Machine Learning Engineering Nanodegree

Project 1: Predicting Boston Housing Prices

version 0.3

Jouni Huopana
11/18/2015

CONTENTS

1. INTRODUCTION

This document contains information relating to the Boston Housing Pricing dataset, which is used for the development of a machine learning model to help estimate housing prices with given data. Basic statistics of the data are derived and used then to create a cross validated prediction model. The model is then used to predict house price for a given input and the model performance is discussed. The Python code and example run can be found from the Appendix. This document has been created as a project work for the Udacity Machine Learning Engineer Nanodegree.

2. STATISTICAL ANALYSIS AND DATA EXPLORATION

The Boston Housing Price dataset is one of the example datasets in Python's Scikit-learn library. The dataset is divided into a dataset dataframe and into a target vector. Table 1 Description of the Boston Housing Price dataset, shows the summary provided with the dataset.

Table 1 Description of the Boston Housing Price dataset

Data Set Characteristics:	Value
Number of Instances	506
Number of Attributes	13
Median Value (attribute 14) is usually the target	
Attribute Information (in order)	
- CRIM per capita crime rate by town	
- ZN proportion of residential land zoned for lots over 25,000 sq.ft.	
- INDUS proportion of non-retail business acres per town	
- CHAS Charles River dummy variable (= 1 if tract bounds river; 0 otherwise)	
- NOX nitric oxides concentration (parts per 10 million)	
- RM average number of rooms per dwelling	
- AGE proportion of owner-occupied units built prior to 1940	
- DIS weighted distances to five Boston employment centres	
- RAD index of accessibility to radial highways	
- TAX full-value property-tax rate per $10,000	
- PTRATIO pupil-teacher ratio by town	
- B 1000(Bk - 0.63)^2 where Bk is the proportion of blacks by town	
- LSTAT % lower status of the population	
- MEDV Median value of owner-occupied homes in $1000's	
Missing Attribute Values	None
Creator	Harrison, D. and Rubinfeld, D.L.
This is a copy of UCI ML housing dataset.	
http://archive.ics.uci.edu/ml/datasets/Housing	

Analysing the dataset shows that it has the following properties.

Housing price (target value)

Minimum value 5.00

Maximum value 50.00

Mean value 22.53

Median value 21.20

Standard deviation 9.19

Histogram of the housing price is plotted in Figure 1. It supports the calculated values.

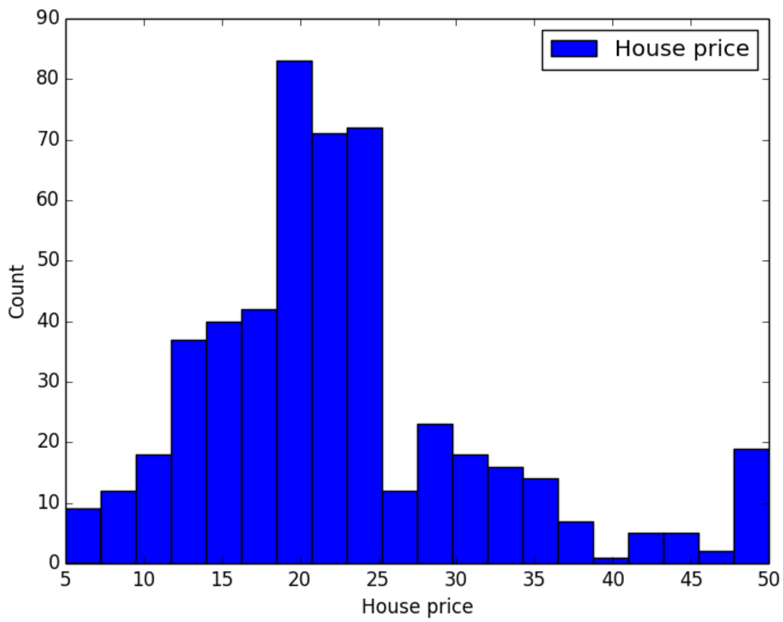

Figure 1 Histogram of the house price

Generally speaking the dataset is a good example set, but small by the number of instances. Also it has no missing values, which makes it easier to use.

3. EVALUATING MODEL PERFORMANCE

To evaluate the performance, mean square error (MSE, equation 1) is used. It is a good and simple method to estimate the error in the model and fits well to be used with regression, as it calculates the summed of the squared “distances” from the prediction to the target values and divides it by the number of samples *n*. This makes it a good method to estimate continues values, such as the target price and penalizes more the larger errors than small ones. Other measurement type as F-beta and log-loss are more suitable for classification problems.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(Prediction - Target)^2 \qquad (1.)$$

The dataset is split into two datasets; training and testing. This is done to estimate the model performance during training and to prevent overfitting. If the whole dataset would be used for the training, it would be very easy to overfit the model. The model would be very good in predicting the training values, but would be sensitive noise in the data and most likely give a bad prediction for data outside the train dataset.

To fight overfitting, cross validation is used to ensure that the training is done only with parts of the data at any time to ensure robustness. This is also useful when the dataset is small. In this case 10-fold cross validation was used to find and optimize the model parameters using grid search. The implementation can be seen in Appendix “Python code for the data analysis and fitting “boston_housing_v03.py””. As the dataset was small, a large number parameters were chosen, which lead to totaling of 16200 fits (in 45 seconds total). From these fits the best parameters where chosen to be implemented in the final model train. Grid search is an effective way to find parameters when you are working with large number of parameters and it is difficult to predict any continuous dependencies and there is no clear global minimum (or maximum). Found parameters for the decision tree regressor model are shown here below.

DecisionTreeRegressor

```
 criterion='mse',
 max_depth=6,
 max_features=8,
max_leaf_nodes=None,
 min_samples_leaf=1,
 min_samples_split=2,
min_weight_fraction_leaf=0.01,
 presort=False,
 random_state=123,
splitter='best')
```

4. Analyzing Model Performance

When looking at the training and testing errors, it can be seen that when the training size increases the error in the training set increases slowly and stabilizes to a certain level, but with the test set the error decreases and starts to become more and more noisy when the training size increases. This can be seen in the figures 2 and 3. When the model has the max depth of 10, it can be seen that the model starts to suffer from overfitting with the higher training sizes. With depth of 1 the lower complexity provides stability even with higher training size, but cannot describe the behavior with needed detail keeping the overall error high.

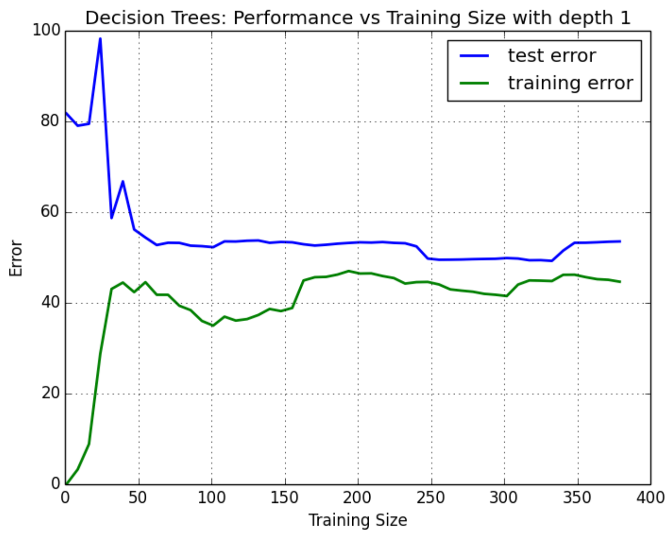

Figure 2 Decision Tree performance with tree depth of 1

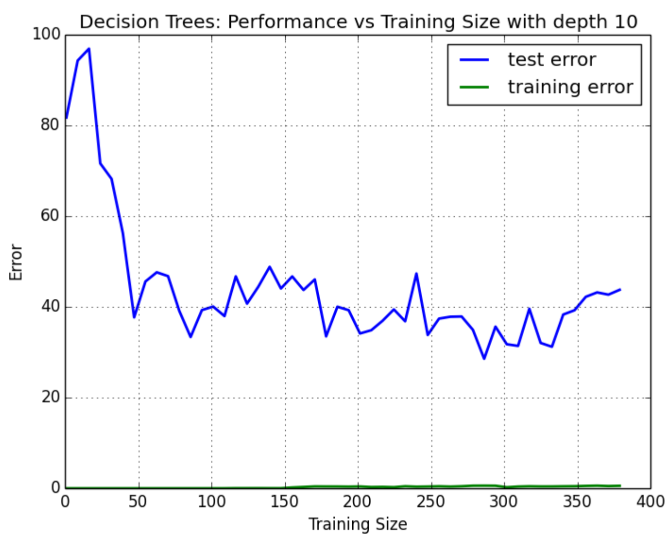

Figure 3 Decision Tree performance with tree depth of 10

The model complexity graph, show in Figure 4, shows the decrease of the training error as the depth of the model increases. The test error shows relatively good behavior quite early, but at that stage the train error is still quite high. When moving to larger depths the test error slightly increases and becomes a bit noisier. Taking the compromise of the both training and test errors, it can be said that the optimal depth is between 4 and 8. The performed grid search confirms this assessment by settling to the value of 6.

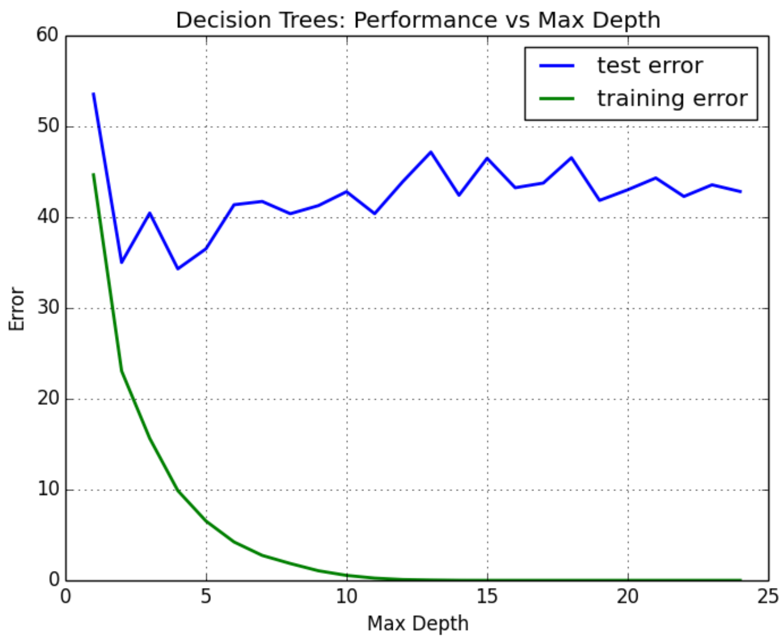

Figure 4 Complexity graph of the decision tree model

5. MODEL PREDICTION

Model makes predicted housing price with parameters obtained in the grid search. As part of this work a vector was given to predict the house price with given values. The vector describing the house is shown below.

House: [11.95 0.00 18.10 0.00 0.66 5.61 90.00 1.39 24.00 680.00 20.20 332.09 12.13]

Prediction: [20.41]

With the given house vector the model prediction is 20.41.

It can be expected that the value is a decent prediction of the target value. The estimated vector has some values which are close to the mean values of the data set and only few which closer to the extreme values. On the other hand as it can be seen form figure 5, the dataset is almost the widest at the predicted value, which means that in can have some uncertainties to it.

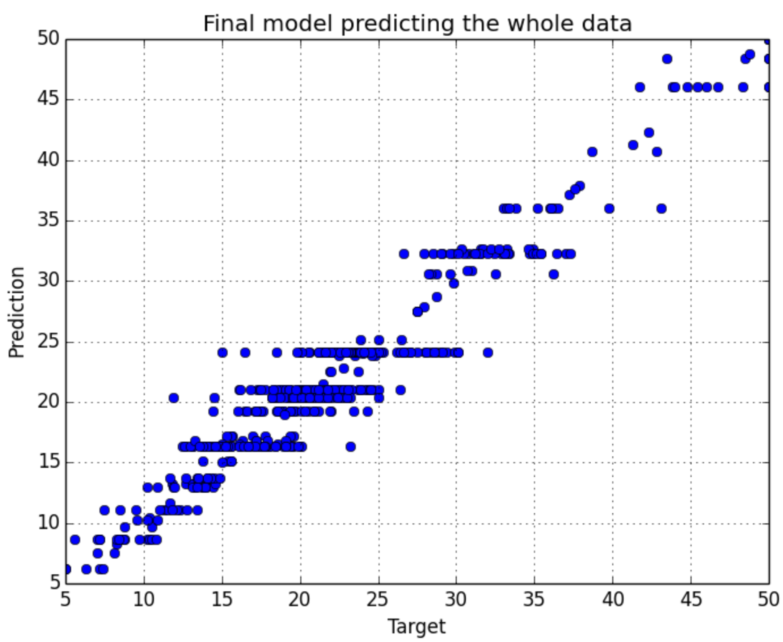

Figure 5 Comparison of target and prediction values for the whole dataset

APPENDIX

PYTHON CODE FOR THE DATA ANALYSIS AND FITTING "BOSTON_HOUSING_V03.PY"

```
## This code has been originally downloaded as a part of Udacity nanodegree.
## It has been modified by Jouni Huopana 15th of Nov 2015, in order to answer
## to posed questions. All modifications are for test use only.

"""Load the Boston dataset and examine its target (label) distribution."""
# version 0.2
# increased test sample size from 0.1 to 0.25
# added a histogram plot for the housing price
# version 0.3
# Corrections on the scoring method used


# Load libraries
import numpy as np
import pylab as pl
from sklearn import datasets
from sklearn.tree import DecisionTreeRegressor

################################
### ADD EXTRA LIBRARIES HERE ###
################################
from sklearn.cross_validation import train_test_split
from sklearn.metrics import make_scorer
from sklearn.metrics import mean_squared_error
from sklearn import grid_search

## Nicer format for floats in commad line
## Source for the formating code
## http://stackoverflow.com/questions/21008858/formatting-floats-in-a-numpy-array
float_formatter = lambda x: "%.2f" % x
np.set_printoptions(formatter={'float_kind':float_formatter})

## For saving figures set fig_save = 1
fig_save = 1

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

    boston = datasets.load_boston()
    return boston


def explore_city_data(city_data):
    """Calculate the Boston housing statistics."""

    # Get the labels and features from the housing data
    housing_prices = city_data.target
    housing_features = city_data.data
```

```
####################################
### Step 1. YOUR CODE GOES HERE ###
####################################

# Please calculate the following values using the Numpy library
# Size of data?

nrow, ncol = housing_features.shape
print("\n******************************************************\n")
print("  The Boston city data has the following properties:")
print("\n******************************************************\n")
print("There is %i rows in the data." % nrow)
print("There is %i columns in the data." % ncol)
print("Total of %i values." % housing_features.size)

print(city_data.DESCR)

# Minimum value?
# Calculating the minimum value with the Numpy's min function
tmin = np.min(housing_features)
print("\nMinimum value of the whole dataset is %.2f." % tmin)
# Calculating the column minimums with the Numpy's amin function
cmin = np.amin(housing_features, axis=0)
print("Column specific minimums are :")
print(cmin)
print("Housing price minimum is %.2f" % np.min(housing_prices))
# Maximum Value?
# Corresponding Numpy functions are used for the maximums, means, medians,
# means and standard deviations.
tmax = np.max(housing_features)
print("\nMaximum value of the whole dataset is %.2f." % tmax)
cmax = np.amax(housing_features, axis=0)
print("Column specific maximums are :")
print(cmax)
print("Housing price maximum is %.2f" % np.max(housing_prices))
# Calculate mean?
cmean = np.mean(housing_features, axis=0)
print("\nColumn specific means are :")
print(cmean)
print("Housing price mean is %.2f" % np.mean(housing_prices))
# Calculate median?
cmed  = np.median(housing_features, axis=0)
print("\nColumn specific medians are :")
print(cmed)
print("Housing price median is %.2f" % np.median(housing_prices))
# Calculate standard deviation?
cstd  = np.median(housing_features, axis=0)
print("\nColumn specific standard deviations are :")
print(cstd)
print("Housing price standard deviation is %.2f" % np.std(housing_prices))

#Plot price histogram
n, bins, patches = pl.hist(housing_prices, 20, histtype='bar', label=['House
price'])
pl.xlabel('House price')
pl.ylabel('Count')
pl.legend()
```

```python
    # Figure saving
    if fig_save==1:
        pl.savefig('hist.png')
    pl.show()


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

    ###################################
    ### Step 2. YOUR CODE GOES HERE ###
    ###################################

    # Calculating mean square error for the prediction
    # Scikit has its own function, but own one writen for practice
    # mse = mean_squared_error(label, prediction)
    mse = (1./prediction.size)*sum(np.power((prediction-label),2))
    # http://scikit-learn.org/stable/modules/classes.html#sklearn-metrics-metrics
    return mse


def split_data(city_data):
    """Randomly shuffle the sample set. Divide it into training and testing set."""

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

    ###################################
    ### Step 3. YOUR CODE GOES HERE ###
    ###################################

    # Creating the train and test sets with scikit's train_teast_split function
    # It provides easy set split with random sets.
    # Originally test_size=0.1 according to feedback changed to 0.25
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25,
random_state=123)

    return X_train, y_train, X_test, y_test


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.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])
```

```python
        # 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(depth, sizes, train_err, test_err)


def learning_curve_graph(depth, sizes, train_err, test_err):
    """Plot training and test error as a function of the training size."""
    # depth also passed for more accurate plot titles

    pl.figure()
    pl.title('Decision Trees: Performance vs Training Size with depth %i' % depth)
    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')
    # Grid added for clarity
    pl.grid()
    # Figure saving
    if fig_save==1:
        pl.savefig('dt_d_%i.png' % depth)
    pl.show()


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
learn."""

    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()
    # Grid added for clarity
    pl.grid()
    pl.xlabel('Max Depth')
    pl.ylabel('Error')
    # Figure saving
    if fig_save==1:
        pl.savefig('comp.png')
    pl.show()


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()

    parameters = {'max_depth':(1,2,3,4,5,6,7,8,9,10)}

    ###################################
    ### Step 4. YOUR CODE GOES HERE ###
    ###################################

    # 1. Find the best performance metric
    # should be the same as your performance_metric procedure
    # http://scikit-
learn.org/stable/modules/generated/sklearn.metrics.make_scorer.html
    rmse_scorer = make_scorer(performance_metric, greater_is_better = False)

    # 2. Use gridearch to fine tune the Decision Tree Regressor and find the best
model
    # http://scikit-
learn.org/stable/modules/generated/sklearn.grid_search.GridSearchCV.html#sklearn.gr
id_search.GridSearchCV

    #Grid searching the Tree regressors parameters
    parameters = {'max_features':[1, 2, 3, 4, 5, 6, 7, 8 ,9],
                  'max_depth':[1, 2, 3, 4, 5 ,6 ,7 ,8, 9],
                  'min_samples_leaf':[1,2,3,4],
                  'min_weight_fraction_leaf':[0.01,0.05,0.1,0.2,0.3],
                  'random_state':[123]}

    # Fit the learner to the training data
    # Default scorer for the DecisionTreeRegressor is mse
    reg = grid_search.GridSearchCV(regressor, parameters, cv=10, verbose=1,
scoring=rmse_scorer)
    print "Final Model: "
```

```python
    print reg.fit(X, y)

    # Printing the best model form the grid search
    print(reg.best_estimator_)

    # Use the model to predict the output of a particular sample
    x = np.array([11.95, 0.00, 18.100, 0, 0.6590, 5.6090, 90.00, 1.385, 24, 680.0, 
20.20, 332.09, 12.13])
    y = reg.predict(x)
    print "House: " + str(x)
    print "Prediction: " + str(y)

    #Plotting the final fit with the all of the data
    y = reg.predict(X)
    pl.plot(city_data.target, y, 'bo', label = 'Train data')
    pl.grid()
    pl.title('Final model predicting the whole data')
    pl.xlabel('Target')
    pl.ylabel('Prediction')
    # Figure saving
    if fig_save==1:
        pl.savefig('final_model.png')
    pl.show()



def main():
    """Analyze the Boston housing data. Evaluate and validate the
    performanance of a Decision Tree regressor on the housing data.
    Fine tune the model to make prediction on unseen data."""

    # Load data
    city_data = load_data()

    # Explore the data
    explore_city_data(city_data)

    # Training/Test dataset split
    X_train, y_train, X_test, y_test = split_data(city_data)

    # Learning Curve Graphs
    max_depths = [1,2,3,4,5,6,7,8,9,10]
    for max_depth in max_depths:
        learning_curve(max_depth, X_train, y_train, X_test, y_test)

    # Model Complexity Graph
    model_complexity(X_train, y_train, X_test, y_test)

    # Tune and predict Model
    fit_predict_model(city_data)


if __name__ == "__main__":
    main()
```

EXAMPLE OF THE COMMAND LINE OUTPUT:

```
******************************************************

  The Boston city data has the following properties:

******************************************************

There is 506 rows in the data.
There is 13 columns in the data.
Total of 6578 values.
Boston House Prices dataset

Notes
------
Data Set Characteristics:

    :Number of Instances: 506

    :Number of Attributes: 13 numeric/categorical predictive

    :Median Value (attribute 14) is usually the target

    :Attribute Information (in order):
        - CRIM     per capita crime rate by town
        - ZN       proportion of residential land zoned for lots over 25,000 sq.ft.
        - INDUS    proportion of non-retail business acres per town
        - CHAS     Charles River dummy variable (= 1 if tract bounds river; 0 otherwise)
        - NOX      nitric oxides concentration (parts per 10 million)
        - RM       average number of rooms per dwelling
        - AGE      proportion of owner-occupied units built prior to 1940
        - DIS      weighted distances to five Boston employment centres
        - RAD      index of accessibility to radial highways
        - TAX      full-value property-tax rate per $10,000
        - PTRATIO  pupil-teacher ratio by town
        - B        1000(Bk - 0.63)^2 where Bk is the proportion of blacks by town
        - LSTAT    % lower status of the population
        - MEDV     Median value of owner-occupied homes in $1000's

    :Missing Attribute Values: None

    :Creator: Harrison, D. and Rubinfeld, D.L.

This is a copy of UCI ML housing dataset.
http://archive.ics.uci.edu/ml/datasets/Housing


This dataset was taken from the StatLib library which is maintained at Carnegie Mellon University.

The Boston house-price data of Harrison, D. and Rubinfeld, D.L. 'Hedonic
prices and the demand for clean air', J. Environ. Economics & Management,
```

vol.5, 81-102, 1978. Used in Belsley, Kuh & Welsch, 'Regression diagnostics ...', Wiley, 1980. N.B. Various transformations are used in the table on pages 244-261 of the latter.

The Boston house-price data has been used in many machine learning papers that address regression
problems.

References

- Belsley, Kuh & Welsch, 'Regression diagnostics: Identifying Influential Data and Sources of Collinearity', Wiley, 1980. 244-261.
- Quinlan,R. (1993). Combining Instance-Based and Model-Based Learning. In Proceedings on the Tenth International Conference of Machine Learning, 236-243, University of Massachusetts, Amherst. Morgan Kaufmann.
- many more! (see http://archive.ics.uci.edu/ml/datasets/Housing)

Minimum value of the whole dataset is 0.00.
Column specific minimums are :
[0.01 0.00 0.46 0.00 0.39 3.56 2.90 1.13 1.00 187.00 12.60 0.32 1.73]
Housing price minimum is 5.00

Maximum value of the whole dataset is 711.00.
Column specific maximums are :
[88.98 100.00 27.74 1.00 0.87 8.78 100.00 12.13 24.00 711.00 22.00 396.90
 37.97]
Housing price maximum is 50.00

Column specific means are :
[3.59 11.36 11.14 0.07 0.55 6.28 68.57 3.80 9.55 408.24 18.46 356.67 12.65]
Housing price mean is 22.53

Column specific medians are :
[0.26 0.00 9.69 0.00 0.54 6.21 77.50 3.21 5.00 330.00 19.05 391.44 11.36]
Housing price median is 21.20

Column specific standard deviations are :
[0.26 0.00 9.69 0.00 0.54 6.21 77.50 3.21 5.00 330.00 19.05 391.44 11.36]
Housing price standard deviation is 9.19
Decision Tree with Max Depth:
1
Decision Tree with Max Depth:
2
Decision Tree with Max Depth:
3
Decision Tree with Max Depth:
4
Decision Tree with Max Depth:
5
Decision Tree with Max Depth:

```
6
Decision Tree with Max Depth:
7
Decision Tree with Max Depth:
8
Decision Tree with Max Depth:
9
Decision Tree with Max Depth:
10
Model Complexity:
Final Model:
Fitting 10 folds for each of 1620 candidates, totalling 16200 fits
GridSearchCV(cv=10, error_score='raise',
       estimator=DecisionTreeRegressor(criterion='mse', max_depth=None, max_features=None,
           max_leaf_nodes=None, min_samples_leaf=1, min_samples_split=2,
           min_weight_fraction_leaf=0.0, presort=False, random_state=None,
           splitter='best'),
       fit_params={}, iid=True, n_jobs=1,
       param_grid={'max_features': [1, 2, 3, 4, 5, 6, 7, 8, 9], 'random_state': [123],
'min_weight_fraction_leaf': [0.01, 0.05, 0.1, 0.2, 0.3], 'max_depth': [1, 2, 3, 4, 5, 6, 7, 8, 9],
'min_samples_leaf': [1, 2, 3, 4]},
       pre_dispatch='2*n_jobs', refit=True,
       scoring=make_scorer(performance_metric, greater_is_better=False),
       verbose=1)
DecisionTreeRegressor(criterion='mse', max_depth=6, max_features=8,
           max_leaf_nodes=None, min_samples_leaf=1, min_samples_split=2,
           min_weight_fraction_leaf=0.01, presort=False, random_state=123,
           splitter='best')
House: [11.95 0.00 18.10 0.00 0.66 5.61 90.00 1.39 24.00 680.00 20.20 332.09 12.13]
Prediction: [20.41]
```