Machine Learning Engineer Nanodegree

Model Evaluation & Validation

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 (https://archive.ics.uci.edu/ml/datasets/Housing), which is provided by the **UCI Machine Learning Repository**.

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 (http://daringfireball.net/projects/markdown/syntax), 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 [1]:
```
# 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 selli
CLIENT_FEATURES = [[11.95, 0.00, 18.100, 0, 0.6590, 5.6090, 90.00, 1.385,

# 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!
```

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.

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 [2]:
```
# Number of houses in the dataset
total_houses = np.array(housing_prices).size

# Number of features in the dataset
total_features = np.array(housing_features).shape[1]
# Minimum housing value in the dataset
minimum_price = np.amin(housing_prices)

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

# 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
```

Question 1

As a reminder, you can view a description of the Boston Housing dataset here (https://archive.ics.uci.edu/ml/datasets/Housing), 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: The three most significant attributes are: CRIM, RM, LSTAT.

CRIM - This measures the crime rate on a "per head" basis. Generally, the suburbs with higher crime rates tend to have houses with lower value. This is particularly evident towards the end of the data set.

RM - This specifies the average number of rooms per house in a particular suburb. If other values are equal, the higher the number of rooms, the more the property is valued at.

LSTAT - This measures the pecentage of population that are of lesser means. Suburbs containing houses lower values shows a higher percentage of LSTAT.

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 [3]:
```
print CLIENT_FEATURES
```

```
[[11.95, 0.0, 18.1, 0, 0.659, 5.609, 90.0, 1.385, 24, 680.0, 20.2, 332
.09, 12.13]]
```

Answer:

CRIM corresponds to the 1st value - 11.95

RM corresponds to the 6th value - 5.609

LSTAT corresponds to the 13th value - 12.13

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.

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 [4]:
```
# Put any import statements you need for this code block here
import numpy as np
from sklearn.cross_validation import train_test_split

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

    # Shuffle and split the data

    X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size = 0.30, random_state = 0)

    # 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_feature
    print "Successfully shuffled and split the data!"
except:
    print "Something went wrong with shuffling and splitting the data."
```

```
Successfully shuffled and split the data!
```

Question 4

Why do we split the data into training and testing subsets for our model?

Answer:

We split our data into training and testing subsets because, if we utilize the entire dataset for coming up with a correct fit for the model, we would be in danger of overfitting. The code, in turn, may fail to generalise well. Hence it's always recommended to set aside a portion of the data (in our case - 30%) for testing. The algorithm/fit is based on the training data and it'll be applied to a previously unseen data (testing data) to see if it generalises well.

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 (http://scikit-learn.org/stable/modules/classes.html#sklearn-metrics-metrics) 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 import 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 calculation!*" is printed.

In [5]:
```
# Put any import statements you need for this code block here
from sklearn.metrics import mean_absolute_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_absolute_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!
```

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:

Since we are dealing with continous data, we'll need to apply a regression metric. Hence we can eliminate the first four metrics (Accuracy, Precision, Recall and F1 score) since they're all Classification metrics.

Among the remaining, I've chosen to go with Mean Absolute Error. The reason for this is that Mean Squared Error emphasises larger errors. MAE on the other hand is a bit more forgiving on outliers. On a dataset such as the one we're dealing with here (comparitively more number of outliers) where an unnaturally large error can influence the price by a significant margin, it would be prudent to go with MAE.

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 2**. See the sklearn `make_scorer` documentation (http://scikit-learn.org/stable/modules/generated/sklearn.metrics.make_scorer.html).
- Build a GridSearchCV object using `regressor`, `parameters`, and `scoring_function`. See the sklearn documentation on GridSearchCV (http://scikit-learn.org/stable/modules/generated/sklearn.grid_search.GridSearchCV.html).

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 [6]:
```
# Put any import statements you need for this code block
from sklearn.metrics import 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_absolute_error, greater_is_better

    # Make the GridSearchCV object
    reg = GridSearchCV(regressor, parameters, scoring_function)

    # Fit the learner to the data to obtain the optimal model with tuned
    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!
```

Question 5

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

Answer:

Grid Search Algorithm is an approach to finding the best parameter for the model. It evaluates the model based on the individual specified parameters or a combination of them.

It is appicable when there is a combination of factors/parameters affecting the best fit for a given model. It evaluates each combination and comes with the ideal combination of parameters.

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 is a technique to avoid over-fitting in a model which is especially useful in the event of limited data.

Cross-validation is performed by taking a different chunk of training data each time , training the model based on those data, and then applying it to the remaining data (test set). This is iterated till the available data is exhausted. Hence, every chunk of data is used as a training as well as a testing set.

For example, in k-fold cross-validation which is the basic form of cross validation, the data is first partitioned in 'k' equally sized chunks. Then, 'k' iterations of testing and validation is performed during each iteration of which, a different chunk of data is used for testing and training.

Cross validation is an essential subset of grid search. Grid search fetches the best fit or parameter combination by cycling through all the input parameters which would be impossible without cross validation. Cross-validation, due to the fact that it goes through the entire dataset during testing and training, provides an average value which keeps unnatural outliers in check and hence is essential to finding the best possible parameter through grid search.

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 [7]:
```
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 plot

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

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

    # We will vary the training set size so that we have 50 different siz
    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 wi
            regressor = DecisionTreeRegressor(max_depth = depth)

            # 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.pred

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

        # 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', fontsiz
    fig.tight_layout()
    fig.show()
```

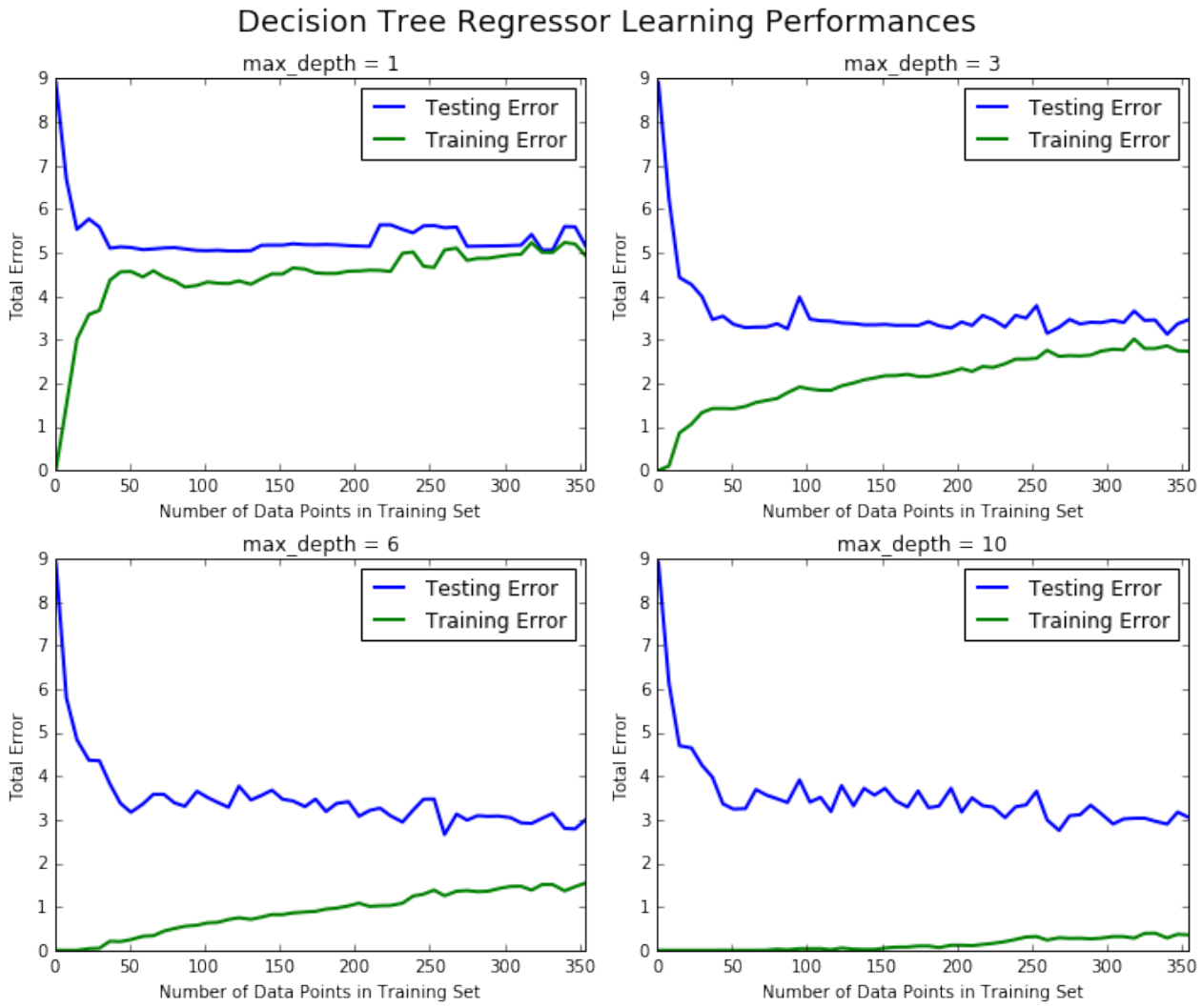
In [8]:
```
def model_complexity(X_train, y_train, X_test, y_test):
    """ Calculates the performance of the model as model complexity incre
        The learning and testing errors rates are then plotted. """

    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 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_tr

        # 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()
```

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 [10]:
```
learning_curves(X_train, y_train, X_test, y_test)
```

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



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.

As the size of the training set increases, the training error increases. The testing has tended towards stabilisation after about 100 data points. This is because the data has learned whatever it can with the given depth.

When the number of data the model has gone through is less than 100, the testing error, which is intially very high, incrementally climbs down. This is becuase the model is slowly learning from more data which directly influences the final value (housing price).

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:

When max depth is 1, not only does the training error not decrease, it keeps climbing. But unlike the other graphs resulting from different "depths" where the training data graph is increasing on a slight incline, the training error shoots up till 50 datapoints are analysed and then stabilises. This is also coupled with the fact that both the testing and training error graphs converge quickly. This is an indication that after the initial burst of anlysis (till 50 points), the model fails to learn from the dataset and that it is a similar story with the testing data. Clearly the model is too simple and suffers from high bias.

When max depth is 10, the training error is very minimal. The model has been overfit and suffers from high variance. Also of note here, is the big difference between the testing and training graphs. This further supports the fact that the model has been overfit to how much worse the model reacts to unseen data (test data) while being acutely (and overly) tuned to the variances in the training data.

In [11]:
```
model_complexity(X_train, y_train, X_test, y_test)
```

```
Creating a model complexity graph. . .
```

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 maximum depth increases, the training error continues to decrease, almost linearly. The testing error meanwhile, stabilises after a depth of 6.

Based on my interpretation, a max depth of 6 results in a model that best generalizes the dataset as the testing error plateaus and stabilises.

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.

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 [12]:
```
print "Final model has an optimal max_depth parameter of", reg.get_params
```

```
Final model has an optimal max_depth parameter of 4
```

Answer:

Optimal max_depth parameter is 5.

The result is one less than my initial intuition.

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 [13]:
```
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: 21.630
```

Answer:

Best selling price seems to be in the region of 21.560

It is very close to the mean and median prices of the homes. The maximum and minimum prices were 50 and 5 respectively.

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:

Yes, I would definitely use this model to predict the selling price of the future clients homes in the Greater Boston Area.

A simple scan of the housing dataset in the repository and comparing the houses with features similar to the one represented in the model would reveal pricing which accurately fluctuates with the fluctuation in the features.

For example, one of the dataset is:

12.04820, 0.00, 18.100, 0, 0.6140, 5.6480, 87.60, 1.9512, 24, 666.0, 20.20, 291.55, 14.10 has a price of 20.80

And our feature set which includes:

11.95, 0.0, 18.1, 0, 0.659, 5.609, 90.0, 1.385, 24, 680.0, 20.2, 332.09, 12.13 has a price of 21.560.

In [ ]:

In [ ]:

In [ ]:

In [ ]: