The Boston Housing Problem

The Boston Housing dataset was originally published by Harrison and Rubinfeld in 1978, in a paper titled *Hedonic prices and the demand for clean air*. The authors used the dataset to investigate the relationship between air quality and housing prices in the Boston metropolitan area. Since then, the dataset has been widely used in machine learning and statistics research as a benchmark for regression tasks.

Each row in the [data table (../_data/housing_data.txt)](../_data/housing_data.txt) contains various [attributes (../_data/housing_names.txt)](../_data/housing_names.txt) of a a district in Boston. The last attribute is the median value of owner occupied homes in the district. We will create regressors that estimate this attribute from other attributes of the district.

Exercise 1: Load the Boston Housing data set to DataFrame and display basic statistics about it!

In [1]:

```
# Extract column names.
fname = 'housing_names.txt'
columns = []
for line in open(fname):
    if line.startswith('    ') and line[4].isdigit():
        columns.append(line.split()[1])
```

In [2]:

```
columns
```

Out[2]:

```
['CRIM',
 'ZN',
 'INDUS',
 'CHAS',
 'NOX',
 'RM',
 'AGE',
 'DIS',
 'RAD',
 'TAX',
 'PTRATIO',
 'LSTAT',
 'MEDV']
```

In [ ]:

```

```

In [3]:

```
# Load to DataFrame.
fdata = 'housing_data.txt'
import pandas as pd
df = pd.read_csv(fdata, sep='\t', names=columns)
df
```

Out[3]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	LSTAT	MEDV
0	0.00632	18.0	2.31	0	0.538	6.575	65.2	4.0900	1	296.0	15.3	4.98	24.0
1	0.02731	0.0	7.07	0	0.469	6.421	78.9	4.9671	2	242.0	17.8	9.14	21.6
2	0.02729	0.0	7.07	0	0.469	7.185	61.1	4.9671	2	242.0	17.8	4.03	34.7
3	0.03237	0.0	2.18	0	0.458	6.998	45.8	6.0622	3	222.0	18.7	2.94	33.4
4	0.06905	0.0	2.18	0	0.458	7.147	54.2	6.0622	3	222.0	18.7	5.33	36.2
...	...	...	...	...	...	...	...	...	...	...	...	...	...
501	0.06263	0.0	11.93	0	0.573	6.593	69.1	2.4786	1	273.0	21.0	9.67	22.4
502	0.04527	0.0	11.93	0	0.573	6.120	76.7	2.2875	1	273.0	21.0	9.08	20.6
503	0.06076	0.0	11.93	0	0.573	6.976	91.0	2.1675	1	273.0	21.0	5.64	23.9
504	0.10959	0.0	11.93	0	0.573	6.794	89.3	2.3889	1	273.0	21.0	6.48	22.0
505	0.04741	0.0	11.93	0	0.573	6.030	80.8	2.5050	1	273.0	21.0	7.88	11.9

506 rows × 13 columns

In [5]:

```
# Check column data types.
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 506 entries, 0 to 505
Data columns (total 13 columns):
 #   Column   Non-Null Count  Dtype  
---  ------   --------------  -----  
 0   CRIM     506 non-null    float64
 1   ZN       506 non-null    float64
 2   INDUS    506 non-null    float64
 3   CHAS     506 non-null    int64  
 4   NOX      506 non-null    float64
 5   RM       506 non-null    float64
 6   AGE      506 non-null    float64
 7   DIS      506 non-null    float64
 8   RAD      506 non-null    int64  
 9   TAX      506 non-null    float64
 10  PTRATIO  506 non-null    float64
 11  LSTAT    506 non-null    float64
 12  MEDV     506 non-null    float64
dtypes: float64(11), int64(2)
memory usage: 51.5 KB
```

In [7]:

```
# Basic column statistics.
df.describe().T
```

Out[7]:

	count	mean	std	min	25%	50%	75%	max
CRIM	506.0	3.613524	8.601545	0.00632	0.082045	0.25651	3.677083	88.9762
ZN	506.0	11.363636	23.322453	0.00000	0.000000	0.00000	12.500000	100.0000
INDUS	506.0	11.136779	6.860353	0.46000	5.190000	9.69000	18.100000	27.7400
CHAS	506.0	0.069170	0.253994	0.00000	0.000000	0.00000	0.000000	1.0000
NOX	506.0	0.554695	0.115878	0.38500	0.449000	0.53800	0.624000	0.8710
RM	506.0	6.284634	0.702617	3.56100	5.885500	6.20850	6.623500	8.7800
AGE	506.0	68.574901	28.148861	2.90000	45.025000	77.50000	94.075000	100.0000
DIS	506.0	3.795043	2.105710	1.12960	2.100175	3.20745	5.188425	12.1265
RAD	506.0	9.549407	8.707259	1.00000	4.000000	5.00000	24.000000	24.0000
TAX	506.0	408.237154	168.537116	187.00000	279.000000	330.00000	666.000000	711.0000

Exercise 2: Build a univariate linear model for each column and measure it's RMSE (root mean squared error)! Use the full data set both for for model building and error measurement!

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2}$$

In [37]:

```
import numpy as np
y = df[columns[-1]].values
ym = y.mean()
y = y - ym
y
```

Out[37]:

```
array([  1.46719368,  -0.93280632,  12.16719368,  10.86719368,
        13.66719368,   6.16719368,   0.36719368,   4.56719368,
        -6.03280632,  -3.63280632,  -7.53280632,  -3.63280632,
        -0.83280632,  -2.13280632,  -4.33280632,  -2.63280632,
         0.56719368,  -5.03280632,  -2.33280632,  -4.33280632,
        -8.93280632,  -2.93280632,  -7.33280632,  -8.03280632,
        -6.93280632,  -8.63280632,  -5.93280632,  -7.73280632,
        -4.13280632,  -1.53280632,  -9.83280632,  -8.03280632,
        -9.33280632,  -9.43280632,  -9.03280632,  -3.63280632,
        -2.53280632,  -1.53280632,   2.16719368,   8.26719368,
        12.36719368,   4.06719368,   2.76719368,   2.16719368,
        -1.33280632,  -3.23280632,  -2.53280632,  -5.93280632,
        -8.13280632,  -3.13280632,  -2.83280632,  -2.03280632,
         2.46719368,   0.86719368,  -3.63280632,  12.86719368,
         2.16719368,   9.06719368,   0.76719368,  -2.93280632,
        -3.83280632,  -6.53280632,  -0.33280632,   2.46719368,
        10.46719368,   0.96719368,  -3.13280632,  -0.53280632,
        -5.13280632,  -1.63280632,   1.66719368,  -0.83280632,
```

In [40]:

```
for column in columns[:-1]:
    p = []
    x = df[column].values
    xm = x.mean()
    x = x - xm
    # univariate linear regression
    w = (x @ y)/(x @ x)
    p0 = x * w
    pr = y - p0
    p = pr**2
    # measure RMSE
    RMSE = np.sqrt(p.mean())
    print(column, RMSE)
```

```
CRIM 8.467038200100824
ZN 8.570396495772854
INDUS 8.04153105080589
CHAS 9.045800910882107
NOX 8.306881987569504
RM 6.603071389222562
AGE 8.510228018625197
DIS 8.896422965780745
RAD 8.492632800301259
TAX 8.117097716353989
PTRATIO 7.915314271320455
LSTAT 6.20346413142642
```

In [39]:

```
```

Out[39]:

```
-0.9500493537579913
```

Exercise 3: Build a multivariate linear model and measure its RMSE! Use the full data set both for for model building and error measurement!

In [50]:

```
X = df[columns[:-1]].values
```

In [51]:

```
w = np.linalg.solve(X.T @ X, X.T @ y)
```

In [52]:

```
p0 = X @ w
```

In [53]:

```
pr = y - p0
p = pr**2
```

In [54]:

```
RMSE = np.sqrt(p.mean())
RMSE
```

Out[54]:

```
4.807264396243256
```

Exercise 4: Introduce a bias term into the multivariate linear model!

In [60]:

```
X2 = np.hstack([X, np.ones((X.shape[0],1))])
```

In [61]:

```
w = np.linalg.solve(X2.T @ X2, X2.T @ y)
p0 = X2 @ w
pr = y - p0
p = pr**2
RMSE = np.sqrt(p.mean())
RMSE
```

Out[61]:

```
4.735998462783738
```

In [62]:

```
# Display the weights associated with the features.
C = columns[:-1]
W = w[:-1]
pd.Series(W,C)
```

Out[62]:

```
CRIM       -0.121389
ZN          0.046963
INDUS       0.013468
CHAS        2.839993
NOX       -18.758022
RM          3.658119
AGE         0.003611
DIS        -1.490754
RAD         0.289405
TAX        -0.012682
PTRATIO    -0.937533
LSTAT      -0.552019
dtype: float64
```

In [71]:

```
# Display the weights associated with the features, after scaling the columns
Xs = X / X.std(axis=0)
```

In [72]:

```
X3 = np.hstack([Xs, np.ones((X.shape[0],1))])
```

In [73]:

```
w = np.linalg.solve(X3.T @ X3, X3.T @ y)
p0 = X3 @ w
pr = y - p0
p = pr**2
RMSE = np.sqrt(p.mean())
RMSE
```

Out[73]:

```
4.735998462783738
```

In [74]:

```
C = columns[:-1]
W = w[:-1]
pd.Series(W,C)
```

Out[74]:

```
CRIM      -1.043097
ZN         1.094220
INDUS      0.092302
CHAS       0.720628
NOX       -2.171487
RM         2.567716
AGE        0.101537
DIS       -3.135992
RAD        2.517429
TAX       -2.135271
PTRATIO   -2.027701
LSTAT     -3.938105
dtype: float64
```

In [ ]:

```

```

Using the same dataset for both model training and evaluation is a bad idea in machine learning because it can lead to **overfitting**. When a model is trained and evaluated on the same dataset, it can achieve high accuracy on that data, but it may not generalize well to new, unseen data. This is because the model has simply memorized the training data instead of learning the underlying patterns that can apply to new data.

Exercice 5: Repeat the previous experiment so that the model is built on a training set and evaluated on a distinct test set!

In [75]:

```
df
```

Out[75]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	LSTAT	MEDV
0	0.00632	18.0	2.31	0	0.538	6.575	65.2	4.0900	1	296.0	15.3	4.98	24.0
1	0.02731	0.0	7.07	0	0.469	6.421	78.9	4.9671	2	242.0	17.8	9.14	21.6
2	0.02729	0.0	7.07	0	0.469	7.185	61.1	4.9671	2	242.0	17.8	4.03	34.7
3	0.03237	0.0	2.18	0	0.458	6.998	45.8	6.0622	3	222.0	18.7	2.94	33.4
4	0.06905	0.0	2.18	0	0.458	7.147	54.2	6.0622	3	222.0	18.7	5.33	36.2
...	...	...	...	...	...	...	...	...	...	...	...	...	...
501	0.06263	0.0	11.93	0	0.573	6.593	69.1	2.4786	1	273.0	21.0	9.67	22.4
502	0.04527	0.0	11.93	0	0.573	6.120	76.7	2.2875	1	273.0	21.0	9.08	20.6
503	0.06076	0.0	11.93	0	0.573	6.976	91.0	2.1675	1	273.0	21.0	5.64	23.9
504	0.10959	0.0	11.93	0	0.573	6.794	89.3	2.3889	1	273.0	21.0	6.48	22.0
505	0.04741	0.0	11.93	0	0.573	6.030	80.8	2.5050	1	273.0	21.0	7.88	11.9

506 rows × 13 columns

In [77]:

```
n = df.shape[0]
rs = np.random.RandomState(42)
idxs = rs.permutation(n)
s = int(n*0.7)
tr = idxs[:s] # indices of the training set
te = idxs[s:] # indices of the test set
```

In [78]:

```
len(tr), len(te)
```

Out[78]:

```
(354, 152)
```

In [79]:

```
X2 = np.hstack([X, np.ones((X.shape[0], 1))])
y = df['MEDV'].values
w = np.linalg.solve(X2[tr].T @ X2[tr], X2[tr].T @ y[tr]) # optimal parameter vector
p = X2 @ w                          # prediction
RMSE_te = ((p[te] - y[te])**2).mean()**0.5       # root mean squared error
RMSE_te
```

Out[79]:

```
4.82601778299211
```

In [ ]:

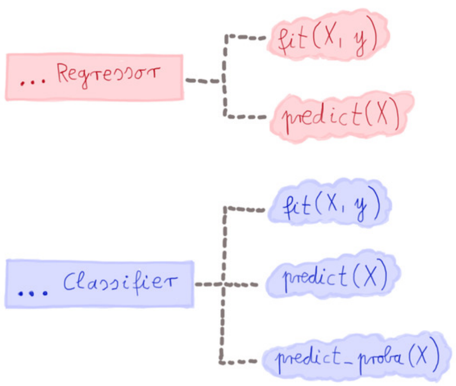
```

```

[scikit-learn (https://scikit-learn.org/stable/)](https://scikit-learn.org/stable/)

Scikit-learn is a popular open-source machine learning library in Python. It provides a range of supervised and unsupervised learning algorithms for tasks such as classification, regression, clustering, and dimensionality reduction. Scikit-learn also includes tools for model selection, preprocessing, and evaluation, making it a comprehensive library for building and evaluating machine learning models.

- scikit-learn is based on NumPy, SciPy and matplotlib.
- The import name of the package is `sklearn`.
- Regressors and classifiers in scikit-learn always have a `fit()` and a `predict()` method.
- The `fit()` methods requires 2 parameters: the input matrix X and a target vector y. Calling the `fit()` method trains a model on the given data.
- The `predict()` method requires an input matrix X and returns the prediction of the trained model for the given inputs.



Execrice 6: Repeat the previous experiments using scikit-learn!

In [80]:

```
# Query the version number of scikit-learn.
import sklearn
sklearn.__version__
```

Out[80]:

```
'1.2.1'
```

In [81]:

```
# This is how we could create a train-test split with scikit-learn.
# However, we will keep using the original split to make the results comparable.

from sklearn.model_selection import ShuffleSplit
tr2, te2 = next(ShuffleSplit(random_state=42, test_size=0.3).split(X))
```

In [85]:

```
# In scikit-learn, the closest thing to RMSE is mean_squared_error.
from sklearn.metrics import mean_squared_error as mse
from sklearn.linear_model import LinearRegression
```

In [86]:

```
# Univariate models.
y = df['MEDV'].values

for column in columns[:-1]:
    x = df[[column]].values
    re = LinearRegression()
    re.fit(x, y)
    p = re.predict(x)
    RMSE = mse(y, p)**0.5
    # w = (x @ y) / (x @ x) # optimal model parameter
    # p = x * w           # prediction
    # RMSE = ((p - y)**2).mean()**0.5 # root mean squared error

    print(column, RMSE)
```

```
CRIM 8.467038200100824
ZN 8.570396495772854
INDUS 8.04153105080589
CHAS 9.045800910882107
NOX 8.306881987569504
RM 6.603071389222561
AGE 8.510228018625199
DIS 8.896422965780747
RAD 8.492632800301259
TAX 8.117097716353987
PTRATIO 7.915314271320455
LSTAT 6.20346413142642
```

In [88]:

```
# Multivariate model without bias, no train-test split.
X = df[columns[:-1]].values
y = df['MEDV'].values
re = LinearRegression(fit_intercept=False)
re.fit(X, y)
p = re.predict(X)
RMSE = mse(y, p)**0.5
print(RMSE)
```

```
5.065952606279903
```

In [89]:

```
# Multivariate model with bias.
X = df[columns[:-1]].values
y = df['MEDV'].values
re = LinearRegression()
re.fit(X[tr], y[tr])
p = re.predict(X)
RMSE = mse(y[te], p[te])**0.5
print(RMSE)
```

```
4.826017782992097
```

In [ ]:

```

```