COSC 2753 | Machine Learning

Week 3 Lab Exercises: **Training a Regression Model**

Introduction

Last week we learned how to read the data and do exploratory data analysis (EDA). The next step in a typical machine learning pipeline is to split data and transform so that we can feed the data to a learning algorithm.

The lab assumes that you have completed `Week 02 lab: Reading data & Exploratory Data Analysis (EDA)`. If you haven't yet, please do so before attempting this lab.

The lab can be executed on either your own machine (with anaconda installation) or computer lab.

- Please refer canvas for instructions on installing anaconda python

Objective

- Continue to familiarise with Python and other ML packages
- Learn to split the data to training/validation and test sets
- Important considetations in splitting the data
- Learn to train a model for regression and complete the introduction to model development pipeline.

Dataset

We examine two regression based datasets in this lab. The first one is to do with house prices, some factors associated with the prices and trying to predict house prices. The second dataset is predicting the amount of share bikes hired every day in Washington D.C., USA, based on time of the year, day of the week and weather factors. These datasets are available in `housing.data.csv` and `bikeShareDay.csv` in the code repository.

First, ensure the two data files are located within the Jupyter workspace.

- If you are on the local machine copy the two data data directories (`BostonHousingPrice`,`Bike-Sharing-Dataset`) to your current folder.

In this course we mostly use datasets that are collected by a third party. If you are interested in collecting your own data for your project, some useful information can be found at: Introduction to Constructing Your Dataset

Problem Formulation

The first step in developing a model is to formulate the problem in a way that we can apply machine learning. To reiterate, the `task` in the Boston house price dataset is to predict the house price (`MEDV`), using some attributes of the house and neighbourhood.

� Observe the data and see if there is a pattern that would allow us to predict the house price using the attributes given? You can use the observations from the EDA for this.

� What category does the task belong to?

> ✔ **Task category:**
>
> - supervised, univariate/multivariate regression

> - We should use the insights gained from observing the data (EDA) in selecting the performance measure. e.g. are there outliers in target?

Data Splitting

As we have discussed in the lecture, in supervised learning we are interested in learning a model using our dataset that can predict the target value for unseen data (Not in the training set). This is called **generalization**. How can we test if the model we developed with our training data would generalize? One approach we can use is to **hold some data from the training process** (Hypothetical unseen data). This hold out data subset (split) is called the `"Test set"` and the remaining data is called the `”Training set"`. The training set may be further subdivided, but more on this later in the regularization lecture. We can use the "Test set" at the end of the development phase to test our model and see if it generalizes.

- **Training set:** Is applied to train, or fit, your model. For example, you use the training set to find the optimal weights, or coefficients, for linear regression, logistic regression, or neural networks.
- **Test set:** Needed for an unbiased evaluation of the final model.

> ⚠ **Warning: The test set should be independent and identically distributed with respect to the training data**
>
> - Should make sure that there is no leakage between the two sets (overlapped train and test instances). This will give unrealistically high performance metric values for your model. e.g. In house price prediction, there may be a house that was sold multiple times and, you might include some instances of this house in the train set and some in the test set. This will result in data leakage.
> - There should be no underlying differences between the two distributions. In other words the characteristics of the test set should not be different to that of the train set. For example all the houses sold in winter in train set and all the houses sold on summer in another set (generally, there is a difference in house prices sold in winter vs summer).
> - More on this in the lectures.

> ⚠ **Warning: The test data should NOT be used for any aspect of the model development process (training).**
> This includes hyper parameter tuning and model selection (a separate validation set should be used for them).

Random splitting

In machine learning, the most common approach taken to split the dataset in to do random sampling (random split). In random sampling we allocate some data instances (selected randomly) to train set and the other instances to test set. One key configuration parameter in this process: what should be the size of the train set and test set respectively. This is most commonly expressed as a percentage between 0 and 1 for either the train or test datasets. For example, a training set with the size of 0.67 (67 percent) means that the remainder percentage 0.33 (33 percent) is assigned to the test set.

There is no optimal split percentage.

You must choose a split percentage that meets your project's objectives with considerations that include:

- Computational cost in training/evaluation the model.
- Training set/Test set representativeness.

Nevertheless, common split percentages include:

- Train: 80%, Test: 20%
- Train: 67%, Test: 33%
- Train: 50%, Test: 50%

Lets first load the house price dataset.
�Use the knowledge from the last week to load the dataset into a pandas dataframe named `bostonHouseFrame`.

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np

## TODO
bostonHouseFrame = pd.read_csv("housing.data.csv", delimiter="\s+")
print(bostonHouseFrame)

```

```
         CRIM    ZN  INDUS  CHAS    NOX     RM   AGE     DIS  RAD    TAX  \
0     0.00632  18.0   2.31     0  0.538  6.575  65.2  4.0900    1  296.0   
1     0.02731   0.0   7.07     0  0.469  6.421  78.9  4.9671    2  242.0   
2     0.02729   0.0   7.07     0  0.469  7.185  61.1  4.9671    2  242.0   
3     0.03237   0.0   2.18     0  0.458  6.998  45.8  6.0622    3  222.0   
4     0.06905   0.0   2.18     0  0.458  7.147  54.2  6.0622    3  222.0   
..        ...   ...    ...   ...    ...    ...   ...     ...  ...    ...   
501   0.06263   0.0  11.93     0  0.573  6.593  69.1  2.4786    1  273.0   
502   0.04527   0.0  11.93     0  0.573  6.120  76.7  2.2875    1  273.0   
503   0.06076   0.0  11.93     0  0.573  6.976  91.0  2.1675    1  273.0   
504   0.10959   0.0  11.93     0  0.573  6.794  89.3  2.3889    1  273.0   
505   0.04741   0.0  11.93     0  0.573  6.030  80.8  2.5050    1  273.0   

     PTRATIO       B  LSTAT  MEDV  
0       15.3  396.90   4.98  24.0  
1       17.8  396.90   9.14  21.6  
2       17.8  392.83   4.03  34.7  
3       18.7  394.63   2.94  33.4  
4       18.7  396.90   5.33  36.2  
..       ...     ...    ...   ...  
501     21.0  391.99   9.67  22.4  
502     21.0  396.90   9.08  20.6  
503     21.0  396.90   5.64  23.9  
504     21.0  393.45   6.48  22.0  
505     21.0  396.90   7.88  11.9  

[506 rows x 14 columns]
```

The scikit-learn Python machine learning library provides an implementation of the train-test splitting via function `train_test_split()`. Lets use this to randomly split our data to 80% train set and 20% test set.

```
from sklearn.model_selection import train_test_split

with pd.option_context('mode.chained_assignment', None):
    bostonHouseTrainFrame, bostonHouseTestFrame = train_test_split(bostonHouseFrame, test_size=0.2, shuffle=True)

```

```
print("Nunber of instances in the original dataset is {}. After spliting Train has {} instances and test has {} instances."
      .format(bostonHouseFrame.shape[0], bostonHouseTrainFrame.shape[0], bostonHouseTestFrame.shape[0]))
```

```
Nunber of instances in the original dataset is 506. After spliting Train has 404 instances and test has 102 instances.
```

Checking your splits

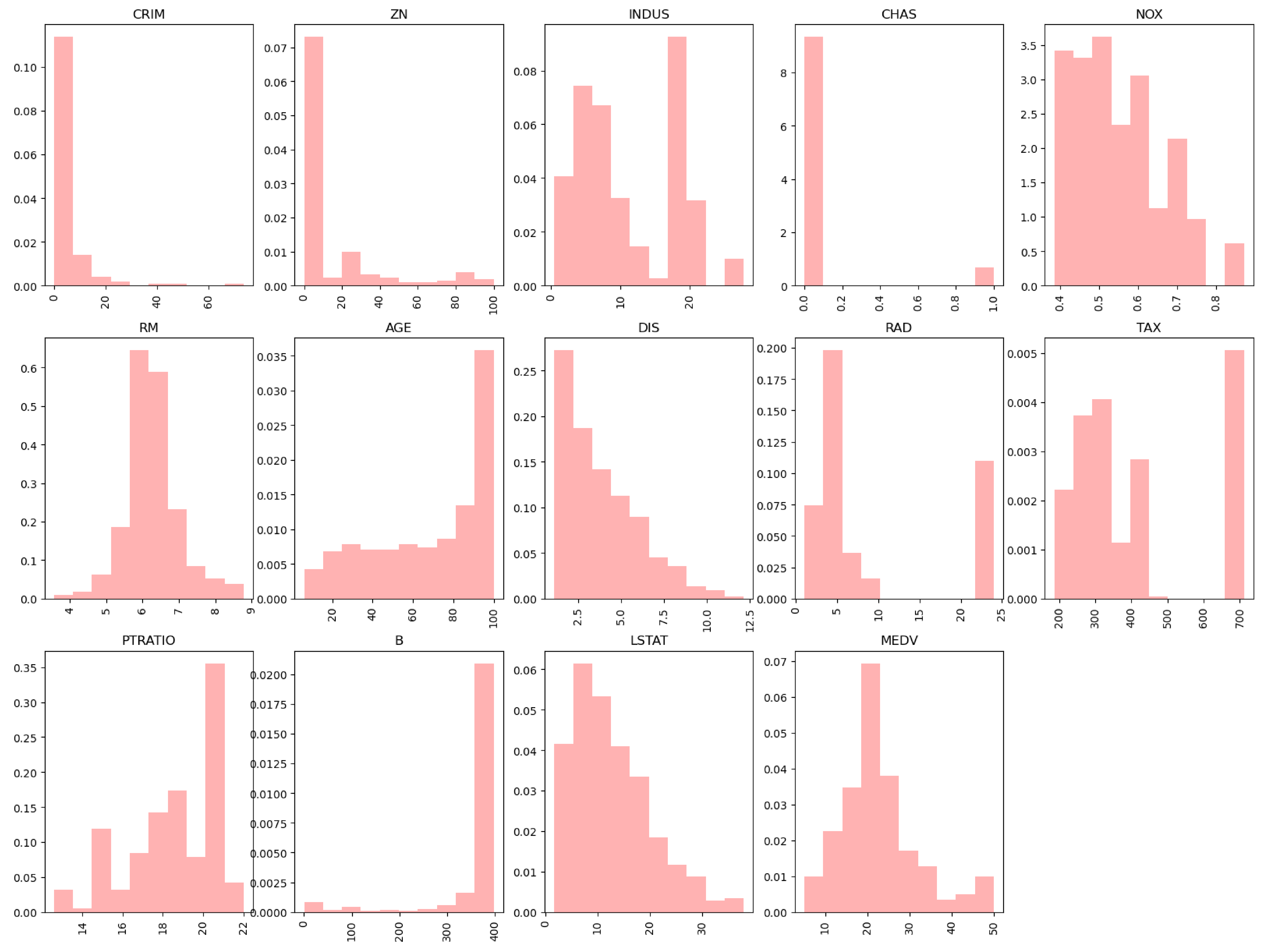
As discussed, random splitting may lead to leakage (two splits are not independent). We need to understand the dataset to make sure there are no hidden sources of leakage in the data. This is one place we can use the knowledge we gained through the EDA.

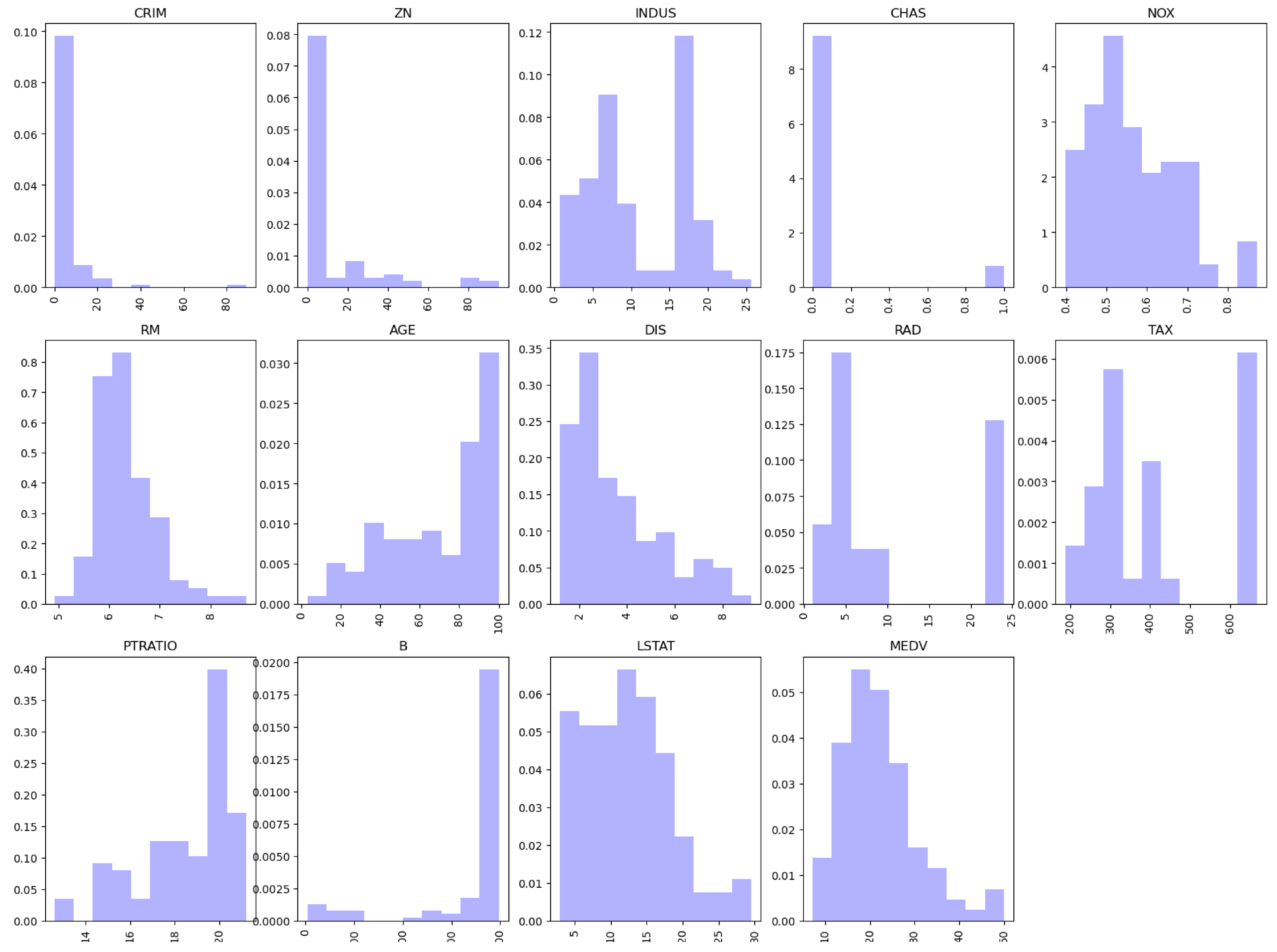
> ✑ **Task: Use the EDA observations from last week to see if there is any issue with the random splitting process.**

We can use histogram plots to see if the two partitions (splits) are identically distributed.

� Use the knowledge from last week to plot the histograms for each attribute for the two splits. Use different colors for test vs train

```
## TODO
plt.figure(figsize=(20,20))
for i, col in enumerate(bostonHouseTrainFrame.columns):
    plt.subplot(4,5,i+1)
    plt.hist(bostonHouseTrainFrame[col], alpha=0.3, color='r', density=True)
    plt.title(col)
    plt.xticks(rotation='vertical')\

plt.figure(figsize=(20,20))
for i, col in enumerate(bostonHouseTestFrame.columns):
    plt.subplot(4,5,i+1)
    plt.hist(bostonHouseTestFrame[col], alpha=0.3, color='b', density=True)
    plt.title(col)
    plt.xticks(rotation='vertical')

```


CRIM
ZN
INDUS
CHAS
NOX
RM
AGE
DIS
RAD
TAX
PTRATIO
B
LSTAT
MEDV


CRIM
ZN
INDUS
CHAS
NOX
RM
AGE
DIS
RAD
TAX
PTRATIO
B
LSTAT
MEDV

This Python code is using the matplotlib library to create histograms for each column in two different dataframes: `bostonHouseTrainFrame` and `bostonHouseTestFrame`. These dataframes presumably contain data related to houses in Boston, split into a training set and a test set.

The code starts by creating a new figure with a size of 20x20 using `plt.figure(figsize=(20,20))`. Then, it enumerates over each column in the `bostonHouseTrainFrame` dataframe. For each column, it creates a subplot in a 4x5 grid (using `plt.subplot(4,5,i+1)`) and then plots a histogram of the data in that column with `plt.hist()`. The `alpha` parameter is set to 0.3, which controls the transparency of the histogram, and the `color` parameter is set to 'r', which means the histogram will be red. The `density` parameter is set to True, which means the histogram will show the density (normalized count) instead of the actual count. The title of each subplot is set to the name of the column with `plt.title(col)`, and the x-axis labels are rotated vertically with `plt.xticks(rotation='vertical')`.

The same process is then repeated for the `bostonHouseTestFrame` dataframe, but the color of the histograms is set to blue ('b').

This code is useful for visualizing the distribution of values in each column of the dataframes, which can help with understanding the data and checking for any anomalies or patterns.

� What observations did you make?

> ✔ **Observations:**
>
> - The distribution of train set attributes is approximately equal to the distribution of test set attributes.

> ⚠ **Warning: Make sure you use the same bins for both plots (test/train)**

Now you know how to randomly split the data. Random splitting is not the only method to split data. The method used may vary based on many factors like problem type, nature of data etc. For example if we have time series data, we can use TimeSeriesSplit. It is also common in ML to write your own custom function to do data splitting where special measures are required to keep the data independent or identical.

Univariate Regression

We will first study how to do univariate regression.

If you recall from the last lab, we found that possibly the 'RM' (number of rooms) and 'LSTAT' (unsure) variables seem to have a linear relationship with the house price ('MEDV'). Hence, we will try these variables as the independent variable to predict the house price, i.e., the dependent variable.

- Create an independent variable that is just based on the 'RM' column and a dependent variable of 'MEDV'.
- Assign the values of the 'RM' column to a variable named 'house_uniRM_x' and assign the values of the 'MEDV' column to a variable named 'house_y'.

� There are at least three ways to create the house_uniRM_x and house_y variables based on those columns. Do some quick online research to slice the bostonHousePrice dataset and extract the two columns, by using the following methods:

- (1) using square brackets []
- (2) using the pandas function .loc
- (3) using the pandas function .iloc

```
#print(bostonHouseFrame)
#house_uniRM_x = bostonHouseFrame.loc[:,'RM']
#house_uniRM_x= house_uniRM_x.values.reshape(-1,1)
house_uniRM_x = bostonHouseFrame[['RM']]
house_y = bostonHouseFrame['MEDV']
```

```
print(house_y)
print(house_uniRM_x.shape)
```

```
0      24.0
1      21.6
2      34.7
3      33.4
4      36.2
       ...
501    22.4
502    20.6
503    23.9
504    22.0
505    11.9
Name: MEDV, Length: 506, dtype: float64
(506, 1)
```

Next, we want to create some data to train the model, and some other data to evaluate how good the model is. We may be tempted to use 100% of the data for training, then select a subset for evaluation (say 20% of the data). However, we will find out later that this isn't a good idea generally, as the data we use to test is the same we use to train, and likely to cause overfitting and what is called bias. We will discuss this more in a later week, but for now, just note that it isn't a good idea to do so.

Instead, we will split the data into a training set, which we use to fit the model/hypothesis, and a testing set, which we will use to evaluate the performance of the fitted model/hypothesis. There should be no overlap between the two sets. How we achieve this is typically randomly split the data, say 80% for training and 20% for testing. We can do this ourselves, but like many machine learning functionality these days, they are already implemented, and of course, they are available in the libraries we have imported.

```
# create testing and training data for RM variable
#from sklearn.model_selection import train_test_split
trainX, testX, trainY, testY = train_test_split(house_uniRM_x, house_y, test_size=0.2,shuffle=True)
print(trainX.shape)
print(testX.shape)
print(trainY.shape)
print(testY.shape)
```

```
(404, 1)
(102, 1)
(404,)
(102,)
```

The function that does the work is ‘train_test_split()’, part of sklearn.model_selection package. It essentially does what we desire, with the first two arguments to it are the X and Y variable datasets respectively (they must be of the same number of rows/instances), and a test_size parameter which specifies how big the test dataset is (in this case, 20% of the data, which is a typical setting for this). This function returns ‘trainX’ (training data of the X variable), ‘testX’ (testing data of X), ‘trainY’ (training data of Y) and ‘testY’ (testing data of Y). The last four statements just print out the size of the resulting training and testing datasets to show you that the training (test) datasets contain the same number of rows. We will use trainX and trainY to train the linear regression model, then textX and testY for testing and evaluation.

We are now ready to construct the linear regression model/hypothesis and fit the theta parameters.

```
from sklearn import linear_model
linReg = linear_model.LinearRegression()
```

This constructs the linear regression model object, assigned to variable ‘linReg’. We then fit the training data to the model:

```
linReg.fit(trainX, trainY)
```

```
▾ LinearRegression
LinearRegression()
```

linReg.fit() fits the X and Y training data and optimises the parameters to minimise the loss function. It might not exactly use gradient descent, but the ideas are similar and as stated in lectures, gradient descent as a general optimisation is the crux of many optimisation and parameter fitting algorithms.

Let's have a look at what the parameters look like:

```
print(linReg.intercept_)
print(linReg.coef_)
```

```
-37.51187405483272
[9.53058523]
```

linReg.intercept_ is the y-intercept, or essentially the theta parameter.

linReg.coef_ is the slope of the univariate linear regression line or the theta_one variable.

With a model/hypothesis, we can now do prediction! We use the testing data for that:

```
pred_uniRM_y = linReg.predict(testX)
```

Okay, we now have predictions, but how did we go? What we want to do is to compare the predicted value from our model (given testX) with the actual Y values (testY). We can estimate the error (using the mean squared error loss function we been discussing in lectures for fitting the parameters), as follows:

```
from sklearn.metrics import mean_squared_error
print('Mean squared error ', mean_squared_error(testY, pred_uniRM_y))
```

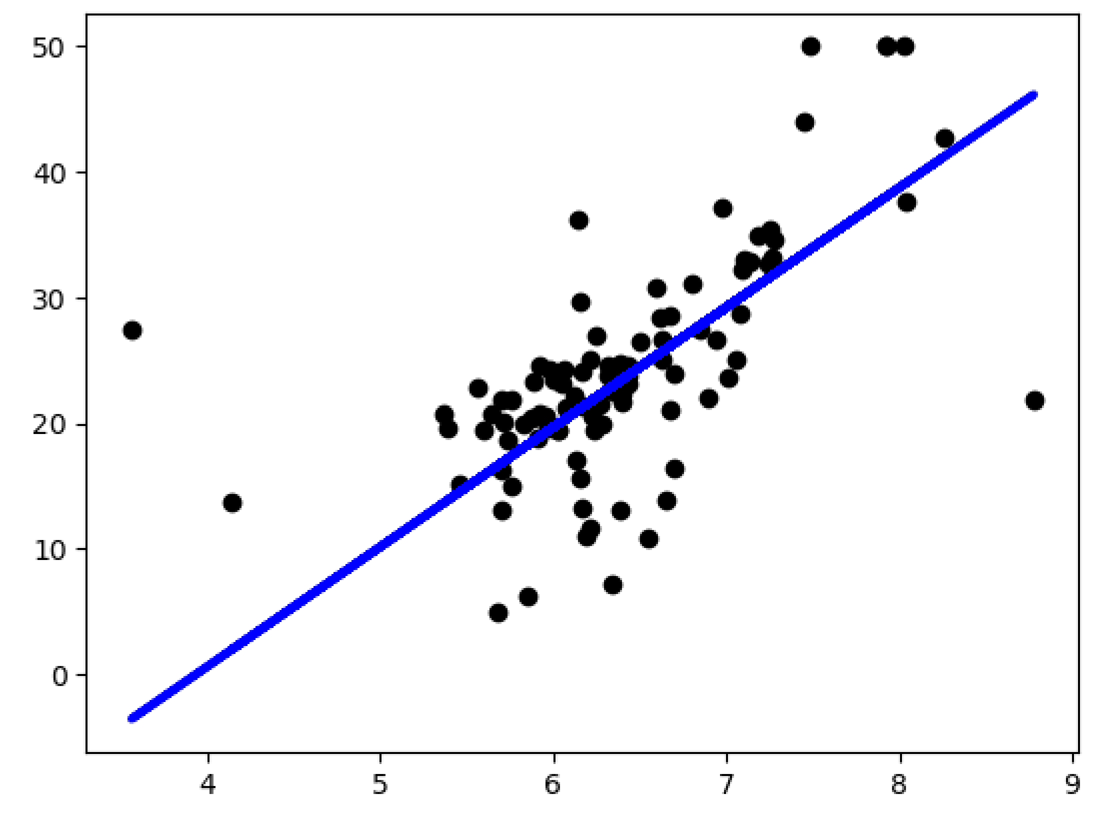
```
Mean squared error  48.415976128209145
```

What is the mean squared error value you obtained?

In addition, it is useful for regression to plot the testing data against the model/hypothesis, which is a line in our univariable linear regression. Type in the following:

```
plt.scatter(testX, testY, color='black')
plt.plot(testX, pred_uniRM_y, color='blue', linewidth=3)
```

```
[<matplotlib.lines.Line2D at 0x204e909b510>]
```



The provided Python code is using the matplotlib library to create a scatter plot and a line plot on the same figure.

The first line of code, `plt.scatter(testX, testY, color='black')`, is creating a scatter plot. The `testX` and `testY` variables are the x and y coordinates of the points in the scatter plot, respectively. The `color` parameter is set to 'black', which means the points will be black.

The second line of code, `plt.plot(testX, pred_uniRM_y, color='blue', linewidth=3)`, is creating a line plot. The `testX` variable is the x coordinates of the points on the line, and `pred_uniRM_y` is the y coordinates. The `color` parameter is set to 'blue', which means the line will be blue. The `linewidth` parameter is set to 3, which controls the thickness of the line.

The scatter plot and line plot are overlaid on the same figure because the code does not create a new figure or subplot between the two calls to `plt.scatter()` and `plt.plot()`. This can be useful for comparing the actual values (represented by the scatter plot) with the predicted values (represented by the line plot).

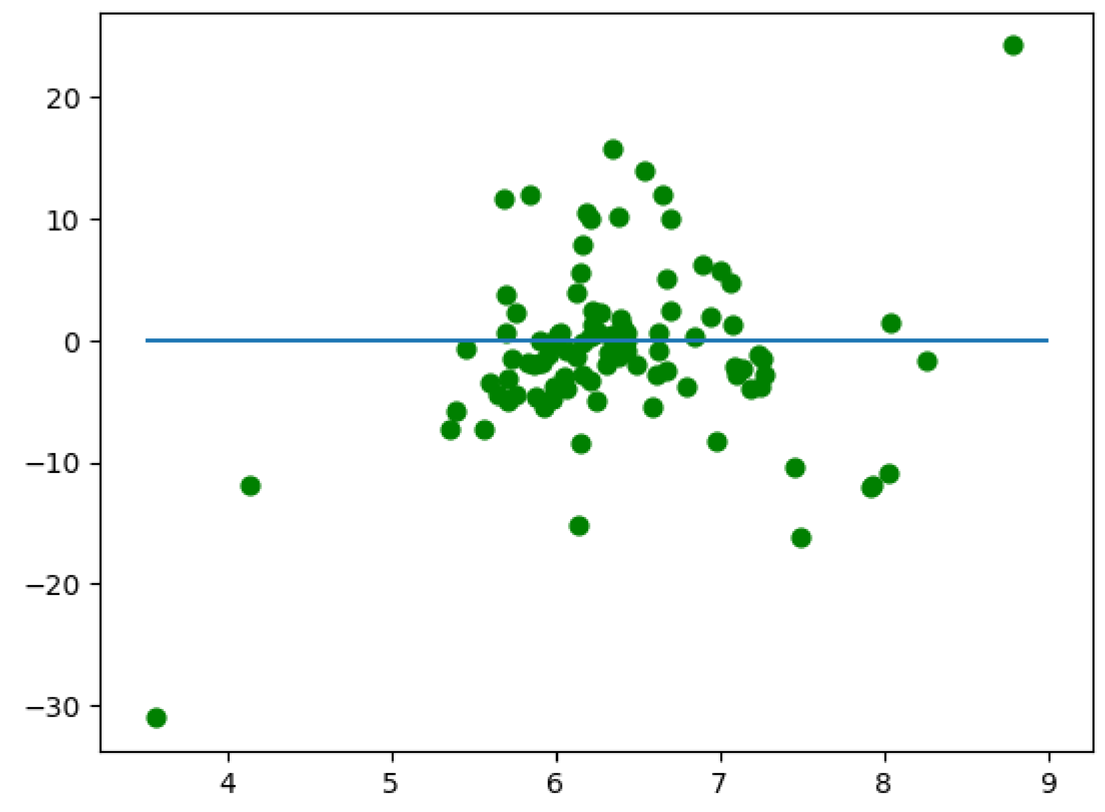
The functions `scatter()` and `plot()` are part of the matplotlib library's pyplot module, which provides a MATLAB-like interface for creating plots. The `scatter()` function creates a scatter plot, and the `plot()` function creates a line plot. Both functions take in x and y coordinates as their first two arguments, and they have many optional parameters for customizing the appearance of the plot.

This uses our reliable plotting package matplotlib and pyplot to first, produce a scatterplot of our testing data (X,Y) pairs, then draw a blue line for our model/hypothesis. What information does the plot visualise?

Another interesting visualisation and based on the same information is the residual plot, which shows what is the difference in predicted and actual values, across different X values, for the testing data;

```
plt.scatter(testX, linReg.predict(testX) - testY, c='g', s=40)
plt.hlines(y=0, xmin=3.5, xmax=9)
```

```
<matplotlib.collections.LineCollection at 0x204df8a6590>
```



This Python code is using the matplotlib library to create a scatter plot and a horizontal line. The scatter plot is used to visualize the residuals of a linear regression model, and the horizontal line is used as a reference line at y=0.

The first line of code, `plt.scatter(testX, linReg.predict(testX) - testY, c='g', s=40)`, is creating a scatter plot. The `testX` variable is the x coordinates of the points in the scatter plot. The y coordinates are calculated by predicting the y values using the linear regression model `linReg` and the test data `testX`, and then subtracting the actual y values `testY`. This gives the residuals, which are the differences between the predicted and actual values. The `c` parameter is set to 'g', which means the points will be green. The `s` parameter is set to 40, which controls the size of the points.

The second line of code, `plt.hlines(y=0, xmin=3.5, xmax=9)`, is creating a horizontal line at y=0. The `xmin` and `xmax` parameters are set to 3.5 and 9, respectively, which means the line will start at x=3.5 and end at x=9.

This plot is useful for checking the assumptions of a linear regression model. If the model is a good fit for the data, the residuals should be randomly scattered around y=0, and there should be no clear pattern. If there is a pattern, it suggests that the model is not capturing some aspect of the data.

If the model is perfect, then for each testing data then there is 0 residual (or 0 error between predicted versus actual), which is represented by the horizontal line in the diagram. If the model under-estimates the actual value, it will be below this horizontal line, and if over-estimates, than above. This plot can quickly give you a sense of where the errors are occurring and whether there are outlier points that are causing large errors (we will examine this later in the course and ways to deal with this).

> **☞ With your dataframe slicing skill, can you count the number of residuals that are larger than zero and those that are smaller than zero?

```
positive_residuals = (linReg.predict(testX) - testY) > 0
negative_residuals = (linReg.predict(testX) - testY) < 0

num_positive_residuals = sum(positive_residuals)
num_negative_residuals = sum(negative_residuals)

print("Number of residuals larger than zero:", num_positive_residuals)
print("Number of residuals smaller than zero:", num_negative_residuals)
```

```
Number of residuals larger than zero: 43
Number of residuals smaller than zero: 59
```

This Python code is calculating and printing the number of positive and negative residuals from a linear regression model. Residuals are the differences between the predicted and actual values, and they can be used to assess the fit of the model.

The first two lines of code are creating boolean arrays `positive_residuals` and `negative_residuals`. The `linReg.predict(testX)` function is used to predict the y values using the linear regression model `linReg` and the test data `testX`. The actual y values `testY` are then subtracted from these predicted values to get the residuals. The `>` and `<` operators are used to create boolean arrays where each element is `True` if the corresponding residual is greater than or less than zero, respectively, and `False` otherwise.

The next two lines of code are calculating the number of positive and negative residuals by summing the boolean arrays. In Python, `True` is equivalent to 1 and `False` is equivalent to 0, so the `sum()` function can be used to count the number of `True` values in a boolean array.

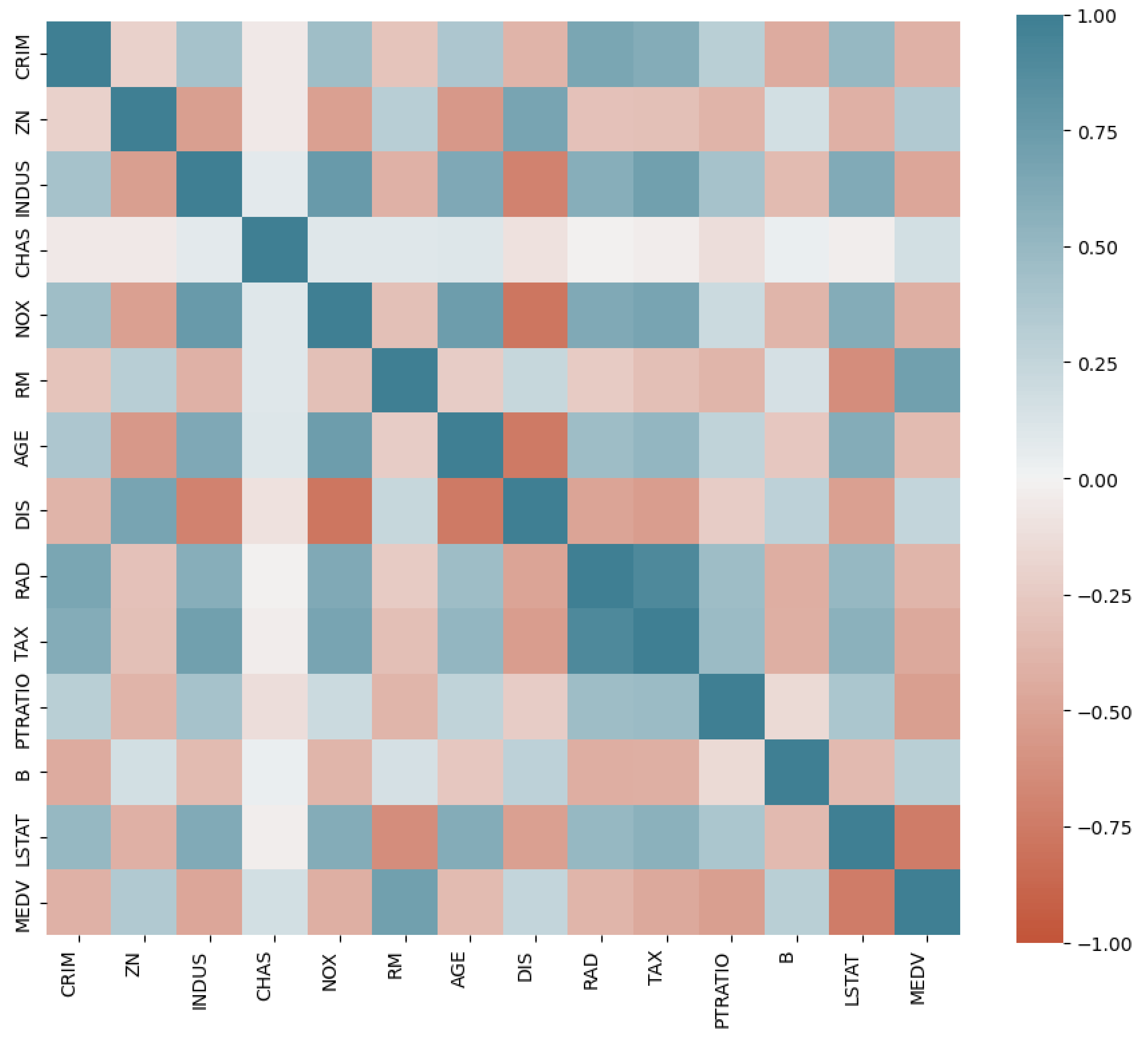
Finally, the `print()` function is used to print the number of positive and negative residuals. This information can be useful for understanding the distribution of residuals, which can help with assessing the fit of the model. For example, if there are significantly more positive residuals than negative residuals, it might indicate that the model is systematically underestimating the y values.

Further exercises

We have fitted a linear regression model/hypothesis for the 'RM' variable. Use the seaborn package to visualise the correlation plot and identify the next potential predictor of the house price. You should identify 'LSTAT' as the next potential predictor. Why is it?

Repeat the above analysis using the 'LSTAT' variable, and comment on which predictor you think is performing better. LSTAT variable has a negative (inverse) linear relationship with 'MEDV', where when LSTAT increases MEDV decreases – do you think it matters?

```python
import seaborn as sns

f, ax = plt.subplots(figsize=(11, 9))
corr = bostonHouseTrainFrame.corr()
ax = sns.heatmap(
    corr,
    vmin=-1, vmax=1, center=0,
    cmap=sns.diverging_palette(20, 220, n=200),
    square=True
)
ax.set_xticklabels(
    ax.get_xticklabels(),
    rotation=90,
    horizontalalignment='right'
);
```



This Python code is using the seaborn and matplotlib libraries to create a heatmap of the correlation matrix for a dataframe `bostonHouseTrainFrame`.

The first line of code imports the seaborn library, which is a Python data visualization library based on matplotlib. It provides a high-level interface for drawing attractive and informative statistical graphics.

The second line of code creates a new figure and a single subplot. The `plt.subplots(figsize=(11, 9))` function is used to create the figure and subplot, and the `figsize` parameter is set to (11, 9), which controls the size of the figure.

The third line of code calculates the correlation matrix of the `bostonHouseTrainFrame` dataframe. The `corr()` function is used to compute pairwise correlation of columns, excluding NA/null values. The correlation coefficient ranges from -1 to 1. A value closer to 1 implies a strong positive correlation and a value closer to -1 implies a strong negative correlation.

The next block of code creates a heatmap of the correlation matrix using the seaborn function `sns.heatmap()`. The `vmin` and `vmax` parameters are set to -1 and 1, respectively, which set the color scale limits. The `center` parameter is set to 0, which centers the colormap at that value. The `cmap` parameter is set to a diverging palette, which means the colors diverge from the center value to the `vmin` and `vmax` values. The `square` parameter is set to `True`, which means each cell will be square-shaped.

The final block of code sets the x-axis tick labels. The `ax.get_xticklabels()` function is used to get the current x-axis tick labels, and the `rotation` parameter is set to 90, which means the labels will be rotated 90 degrees. The `horizontalalignment` parameter is set to 'right', which means the labels will be aligned to the right.

This heatmap is useful for quickly visualizing the relationships between each pair of variables. For example, if two variables have a strong positive correlation, their corresponding cell in the heatmap will be a bright color, and if they have a strong negative correlation, the cell will be a dark color.

Why 'LSTAT' as a Predictor?

Statistical Reasoning: Often in housing datasets, the socio-economic status of the neighborhood, represented by 'LSTAT', has a significant impact on house prices. Lower status of the population in an area could inversely affect the median value of houses, as these areas might be less desirable.

Negative Correlation with 'MEDV': A negative (inverse) relationship between 'LSTAT' and 'MEDV' suggests that as 'LSTAT' increases (indicating a higher percentage of lower-status population), 'MEDV' tends to decrease. This relationship can be a strong indicator that 'LSTAT' is a meaningful predictor for 'MEDV'.

We should identify 'LSTAT' as the next potential predictor because the coefficient is near -1

```python
#print(bostonHouseFrame)
#house_uniRM_x = bostonHouseFrame.loc[:,'RM']
#house_uniRM_x= house_uniRM_x.values.reshape(-1,1)
house_uniRM_x = bostonHouseFrame[['LSTAT']]
house_y = bostonHouseFrame['MEDV']
```

```python
# create testing and training data for RM variable
#from sklearn.model_selection import train_test_split
trainX, testX, trainY, testY = train_test_split(house_uniRM_x, house_y, test_size=0.2,shuffle=True)
print(trainX.shape)
print(testX.shape)
print(trainY.shape)
print(testY.shape)
```

```
(404, 1)
(102, 1)
(404,)
(102,)
```

```python
from sklearn import linear_model
linReg = linear_model.LinearRegression()
linReg.fit(trainX, trainY)
```

```
▾ LinearRegression
LinearRegression()
```

```python
print(linReg.intercept_)
print(linReg.coef_)
```

```
34.150741417736576
[-0.94502095]
```

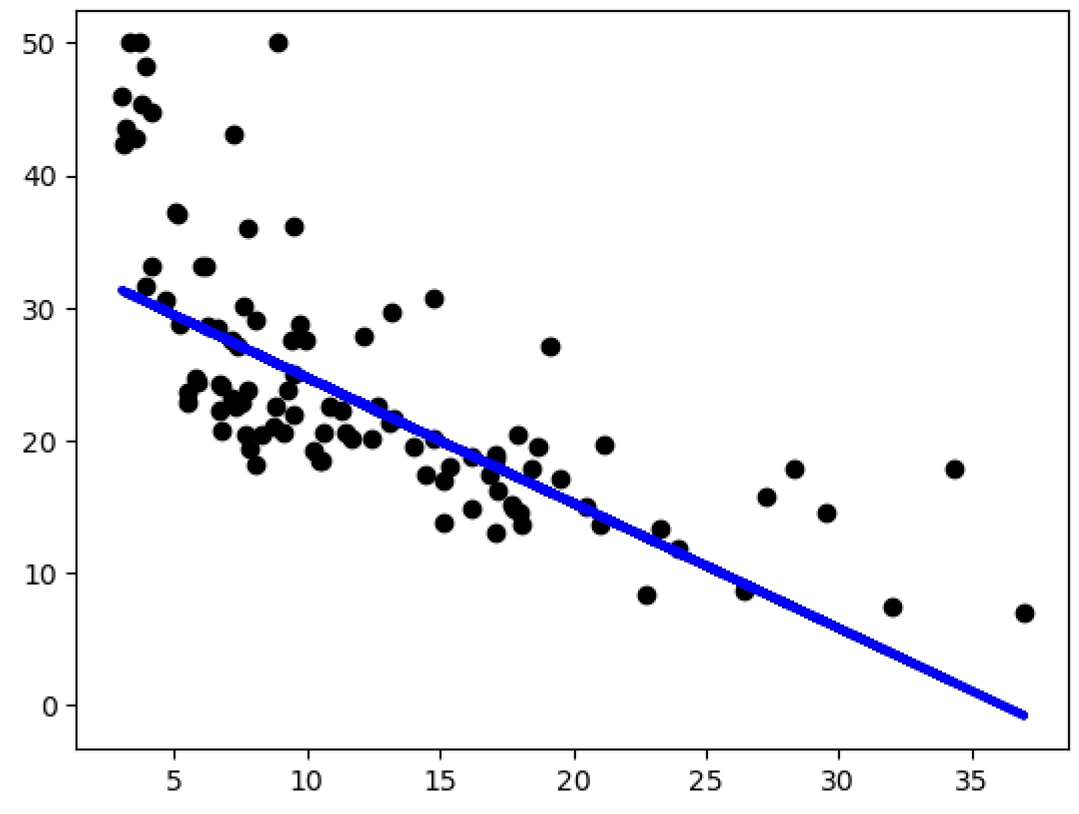
```python
pred_uniRM_y = linReg.predict(testX)

```

```python
from sklearn.metrics import mean_squared_error
print('Mean squared error ', mean_squared_error(testY, pred_uniRM_y))
```

```
Mean squared error  50.5312558706251
```

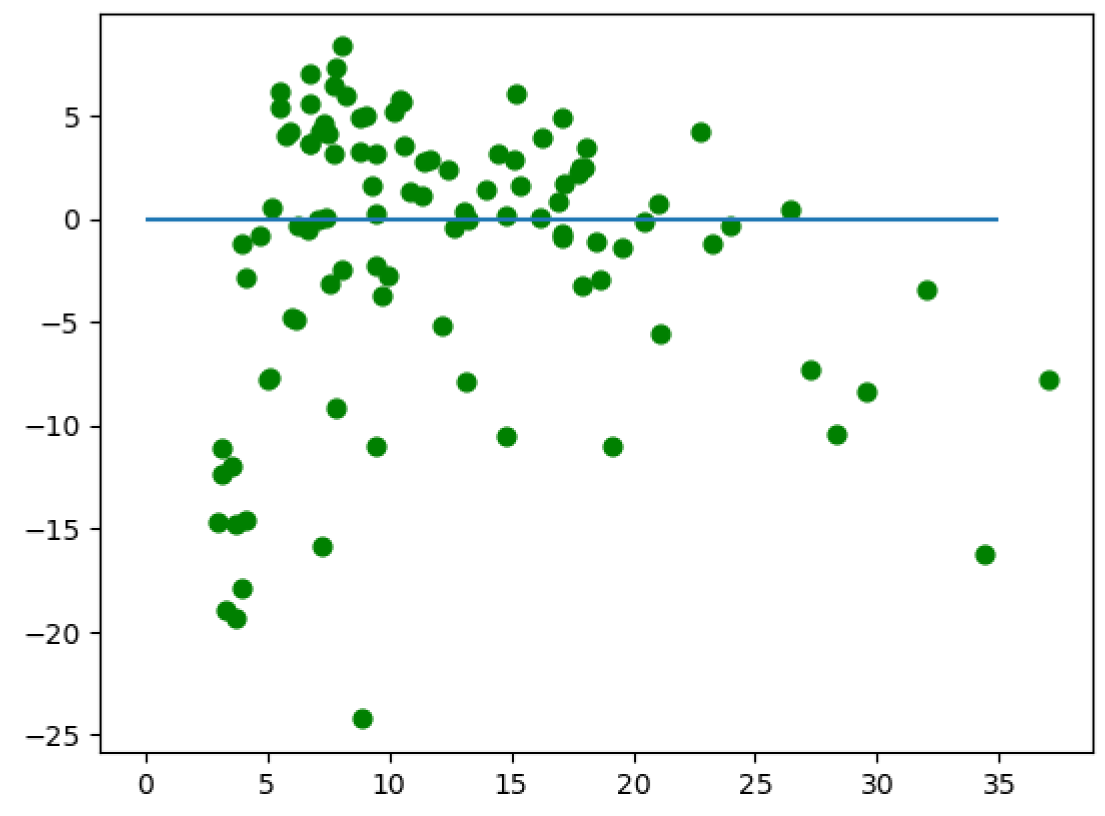
```python
plt.scatter(testX, testY, color='black')
plt.plot(testX, pred_uniRM_y, color='blue', linewidth=3)
```

```
[<matplotlib.lines.Line2D at 0x204e81aed90>]
```



```python
plt.scatter(testX, linReg.predict(testX) - testY, c='g', s=40)
plt.hlines(y=0, xmin=0, xmax=35)
```

```
<matplotlib.collections.LineCollection at 0x204e8162250>
```



```python
positive_residuals = (linReg.predict(testX) - testY) > 0
negative_residuals = (linReg.predict(testX) - testY) < 0

num_positive_residuals = sum(positive_residuals)
num_negative_residuals = sum(negative_residuals)

print("Number of residuals larger than zero:", num_positive_residuals)
print("Number of residuals smaller than zero:", num_negative_residuals)
```

```
Number of residuals larger than zero: 52
Number of residuals smaller than zero: 50
```

Multivariate Regression

So far, we have only used one X variable to predict MEDV. What if we used all the variables? To create such data, we first do the following:

```
house_multi_X = bostonHouseFrame.drop('MEDV', axis=1)
house_multi_Y = bostonHouseFrame[['MEDV']]
```

```
print(house_multi_X.shape)
print(house_multi_Y.shape)
```

```
(506, 13)
(506, 1)
```

```
train_multi_X, test_multi_X, train_multi_Y, test_multi_Y = train_test_split(house_multi_X, house_multi_Y, test_size=0.2,shuffle=True)
```

The Y variable is the same, but the X variable is interesting. We use the drop() method of data frames, which essentially drops a column from it – in our case, we drop the 'MEDV' column. X is now a data frame without the MEDV column, print it out the check.

Now repeat the same analysis, noting that train_test_split(), linReg.fit() and linReg.predict() can work with multivariate linear regression (i.e. where we have multiple X variables) as well. Make sure you print out the intercept and coefficients, print(linReg.intercept_) print(linReg.coef_), to see what are the fits and do the mean squared error.

```
from sklearn import linear_model
multi_linReg = linear_model.LinearRegression()
```

```
history = multi_linReg.fit(train_multi_X, train_multi_Y)
```

```
print(history)
```

```
LinearRegression()
```

```
print(multi_linReg.intercept_)
print(multi_linReg.coef_)
```

```
[40.6244667]
[[-1.23048800e-01  5.08592098e-02  6.42045817e-02  2.26315950e+00
  -2.08378247e+01  3.50782309e+00  1.79264471e-02 -1.42687070e+00
   2.57999699e-01 -9.28333717e-03 -1.04347982e+00  7.27109363e-03
  -6.02029566e-01]]
```

```
pred_uniRM_y = multi_linReg.predict(test_multi_X)

```

```
from sklearn.metrics import mean_squared_error
print('Mean squared error ', mean_squared_error(test_multi_Y, pred_uniRM_y))
```

```
Mean squared error  28.31894705974442
```

To determine which predictor is better (say, comparing 'LSTAT' with another variable like 'CRIM'), you would typically look at several metrics:

Correlation Coefficient: A higher absolute value of the correlation coefficient with 'MEDV' suggests a stronger linear relationship. Regression Metrics: Metrics like R-squared, Mean Squared Error (MSE), or Mean Absolute Error (MAE) in a regression model can indicate how well the model is performing with the predictor.

Exercise: Work on the Bike Share Data

> ✑ **Task: Do the linear regression on the Bike Share Data.**
> Now you seen how to do this task for the house price dataset. Repeat the same process for the Daily Bike Share rental data.

```
# Exercise: Analyze the Bike Share Data

# Import necessary libraries
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

"""
This function reads the Bike Share data from a CSV file,
displays the shape of the dataframe, and returns the dataframe.
"""
# Read the Bike Share data
bike_data = pd.read_csv(r'bikeShareDay-1.csv')

bike_multi_X = bike_data.drop(['cnt', 'dteday'], axis=1)
bike_multi_Y = bike_data[['cnt']]



```

```
print(bike_multi_X.shape)
print(bike_multi_Y.shape)
```

```
(731, 14)
(731, 1)
```

```
"""
Splits the given data into training and testing sets.

Parameters:
- bike_multi_X: The input features for the regression model.
- bike_multi_Y: The target variable for the regression model.

Returns:
- train_multi_X: The training set of input features.
- test_multi_X: The testing set of input features.
- train_multi_Y: The training set of target variable.
- test_multi_Y: The testing set of target variable.
"""
train_multi_X, test_multi_X, train_multi_Y, test_multi_Y = train_test_split(bike_multi_X, bike_multi_Y, test_size=0.2,shuffle=True)
```

```
from sklearn import linear_model
multi_linReg = linear_model.LinearRegression()
```

```
history = multi_linReg.fit(train_multi_X, train_multi_Y)
print(history)
```

```
LinearRegression()
```

```
print(multi_linReg.intercept_)
print(multi_linReg.coef_)
```

```
[-2.72848411e-12]
[[-1.90548246e-15  3.44628285e-13  3.71338614e-12  1.75180116e-13
  -5.83095051e-13  6.49230637e-15  1.29965514e-13 -3.07807150e-13
  -7.83887354e-12  8.62363877e-12 -1.44242690e-13 -6.39370365e-13
   1.00000000e+00  1.00000000e+00]]
```

```
pred_uniRM_y = multi_linReg.predict(test_multi_X)
```

```
from sklearn.metrics import mean_squared_error
print('Mean squared error ', mean_squared_error(test_multi_Y, pred_uniRM_y))
```

```
Mean squared error  2.8050109568749193e-24
```

Some other data pre-processing steps

> **☞ Goal: Do some pre-process steps before fitting the data for training

Feature Scaling

In a typical dataset, you might have different numerical features with widely different ranges. For example during the EDA we discovered that the attribute NOX takes values in the range `[0,1]` whereas TAX takes values in the range `[0, 700]`. Furthermore, you may have some features that have a skewed distribution. Such characteristics in data may sometimes cause problems for the learning algorithms (specially gradient based methods and distance based methods). Therefore it is common to use feature scaling.

> ⚠ **Important: Feature scaling, is usually guided by the EDA.**
> The histograms and other individual feature visualizations often provided useful information for feature scaling and can be used to justify one approach over another.

Two most common methods employed for feature normalization are min-max scaling and standard scaling

- **Min-max scaling:** An individual feature is transformed so that the values are mapped to the range `[0,1]`. Ref.
- **Standard scaling:** An individual feature is transformed so that the transformed values have zero mean and unit variance. Ref.

Lets apply the above two methods to the feature `RM` in boston house price dataset.

```
import pandas as pd
bostonHouseFrame = pd.read_csv("housing.data.csv", delimiter="\s+")
```

```
from sklearn.model_selection import train_test_split

with pd.option_context('mode.chained_assignment', None):
    bostonHouseTrainFrame, bostonHouseTestFrame = train_test_split(bostonHouseFrame, test_size=0.2, shuffle=True)


```

The `train_test_split` function is wrapped in a `with pd.option_context('mode.chained_assignment', None):` block. This is a context manager that temporarily sets the specified pandas option. In this case, it's setting the 'mode.chained_assignment' option to `None`, which means chained assignment warnings will be ignored. Chained assignment is when you assign to a DataFrame subset, which can sometimes lead to unexpected results. By setting this option to `None`, any warnings about this will be suppressed.

```
from sklearn.preprocessing import MinMaxScaler
from sklearn.preprocessing import StandardScaler

MinMaxScaler_RM = MinMaxScaler().fit(bostonHouseTrainFrame[['RM']])
RM_minmax = MinMaxScaler_RM.transform(bostonHouseTrainFrame[['RM']])

StandardScaler_RM = StandardScaler().fit(bostonHouseTrainFrame[['RM']])
RM_standard = StandardScaler_RM.transform(bostonHouseTrainFrame[['RM']])
```

This Python code is using the `MinMaxScaler` and `StandardScaler` classes from the `sklearn.preprocessing` module to scale the 'RM' feature of the `bostonHouseTrainFrame` DataFrame.

The `MinMaxScaler` class provides a way to normalize features by scaling each feature to a given range, typically between zero and one, or so that the maximum absolute value of each feature is scaled to unit size. This type of scaling often is used when the parameters need to be on the same positive scale, but the outliers are not very important.

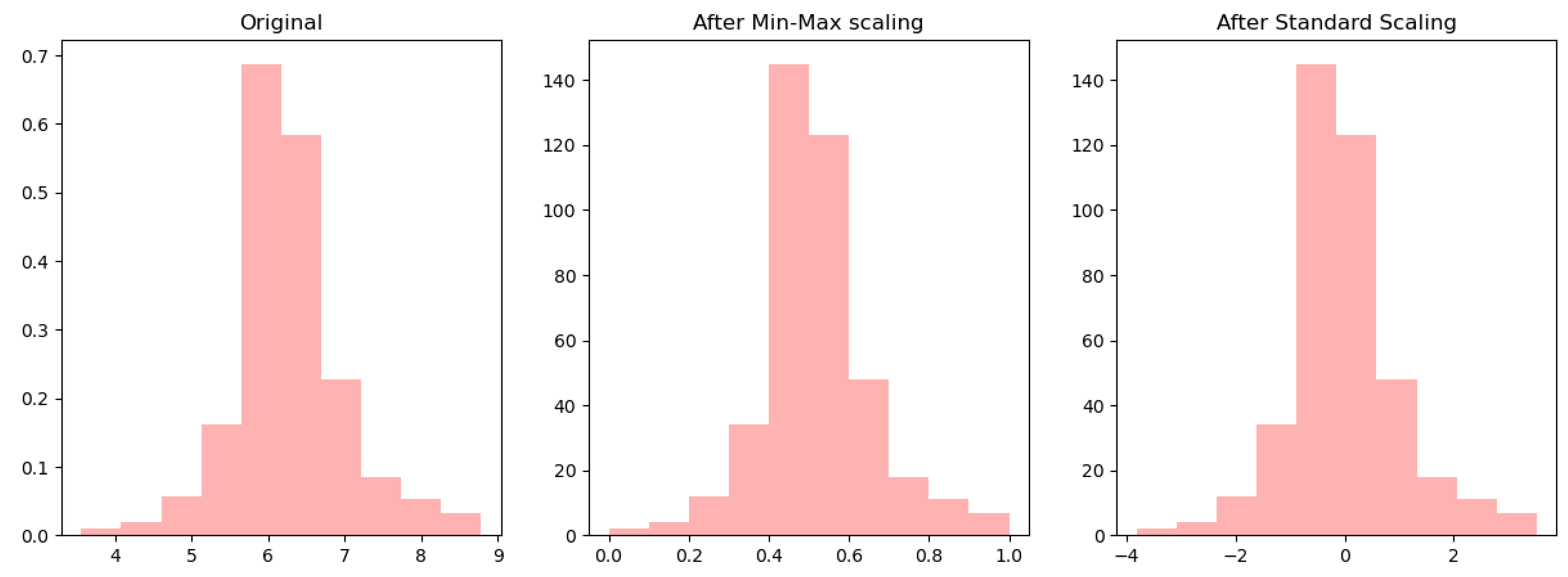
The `StandardScaler` class standardizes features by removing the mean and scaling to unit variance. The standard score of a sample `x` is `(x - u) / s`, where `u` is the mean of the training samples, and `s` is the standard deviation of the training samples. This type of scaling is often used when we want to assume that all features are centered around zero and have variance in the same order.

In the provided code, the `fit` method is used to compute the minimum and maximum to be used for later scaling. It takes as input a 2D array-like object where each row is a sample and each column is a feature. The `transform` method is then used to scale the 'RM' feature according to the previously computed minimum and maximum. The transformed data is stored in the `RM_minmax` and `RM_standard` variables for the `MinMaxScaler` and `StandardScaler` respectively.

The transformed data will have the same number of samples as the input data, but the values will be scaled according to the chosen scaling method. This can be useful for many machine learning algorithms that perform better or converge faster when features are on a relatively similar scale and/or close to normally distributed.

Now lets plot the feature distribution before and after to see the difference

```
plt.figure(figsize=(15,5))
plt.subplot(1,3,1)
plt.hist(bostonHouseTrainFrame['RM'], alpha=0.3, color='r', density=True)
plt.title("Original")

plt.subplot(1,3,2)
plt.hist(RM_minmax, alpha=0.3, color='r')
plt.title("After Min-Max scaling")

plt.subplot(1,3,3)
plt.hist(RM_standard, alpha=0.3, color='r')
plt.title("After Standard Scaling")
plt.show()
```



� What observations did you make?

> ✔ **Observations:**
>
> - Both scaling methods do not change the shape of the feature distribution. They only change the range.

lets go with min-max scaling.

> ⚠ **Warning: When normalizing, ensure that the same scaling parameters are applied to all splits (train/test/validation).**
> A common mistake is to use one set of scaling parameters to do the normalization of train data and another on test data. This happens if you apply `fit_transform()` function twice: ones to train set and again for test data.
> The correct approach would be to do the fit() on train data and then apply the transform() to train set and test set separately, to scale the data.

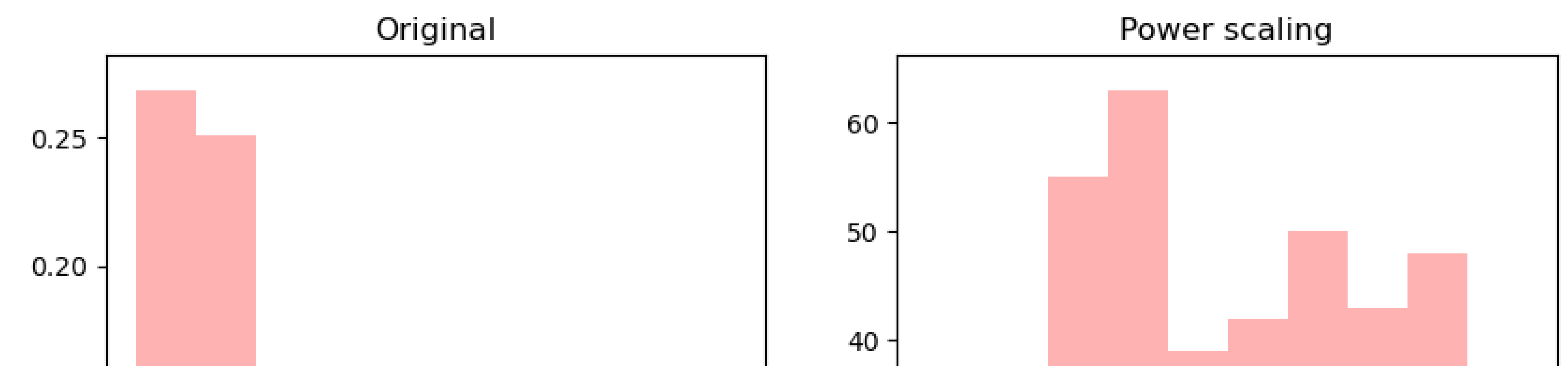
We can also use non linear transformation to map a feature that has a skewed distribution to have a distribution that is close to a gaussian.

> ☞ **Task: Read this article on skewed distribution.**

Lets try a non-linear transformation with attribute `DIS`

```
import matplotlib.pyplot as plt

# Existing code
from sklearn.preprocessing import PowerTransformer

PowerTransformer_CRIM = PowerTransformer(method='yeo-johnson', standardize=False).fit(bostonHouseTrainFrame[['DIS']])
RM_power = PowerTransformer_CRIM.transform(bostonHouseTrainFrame[['DIS']])

plt.figure(figsize=(10,5))
plt.subplot(1,2,1)
plt.hist(bostonHouseTrainFrame['DIS'], alpha=0.3, color='r', density=True)
plt.title("Original")

plt.subplot(1,2,2)
plt.hist(RM_power, alpha=0.3, color='r')
plt.title("Power scaling")
```

```
Text(0.5, 1.0, 'Power scaling')
```



> ☞ **Task: Select the appropriate feature scaling method for all numerical attributes in the boston house dataset.**

There are many other normalization techniques you can try. See [scikit-learn preprocessing documentation] for more information.

There are also methods that can handle outliers. See [Compare the effect of different scalers on data with outliers].

```python
from sklearn.preprocessing import MinMaxScaler, StandardScaler
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error

# Select the numerical attributes
numerical_attributes = ['CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE', 'DIS', 'RAD', 'TAX', 'PTRATIO', 'B', 'LSTAT']

# Apply MinMaxScaler
minmax_scaler = MinMaxScaler()
bostonHouseFrame[numerical_attributes] = minmax_scaler.fit_transform(bostonHouseFrame[numerical_attributes])

X = bostonHouseFrame[numerical_attributes]
y = bostonHouseFrame['MEDV']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Create a linear regression model
linear_reg = LinearRegression()

# Fit the model on the training data
linear_reg.fit(X_train, y_train)

# Predict on the testing data
y_pred = linear_reg.predict(X_test)

# Calculate the mean squared error
mse = mean_squared_error(y_test, y_pred)
print(mse)

# Apply StandardScaler
standard_scaler = StandardScaler()
bostonHouseFrame[numerical_attributes] = standard_scaler.fit_transform(bostonHouseFrame[numerical_attributes])

# Split the data into training and testing sets
X = bostonHouseFrame[numerical_attributes]
y = bostonHouseFrame['MEDV']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Create a linear regression model
linear_reg = LinearRegression()

# Fit the model on the training data
linear_reg.fit(X_train, y_train)

# Predict on the testing data
y_pred = linear_reg.predict(X_test)

# Calculate the mean squared error
mse = mean_squared_error(y_test, y_pred)
print(mse)

```

```
24.291119474973527
24.291119474973517
```

This Python code is using the `MinMaxScaler` and `StandardScaler` classes from the `sklearn.preprocessing` module, the `LinearRegression` class from the `sklearn.linear_model` module, and the `mean_squared_error` function from the `sklearn.metrics` module to preprocess a pandas DataFrame `bostonHouseFrame`, train a linear regression model on it, and evaluate the model's performance.

First, it selects the numerical attributes from the DataFrame. These are the features that the model will be trained on.

Then, it applies the `MinMaxScaler` to these numerical attributes. This scaler transforms each feature by scaling it to a given range, typically between zero and one. The `fit_transform` method is used to compute the minimum and maximum values to be used for scaling and then apply the scaling to the data.