Boston Housing Prices

Udacity Machine Learning Engineer
Nanodegree Program: Project 1

Anderson Daniel Trimm

January 29, 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.

(♣ THIS PART ON CROSS VALIDATION WITH GRIDSEARCH.) However, we still have a parameter in our model - namely the depth of the decision tree, which we need to optimize. We will do this using grid search over a range of possible values for this parameter. Grid search will run our algorithm over the training and testing data for each of these parameters, and we can compare We first import `grid_search` from `sklearn`:

```
from sklearn import grid_search
```