully!"* is printed.

```
In [5]: # Importing a few necessary libraries
        import numpy as np
```

```
import matplotlib.pyplot as pl
from sklearn import datasets
from sklearn.tree import DecisionTreeRegressor

# Make matplotlib show our plots inline (nicely formatted in the notebook)
%matplotlib inline

# Create our client's feature set for which we will be predicting a selling
CLIENT_FEATURES = [[11.95, 0.00, 18.100, 0, 0.6590, 5.6090, 90.00, 1.385, 2

# Load the Boston Housing dataset into the city_data variable
city_data = datasets.load_boston()

# Initialize the housing prices and housing features
housing_prices = city_data.target
housing_features = city_data.data

print "Boston Housing dataset loaded successfully!"
```

```
Boston Housing dataset loaded successfully!
```

3   Statistical Analysis and Data Exploration

In this first section of the project, you will quickly investigate a few basic statistics about the dataset
you are working with. In addition, you'll look at the client's feature set in CLIENT_FEATURES and
see how this particular sample relates to the features of the dataset. Familiarizing yourself with
the data through an explorative process is a fundamental practice to help you better understand
your results.

3.1   Step 1

In the code block below, use the imported numpy library to calculate the requested statistics. You
will need to replace each None you find with the appropriate numpy coding for the proper statistic
to be printed. Be sure to execute the code block each time to test if your implementation is working
successfully. The print statements will show the statistics you calculate!

```
In [6]: # Number of houses in the dataset
        total_houses = housing_features.shape[0]

        # Number of features in the dataset
        total_features = housing_features.shape[1]

        # Minimum housing value in the dataset
        minimum_price = np.min(housing_prices)

        # Maximum housing value in the dataset
        maximum_price = np.max(housing_prices)
```

```python
# Mean house value of the dataset
mean_price = np.mean(housing_prices)

# Median house value of the dataset
median_price = np.median(housing_prices)

# Standard deviation of housing values of the dataset
std_dev = np.std(housing_prices)

# Show the calculated statistics
print "Boston Housing dataset statistics (in $1000's):\n"
print "Total number of houses:", total_houses
print "Total number of features:", total_features
print "Minimum house price:", minimum_price
print "Maximum house price:", maximum_price
print "Mean house price: {0:.3f}".format(mean_price)
print "Median house price:", median_price
print "Standard deviation of house price: {0:.3f}".format(std_dev)
```

```
Boston Housing dataset statistics (in $1000's):

Total number of houses: 506
Total number of features: 13
Minimum house price: 5.0
Maximum house price: 50.0
Mean house price: 22.533
Median house price: 21.2
Standard deviation of house price: 9.188
```

3.2 Question 1

As a reminder, you can view a description of the Boston Housing dataset here, where you can find
the different features under **Attribute Information**. The MEDV attribute relates to the values stored
in our housing_prices variable, so we do not consider that a feature of the data.

*Of the features available for each data point, choose three that you feel are significant and give a brief
description for each of what they measure.*

Remember, you can **double click the text box below** to add your answer!

Answer: LSTAT: % lower status of the population, RM: average number of rooms per dwelling,
DIS: weighted distances to five Boston employment centres.

3.3 Question 2

*Using your client's feature set CLIENT_FEATURES, which values correspond with the features you've
chosen above?*

Hint: Run the code block below to see the client's data.

```
In [7]: chosen_features = ['LSTAT', 'RM', 'DIS']
        client_chosen_features = []
        features = city_data.feature_names.tolist()
        for client_features in CLIENT_FEATURES:
            one_client_chosen_features = []
            for feature in chosen_features:
                index = features.index(feature)
                one_client_chosen_features.append(client_features[index])
            client_chosen_features.append(one_client_chosen_features)
        print client_chosen_features

[[12.13, 5.609, 1.385]]
```

Answer: [[12.13, 5.609, 1.385]]

4   Evaluating Model Performance

In this second section of the project, you will begin to develop the tools necessary for a model to make a prediction. Being able to accurately evaluate each model's performance through the use of these tools helps to greatly reinforce the confidence in your predictions.

4.1   Step 2

In the code block below, you will need to implement code so that the `shuffle_split_data` function does the following: - Randomly shuffle the input data X and target labels (housing values) y. - Split the data into training and testing subsets, holding 30% of the data for testing.

   If you use any functions not already acessible from the imported libraries above, remember to include your import statement below as well!

Ensure that you have executed the code block once you are done. You'll know the `shuffle_split_data` function is working if the statement *"Successfully shuffled and split the data!"* is printed.

```
In [8]: # Put any import statements you need for this code block here
        from sklearn.utils import shuffle

        def shuffle_split_data(X, y):
            """ Shuffles and splits data into 70% training and 30% testing subsets,
                then returns the training and testing subsets. """

            # Shuffle and split the data
            X, y = shuffle(X, y, random_state = 13)
            offset = int(X.shape[0] * 0.7)
            X_train = X[:offset]
            y_train = y[:offset]
            X_test = X[offset:]
            y_test = y[offset:]
```

```python
        # Return the training and testing data subsets
        return X_train, y_train, X_test, y_test


    # Test shuffle_split_data
    try:
        X_train, y_train, X_test, y_test = shuffle_split_data(housing_features,
        print "Successfully shuffled and split the data!"
    except:
        print "Something went wrong with shuffling and splitting the data."

Successfully shuffled and split the data!
```

4.2 Question 3

Why do we split the data into training and testing subsets for our model?
    Answer: Testing subsets for model evaluation.

4.3 Step 3

In the code block below, you will need to implement code so that the `performance_metric`
function does the following: - Perform a total error calculation between the true values of the `y`
labels `y_true` and the predicted values of the `y` labels `y_predict`.

    You will need to first choose an appropriate performance metric for this problem. See the
sklearn metrics documentation to view a list of available metric functions. **Hint:** Look at the
question below to see a list of the metrics that were covered in the supporting course for this
project.

    Once you have determined which metric you will use, remember to include the necessary im-
port statement as well!

Ensure that you have executed the code block once you are done. You'll know the
`performance_metric` function is working if the statement *"Successfully performed a metric calcu-
lation!"* is printed.

```python
In [9]: # Put any import statements you need for this code block here
        from sklearn.metrics import mean_squared_error

        def performance_metric(y_true, y_predict):
            """ Calculates and returns the total error between true and predicted
                based on a performance metric chosen by the student. """

            error = mean_squared_error(y_true, y_predict)
            return error


        # Test performance_metric
        try:
            total_error = performance_metric(y_train, y_train)
```

```
            print "Successfully performed a metric calculation!"
        except:
            print "Something went wrong with performing a metric calculation."

Successfully performed a metric calculation!
```

4.4 Question 4

Which performance metric below did you find was most appropriate for predicting housing prices and analyzing the total error. Why? - Accuracy - Precision - Recall - F1 Score - Mean Squared Error (MSE) - Mean Absolute Error (MAE)

Answer: Mean Squared Error (MSE). The four former metrics are for classification performance, the two latter are for regression problem. The MSE is prefer to bigger error of sample prediction than MAE, so the former is more suitable for our predicting housing prices regression problem.

4.5 Step 4 (Final Step)

In the code block below, you will need to implement code so that the `fit_model` function does the following: - Create a scoring function using the same performance metric as in **Step 3**. See the sklearn `make_scorer` documentation. - Build a GridSearchCV object using `regressor`, `parameters`, and `scoring_function`. See the sklearn documentation on GridSearchCV.

When building the scoring function and GridSearchCV object, *be sure that you read the parameters documentation thoroughly.* It is not always the case that a default parameter for a function is the appropriate setting for the problem you are working on.

Since you are using `sklearn` functions, remember to include the necessary import statements below as well!

Ensure that you have executed the code block once you are done. You'll know the `fit_model` function is working if the statement *"Successfully fit a model to the data!"* is printed.

```
In [10]:  # Put any import statements you need for this code block
          from sklearn.metrics import r2_score, make_scorer
          from sklearn.grid_search import GridSearchCV

          def fit_model(X, y):
              """ Tunes a decision tree regressor model using GridSearchCV on the i
                  and target labels y and returns this optimal model. """

              # Create a decision tree regressor object
              regressor = DecisionTreeRegressor()

              # Set up the parameters we wish to tune
              parameters = {'max_depth':(1,2,3,4,5,6,7,8,9,10)}

              # Make an appropriate scoring function
              scoring_function = make_scorer(mean_squared_error, greater_is_better=F
```

```
            # Make the GridSearchCV object
            reg = GridSearchCV(regressor, param_grid=parameters, scoring=scoring_f

            # Fit the learner to the data to obtain the optimal model with tuned p
            reg.fit(X, y)

            # Return the optimal model
            return reg.best_estimator_


        # Test fit_model on entire dataset
        try:
            reg = fit_model(housing_features, housing_prices)
            print "Successfully fit a model!"
        except:
            print "Something went wrong with fitting a model."

Successfully fit a model!
```

4.6 Question 5

What is the grid search algorithm and when is it applicable?

Answer: Grid search algorithm is a search consist of an estimator, a parameter space, a method for searching or sampling candidates, a cross-validation scheme and a score function. It is used for tuning hyperparameters which cannot learn from estimator directly and select the best estimator which has the best performance score.

4.7 Question 6

What is cross-validation, and how is it performed on a model? Why would cross-validation be helpful when using grid search?

Answer: Cross-validation splits the dataset to two parts. The one is training set, and the other is validation set. Model with hyperparameters first fits on the training set, and then performs on the validation set. Explore hyperparameters on a given grid and find the best one which has the best performance on the validation set. Cross-validation with K-Fold makes full use of the dataset as well as avoids overfitting.

5 Checkpoint!

You have now successfully completed your last code implementation section. Pat yourself on the back! All of your functions written above will be executed in the remaining sections below, and questions will be asked about various results for you to analyze. To prepare the **Analysis** and **Prediction** sections, you will need to intialize the two functions below. Remember, there's no need to implement any more code, so sit back and execute the code blocks! Some code comments are provided if you find yourself interested in the functionality.

```
In [11]: def learning_curves(X_train, y_train, X_test, y_test):
             """ Calculates the performance of several models with varying sizes o
                 The learning and testing error rates for each model are then plott

             print "Creating learning curve graphs for max_depths of 1, 3, 6, and 1

             # Create the figure window
             fig = pl.figure(figsize=(10,8))

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

             # Create four different models based on max_depth
             for k, depth in enumerate([1,3,6,10]):

                 for i, s in enumerate(sizes):

                     # Setup a decision tree regressor so that it learns a tree wit
                     regressor = DecisionTreeRegressor(max_depth = depth, random_st

                     # Fit the learner to the training data
                     regressor.fit(X_train[:s], y_train[:s])

                     # Find the performance on the training set
                     train_err[i] = performance_metric(y_train[:s], regressor.predi

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

                 # Subplot the learning curve graph
                 ax = fig.add_subplot(2, 2, k+1)
                 ax.plot(sizes, test_err, lw = 2, label = 'Testing Error')
                 ax.plot(sizes, train_err, lw = 2, label = 'Training Error')
                 ax.legend()
                 ax.set_title('max_depth = %s'%(depth))
                 ax.set_xlabel('Number of Data Points in Training Set')
                 ax.set_ylabel('Total Error')
                 ax.set_xlim([0, len(X_train)])

             # Visual aesthetics
             fig.suptitle('Decision Tree Regressor Learning Performances', fontsize
             fig.tight_layout()
             fig.show()

In [12]: def model_complexity(X_train, y_train, X_test, y_test):
             """ Calculates the performance of the model as model complexity increa
```

8

```python
            The learning and testing errors rates are then plotted. """

    print "Creating a model complexity graph. . . "

    # We will vary the max_depth of a decision tree model from 1 to 14
    max_depth = np.arange(1, 14)
    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 de
        regressor = DecisionTreeRegressor(max_depth = d, random_state=13)

        # 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_tra

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

    # Plot the model complexity graph
    pl.figure(figsize=(7, 5))
    pl.title('Decision Tree Regressor Complexity Performance')
    pl.plot(max_depth, test_err, lw=2, label = 'Testing Error')
    pl.plot(max_depth, train_err, lw=2, label = 'Training Error')
    pl.legend()
    pl.xlabel('Maximum Depth')
    pl.ylabel('Total Error')
    pl.show()
```

6   Analyzing Model Performance

In this third section of the project, you'll take a look at several models' learning and testing error rates on various subsets of training data. Additionally, you'll investigate one particular algorithm with an increasing `max_depth` parameter on the full training set to observe how model complexity affects learning and testing errors. Graphing your model's performance based on varying criteria can be beneficial in the analysis process, such as visualizing behavior that may not have been apparent from the results alone.

```
In [14]: learning_curves(X_train, y_train, X_test, y_test)

Creating learning curve graphs for max_depths of 1, 3, 6, and 10. . .
```

9

Decision Tree Regressor Learning Performances
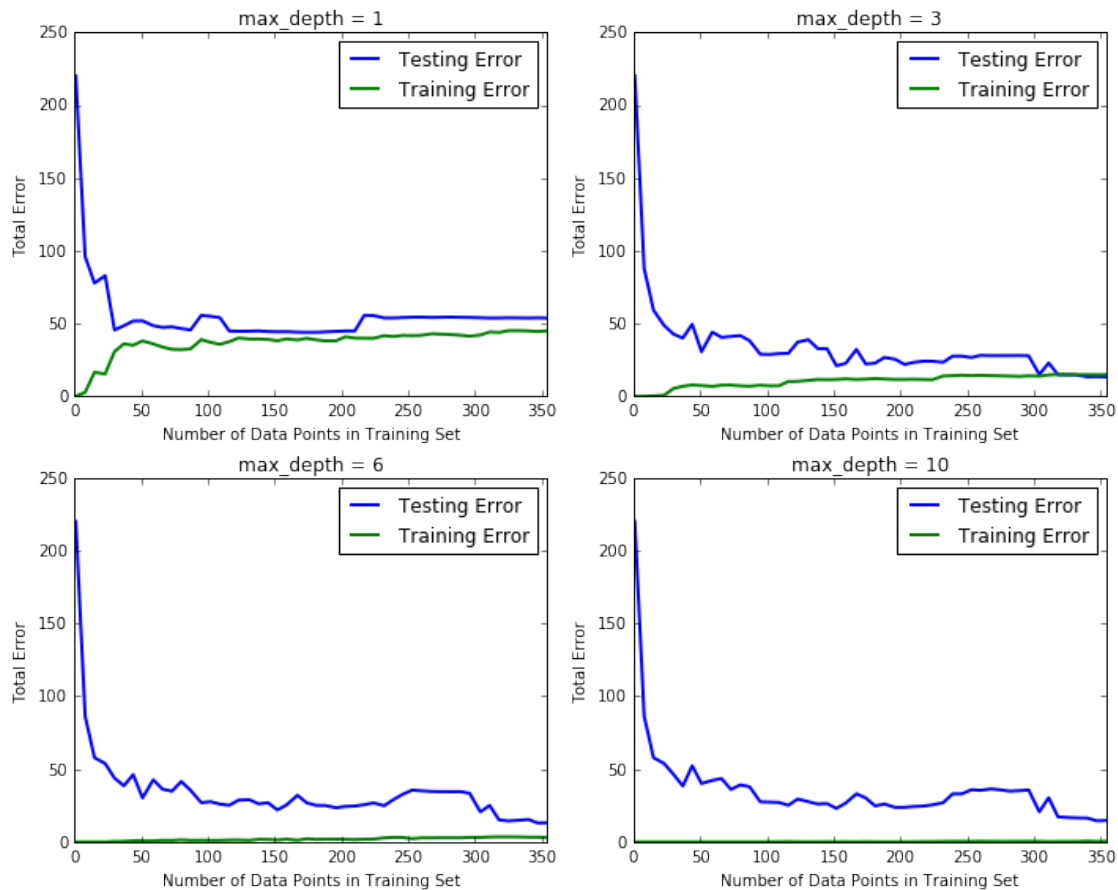
6.1 Question 7

Choose one of the learning curve graphs that are created above. What is the max depth for the chosen model? As the size of the training set increases, what happens to the training error? What happens to the testing error?

Answer: The max depth of the chosen model is 3 which is suitfitting as the number of data larger than 300. In the case of max depth equals 1, the model is underfitting, because the training error is high. In the case of max depth equals 6 or 10, the model is overfitting, because the training error is nearly 0 while the test error remains large. As the size of the training set increases, the training error increases from zero slowly , and the testing error decreases dramatically first, then decreases slowly towards trainning error.
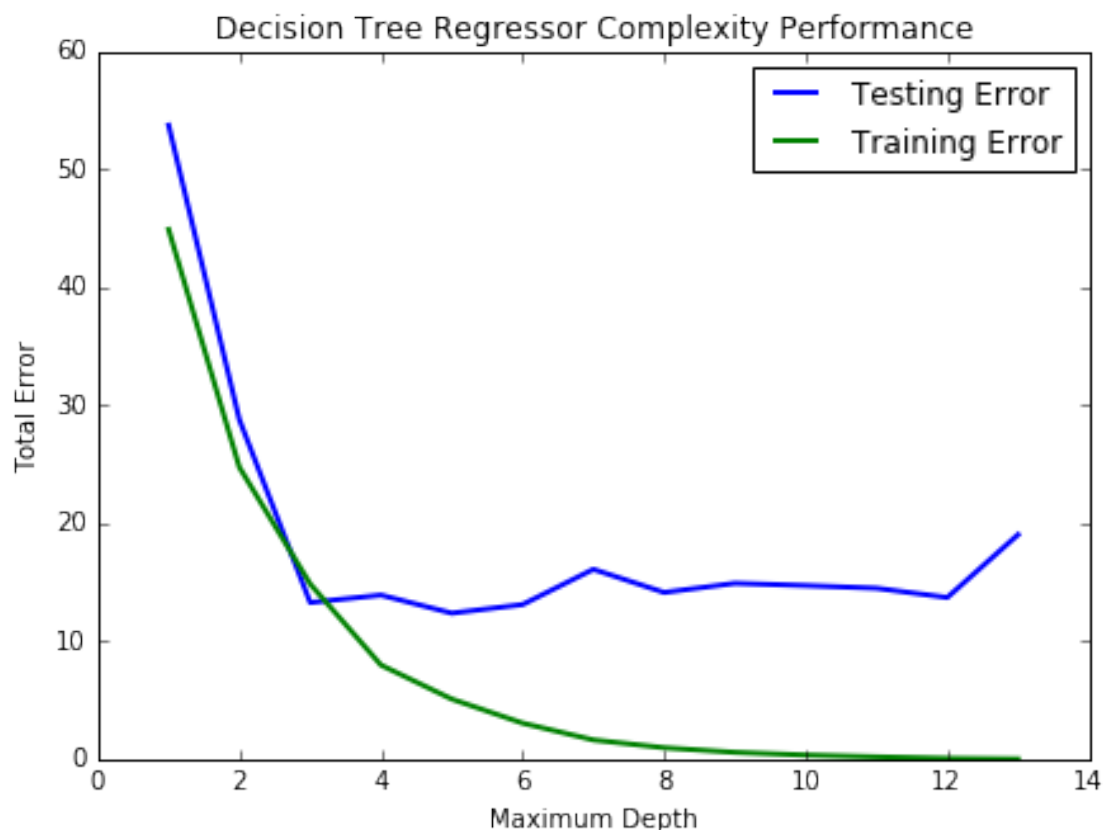
6.2 Question 8

Look at the learning curve graphs for the model with a max depth of 1 and a max depth of 10. When the model is using the full training set, does it suffer from high bias or high variance when the max depth is 1? What about when the max depth is 10?

Answer: Using the full training set, the model suffers from high bias when the max depth is 1, while, high variance when the max depth is 10. The training error in case of max depth equals 1 is larger than the testing error of the max depth 10 case's, so the former suffers from underfitting which is high bias. The max depth of 10's testing error is larger than its training error a lot, so it suffers from overfiting which is high variance.

```
In [17]: model_complexity(X_train, y_train, X_test, y_test)

Creating a model complexity graph. . .
```


6.3 Question 9

From the model complexity graph above, describe the training and testing errors as the max depth increases. Based on your interpretation of the graph, which max depth results in a model that best generalizes the dataset? Why?

Answer: As the max depth increases, the training errors decreases all the way, while, the testing errors decreases first and increases later. The max depth 3 results in a model that best generalizes the dataset. Because the training errors is reletively small and the testing errors is the smallest gl

7   Model Prediction

In this final section of the project, you will make a prediction on the client's feature set using an optimized model from `fit_model`. When applying grid search along with cross-validation to optimize your model, it would typically be performed and validated on a training set and subsequently evaluated on a **dedicated test set**. In this project, the optimization below is performed on the *entire dataset* (as opposed to the training set you made above) due to the many outliers in the data. Using the entire dataset for training provides for a less volatile prediction at the expense of not testing your model's performance.

To answer the following questions, it is recommended that you run the code blocks several times and use the median or mean value of the results.

7.1   Question 10

Using grid search on the entire dataset, what is the optimal `max_depth` parameter for your model? How does this result compare to your intial intuition?
Hint: Run the code block below to see the max depth produced by your optimized model.

```
In [47]: print "Final model has an optimal max_depth parameter of", reg.get_params

Final model has an optimal max_depth parameter of 5
```

Answer: The optimal max_depth parameter for my model is 5 which is larger than the result is the third section where the optimal max_depth is 3, because gridsearchcv makes full use of the dataset.

7.2   Question 11

With your parameter-tuned model, what is the best selling price for your client's home? How does this selling price compare to the basic statistics you calculated on the dataset?
Hint: Run the code block below to have your parameter-tuned model make a prediction on the client's home.

```
In [48]: sale_price = reg.predict(CLIENT_FEATURES)
         print "Predicted value of client's home: {0:.3f}".format(sale_price[0])

Predicted value of client's home: 20.968
```

Answer: 20.968. This is reasonable comparing to the statistics on the dataset where the mean house price is 22.533 and the median house price is 21.2.

7.3   Question 12 (Final Question):

In a few sentences, discuss whether you would use this model or not to predict the selling price of future clients' homes in the Greater Boston area.
Answer: I may use this model to predict the selling price of future clients's homes in the Greater Boston area. As we can see in qrestion 11, the predicted value is reasonable. From the view of statistics, our MSE is about 10 corresponds to a standerd MSE 3 which is small enough comparing to the standard deviation of house price 9.188. Meanwhile, I also want to try some other regression model to achieve a better performance.