Week08_Cohort

December 10, 2021

1 Week 8 Problem Set

```
In [3]: import pandas as pd
        import matplotlib.pyplot as plt
        import seaborn as sns
        import numpy as np
```

1.1 Cohort Sessions

CS1. *Reading Data:* Read CSV file for Boston Housing prices.

- **Task 1:** Read the data set. Hint:
  - Pandas read_csv
  - Boston's housing data set (filename: `housing_processed.csv`):
  - Boston's housing data description.

```
In [4]: # Task 1
        # read CSV file, replace the None
        df = pd.read_csv("housing_processed.csv")

        ###
        ### YOUR CODE HERE
        ###

        display(df)
```

```
        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]
```

In [5]:
```
assert isinstance(df, pd.DataFrame)
assert df.shape == (506, 14)
assert df.columns[0] == 'CRIM' and df.columns[-1] == 'MEDV'
```

- **Task 2:** Display the number of rows and columns. Hint:
  - you can use `df.shape` to get the number of rows and columns

In [6]:
```
# Task 2
# get the shape from the data frame, replace the None
shape = df.shape

# use the 'shape' variable to get the row and the column
# replace the None
row = shape[0]
col = shape[1]

###
### YOUR CODE HERE
###

print(shape)
print(row, col)
```

```
(506, 14)
506 14
```

In [7]:
```
assert shape == (506, 14)
assert row == 506
assert col == 14
```

- **Task 3:** Display the name of all the columns. Hint:
  - you can use `df.columns` to get the name of all the columns
  - check the meaning of each column using the link above

```
In [8]: # Task 3
        # display column names, replace the None
        names = df.columns

        ###
        ### YOUR CODE HERE
        ###

        display(names)
```

```
Index(['CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE', 'DIS', 'RAD', 'TAX',
       'PTRATIO', 'B', 'LSTAT', 'MEDV'],
      dtype='object')
```

```
In [9]: assert isinstance(names, pd.Index)
        assert np.all(names == pd.Index(['CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE', 'DI
               'PTRATIO', 'B', 'LSTAT', 'MEDV']))
```

- **Task 4:** Do the following:
  - Create a subset data set containing only the following columns: "RM", "DIS", "INDUS" for the features. Make sure it is of `pd.DataFrame` type.
  - Create a subset data set containing only "MEDV" for the target. Make sure it is of `pd.DataFrame` type.

```
In [10]: # Task 4
         # Specify the columns you want to extract into a list
         # replace the None
         names = df.columns

         columns = [i for i in names]

         # extract the respective columns from the data frame
         # replace the None
         df_feature = df[["RM", "DIS", "INDUS"]]

         df_target = df["MEDV"].to_frame()

         ###
         ### YOUR CODE HERE
         ###

         display(df_feature)
         display(df_target)
```

```
        RM     DIS  INDUS
0    6.575  4.0900   2.31
1    6.421  4.9671   7.07
2    7.185  4.9671   7.07
3    6.998  6.0622   2.18
4    7.147  6.0622   2.18
..     ...     ...    ...
501  6.593  2.4786  11.93
502  6.120  2.2875  11.93
503  6.976  2.1675  11.93
504  6.794  2.3889  11.93
505  6.030  2.5050  11.93

[506 rows x 3 columns]


     MEDV
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

[506 rows x 1 columns]
```

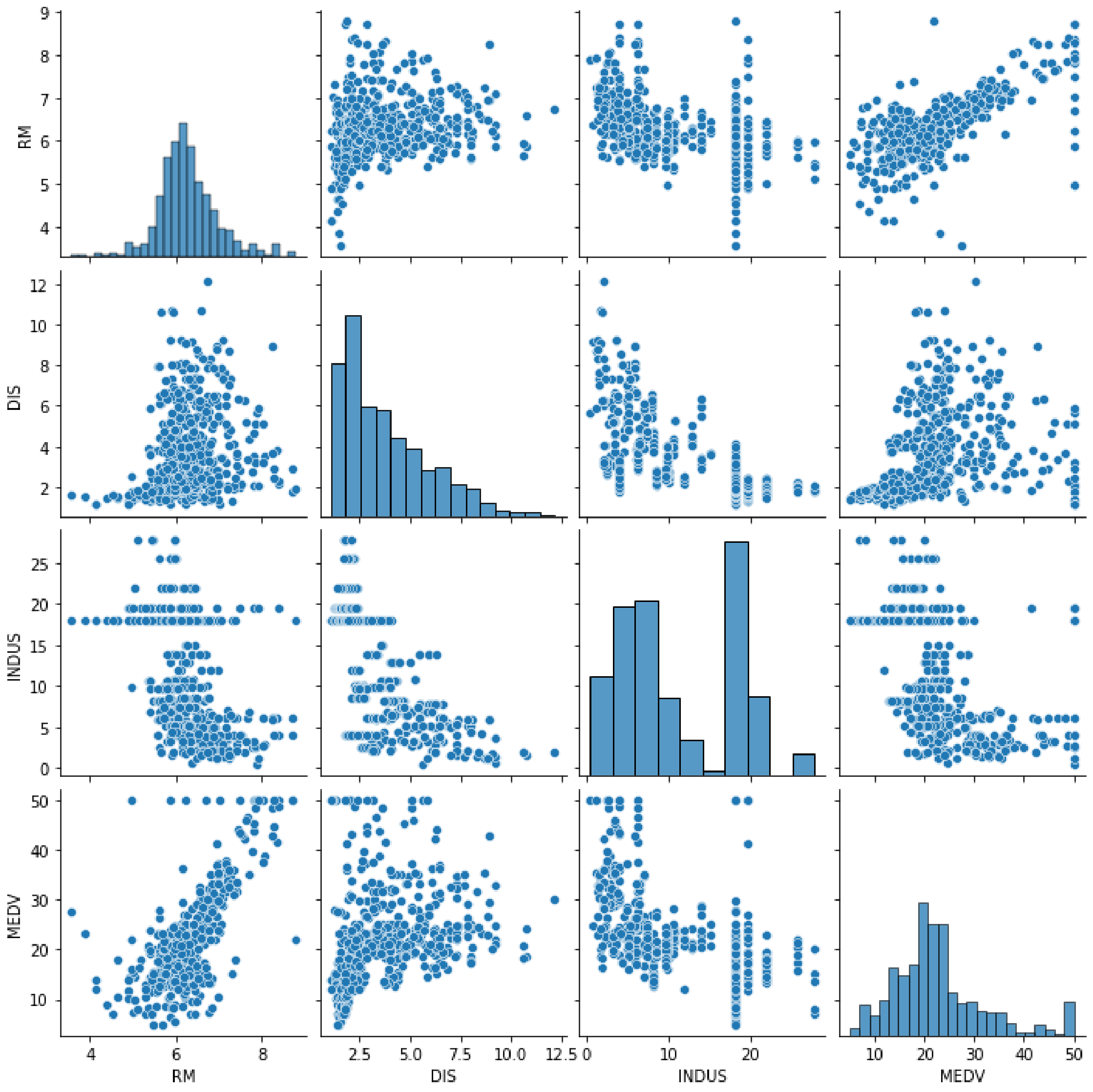
```
In [ ]:
```

```
In [11]: assert isinstance(df_feature, pd.DataFrame)
         assert isinstance(df_target, pd.DataFrame)
         assert df_feature.shape == (506, 3)
         assert df_target.shape == (506, 1)
         assert np.all(df_feature.columns == pd.Index(['RM', 'DIS', 'INDUS']))
         assert df_target.columns == pd.Index(['MEDV'])
```

CS2. *Data Frame Operation:*
Reference: - Indexing and Selecting Data
Create separate and new data frame for the columns: "RM", "DIS", "INDUS", "MEDV" that satisfies each of the following condition:

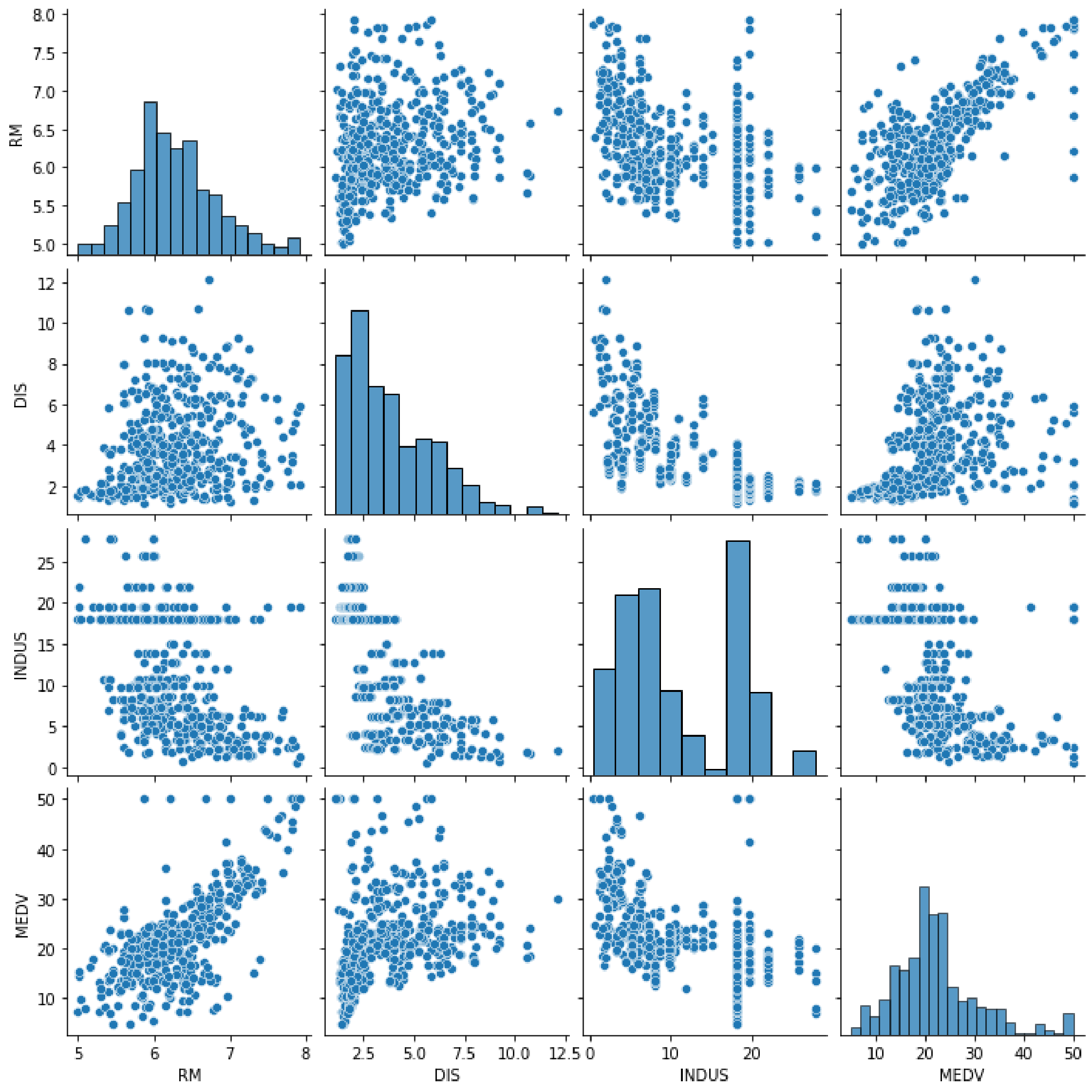
- **Task 1:** All records with weighted distances to ve Boston employment centers between 0 to 3.

```
In [12]: # specify the columns of interest, replace the None
         columns = ["RM", "DIS", "INDUS", "MEDV"]

         # use conditions for row selector and column selector
         # replace the None
         df_1 = df[columns]

         ###
         ### YOUR CODE HERE
         ###

         myplot = sns.pairplot(data=df_1)
```



- **Task 2:** All records with average number of room between 5 to 8.

```
In [13]: # specify the columns of interest, replace the None
         columns = ["RM", "DIS", "INDUS", "MEDV"]

         # use conditions for row selector and column selector
         # replace the None
         df_2 = df.loc[(df['RM'] <= 8) & (df['RM'] >= 5), columns]

         ###
         ### YOUR CODE HERE
         ###

         myplot = sns.pairplot(data=df_2)
```



```
In [14]: df.loc[:15,:]
```

```
Out[14]:        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
         5   0.02985   0.0   2.18     0  0.458  6.430   58.7  6.0622    3  222.0
         6   0.08829  12.5   7.87     0  0.524  6.012   66.6  5.5605    5  311.0
         7   0.14455  12.5   7.87     0  0.524  6.172   96.1  5.9505    5  311.0
         8   0.21124  12.5   7.87     0  0.524  5.631  100.0  6.0821    5  311.0
         9   0.17004  12.5   7.87     0  0.524  6.004   85.9  6.5921    5  311.0
         10  0.22489  12.5   7.87     0  0.524  6.377   94.3  6.3467    5  311.0
         11  0.11747  12.5   7.87     0  0.524  6.009   82.9  6.2267    5  311.0
         12  0.09378  12.5   7.87     0  0.524  5.889   39.0  5.4509    5  311.0
         13  0.62976   0.0   8.14     0  0.538  5.949   61.8  4.7075    4  307.0
         14  0.63796   0.0   8.14     0  0.538  6.096   84.5  4.4619    4  307.0
         15  0.62739   0.0   8.14     0  0.538  5.834   56.5  4.4986    4  307.0

             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
         5      18.7  394.12   5.21  28.7
         6      15.2  395.60  12.43  22.9
         7      15.2  396.90  19.15  27.1
         8      15.2  386.63  29.93  16.5
         9      15.2  386.71  17.10  18.9
         10     15.2  392.52  20.45  15.0
         11     15.2  396.90  13.27  18.9
         12     15.2  390.50  15.71  21.7
         13     21.0  396.90   8.26  20.4
         14     21.0  380.02  10.26  18.2
         15     21.0  395.62   8.47  19.9
```

- **Task 3:** The first 15 records in the table.

```
In [15]: # specify the columns of interest, replace the None
         columns = ["RM", "DIS", "INDUS", "MEDV"]

         # use conditions for row selector and column selector
         # replace the None
         df_3 = df.iloc[:15,:]

         ###
         ### YOUR CODE HERE
         ###
```

```
display(df_3)
```

```
      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
5  0.02985   0.0   2.18     0  0.458  6.430   58.7  6.0622    3  222.0
6  0.08829  12.5   7.87     0  0.524  6.012   66.6  5.5605    5  311.0
7  0.14455  12.5   7.87     0  0.524  6.172   96.1  5.9505    5  311.0
8  0.21124  12.5   7.87     0  0.524  5.631  100.0  6.0821    5  311.0
9  0.17004  12.5   7.87     0  0.524  6.004   85.9  6.5921    5  311.0
10 0.22489  12.5   7.87     0  0.524  6.377   94.3  6.3467    5  311.0
11 0.11747  12.5   7.87     0  0.524  6.009   82.9  6.2267    5  311.0
12 0.09378  12.5   7.87     0  0.524  5.889   39.0  5.4509    5  311.0
13 0.62976   0.0   8.14     0  0.538  5.949   61.8  4.7075    4  307.0
14 0.63796   0.0   8.14     0  0.538  6.096   84.5  4.4619    4  307.0

   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
5     18.7  394.12   5.21  28.7
6     15.2  395.60  12.43  22.9
7     15.2  396.90  19.15  27.1
8     15.2  386.63  29.93  16.5
9     15.2  386.71  17.10  18.9
10    15.2  392.52  20.45  15.0
11    15.2  396.90  13.27  18.9
12    15.2  390.50  15.71  21.7
13    21.0  396.90   8.26  20.4
14    21.0  380.02  10.26  18.2
```

- **Task 4:** The last 15 records in the table.

In [16]:
```
# specify the columns of interest, replace the None
columns = ["RM", "DIS", "INDUS", "MEDV"]

# use conditions for row selector and column selector
# replace the None

df_4 = df.iloc[-15:,:]
```

```
###
### YOUR CODE HERE
###

display(df_4)
```

```
        CRIM   ZN  INDUS  CHAS    NOX     RM   AGE     DIS  RAD    TAX  \
491  0.10574  0.0  27.74     0  0.609  5.983  98.8  1.8681    4  711.0
492  0.11132  0.0  27.74     0  0.609  5.983  83.5  2.1099    4  711.0
493  0.17331  0.0   9.69     0  0.585  5.707  54.0  2.3817    6  391.0
494  0.27957  0.0   9.69     0  0.585  5.926  42.6  2.3817    6  391.0
495  0.17899  0.0   9.69     0  0.585  5.670  28.8  2.7986    6  391.0
496  0.28960  0.0   9.69     0  0.585  5.390  72.9  2.7986    6  391.0
497  0.26838  0.0   9.69     0  0.585  5.794  70.6  2.8927    6  391.0
498  0.23912  0.0   9.69     0  0.585  6.019  65.3  2.4091    6  391.0
499  0.17783  0.0   9.69     0  0.585  5.569  73.5  2.3999    6  391.0
500  0.22438  0.0   9.69     0  0.585  6.027  79.7  2.4982    6  391.0
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
491     20.1  390.11  18.07  13.6
492     20.1  396.90  13.35  20.1
493     19.2  396.90  12.01  21.8
494     19.2  396.90  13.59  24.5
495     19.2  393.29  17.60  23.1
496     19.2  396.90  21.14  19.7
497     19.2  396.90  14.10  18.3
498     19.2  396.90  12.92  21.2
499     19.2  395.77  15.10  17.5
500     19.2  396.90  14.33  16.8
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
```

- **Task 5:** All records with even index numbers, i.e. index 0, 2, 4, ...

```
In [17]: # specify the columns of interest, replace the None
         columns = ["RM", "DIS", "INDUS", "MEDV"]

         # use conditions for row selector and column selector
         # replace the None
```

```
df_5 = df.loc[df.index%2==0,:]
# or df.loc[::2,columns]

###
### YOUR CODE HERE
###

display(df_5)
```

```
        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
2    0.02729   0.0   7.07     0  0.469  7.185   61.1  4.9671    2  242.0
4    0.06905   0.0   2.18     0  0.458  7.147   54.2  6.0622    3  222.0
6    0.08829  12.5   7.87     0  0.524  6.012   66.6  5.5605    5  311.0
8    0.21124  12.5   7.87     0  0.524  5.631  100.0  6.0821    5  311.0
..       ...   ...    ...   ...    ...    ...    ...     ...  ...    ...
496  0.28960   0.0   9.69     0  0.585  5.390   72.9  2.7986    6  391.0
498  0.23912   0.0   9.69     0  0.585  6.019   65.3  2.4091    6  391.0
500  0.22438   0.0   9.69     0  0.585  6.027   79.7  2.4982    6  391.0
502  0.04527   0.0  11.93     0  0.573  6.120   76.7  2.2875    1  273.0
504  0.10959   0.0  11.93     0  0.573  6.794   89.3  2.3889    1  273.0

     PTRATIO       B  LSTAT  MEDV
0       15.3  396.90   4.98  24.0
2       17.8  392.83   4.03  34.7
4       18.7  396.90   5.33  36.2
6       15.2  395.60  12.43  22.9
8       15.2  386.63  29.93  16.5
..       ...     ...    ...   ...
496     19.2  396.90  21.14  19.7
498     19.2  396.90  12.92  21.2
500     19.2  396.90  14.33  16.8
502     21.0  396.90   9.08  20.6
504     21.0  393.45   6.48  22.0

[253 rows x 14 columns]
```

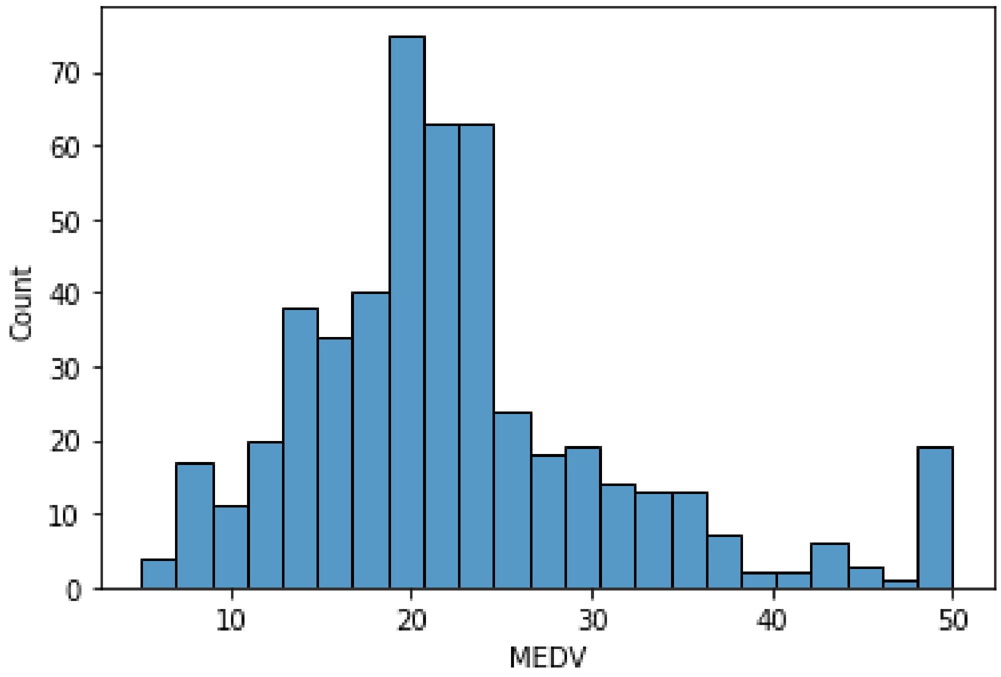
CS3. *Histogram and Box plot:* Plot the histogram for the median value in $1000 for the Boston's housing price.

Reference: - Histogram - Box plot

- **Task 1:** Plot the histogram with default bin values.

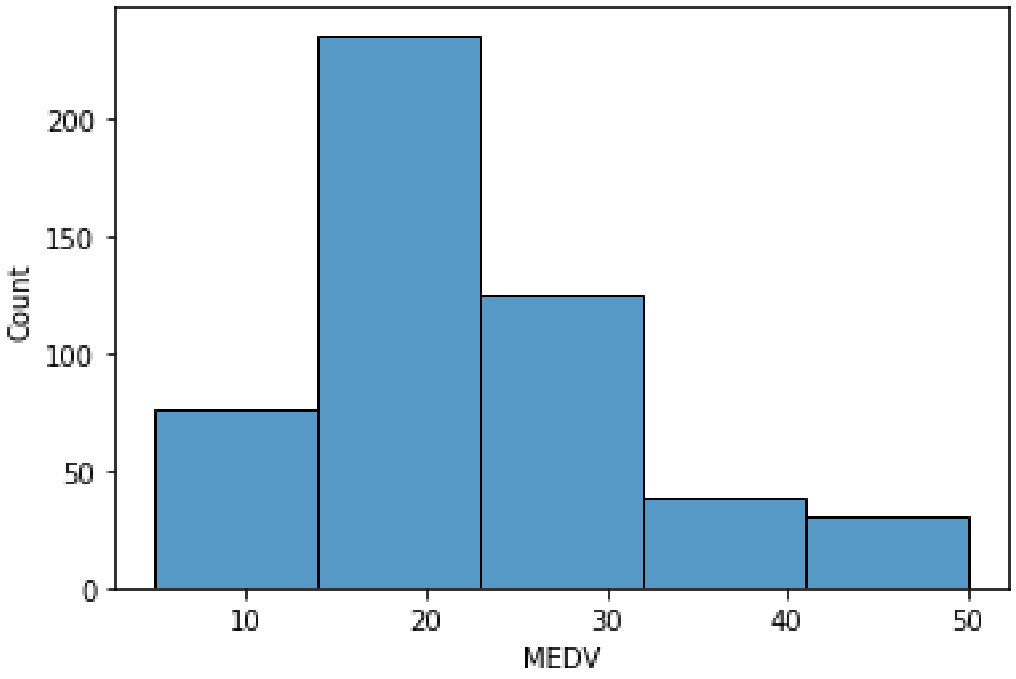
```
In [18]: # plot histogram for MEDV, replace the None
         myplot = sns.histplot(data = df['MEDV'])

         # set the x label, write the code below
```

```
        # set the y label, write the code below

        ###
        ### YOUR CODE HERE
        ###
```



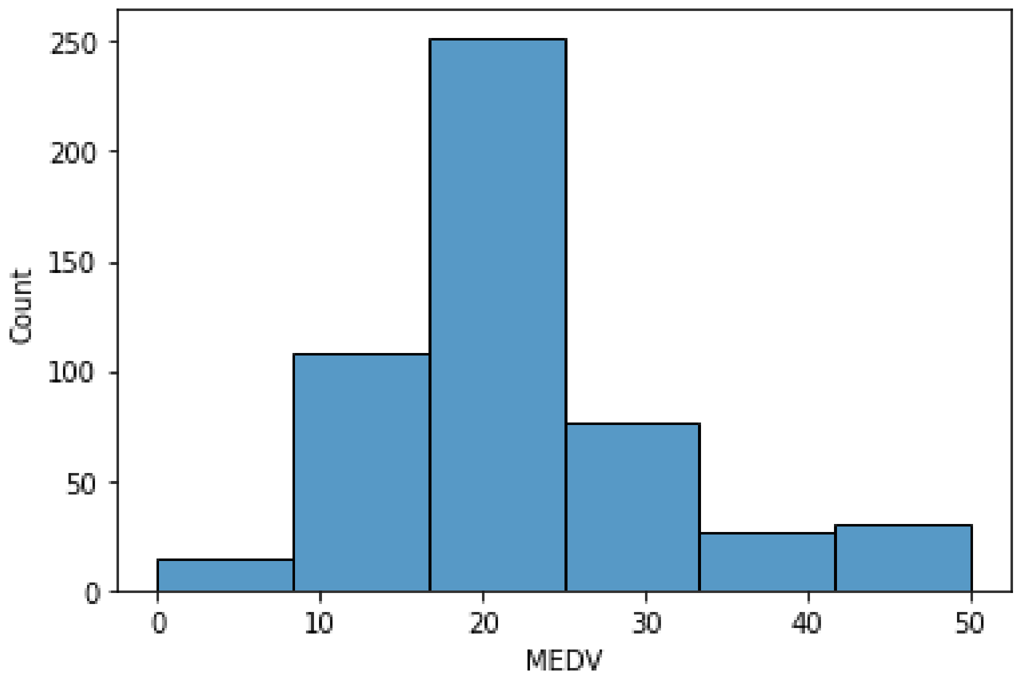
- **Task 2:** Plot the histogram with 5 bins only. Hint:

```
In [19]: # plot histogram for MEDV, replace the None
         myplot = sns.histplot(data = df['MEDV'],bins=5)


         # set the x label, write the code below

         # set the y label, write the code below

         ###
         ### YOUR CODE HERE
         ###
```



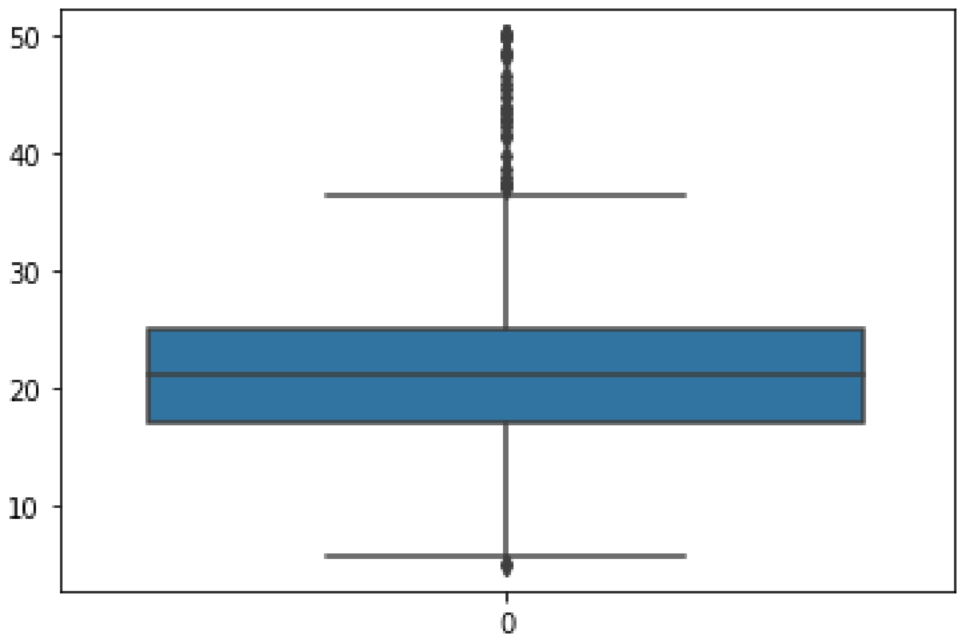
- **Task 3:** Plot the histogram with the following bin edges `0, 10, 20, 30, 40, 50`.

```
In [20]: # plot histogram for MEDV, replace the None
         myplot = sns.histplot(data = df['MEDV'],binrange = (0,50),bins = 6)

         # set the x label, write the code below

         # set the y label, write the code below

         ###
         ### YOUR CODE HERE
         ###
```



- **Task 4:** Plot the same data using a box plot in a horizontal manner.

```
In [21]: # plot boxplot for MEDV, replace the None
         myplot = sns.boxplot(data = df['MEDV'])


         # set the x label, write the code below

         ###
         ### YOUR CODE HERE
         ###
```

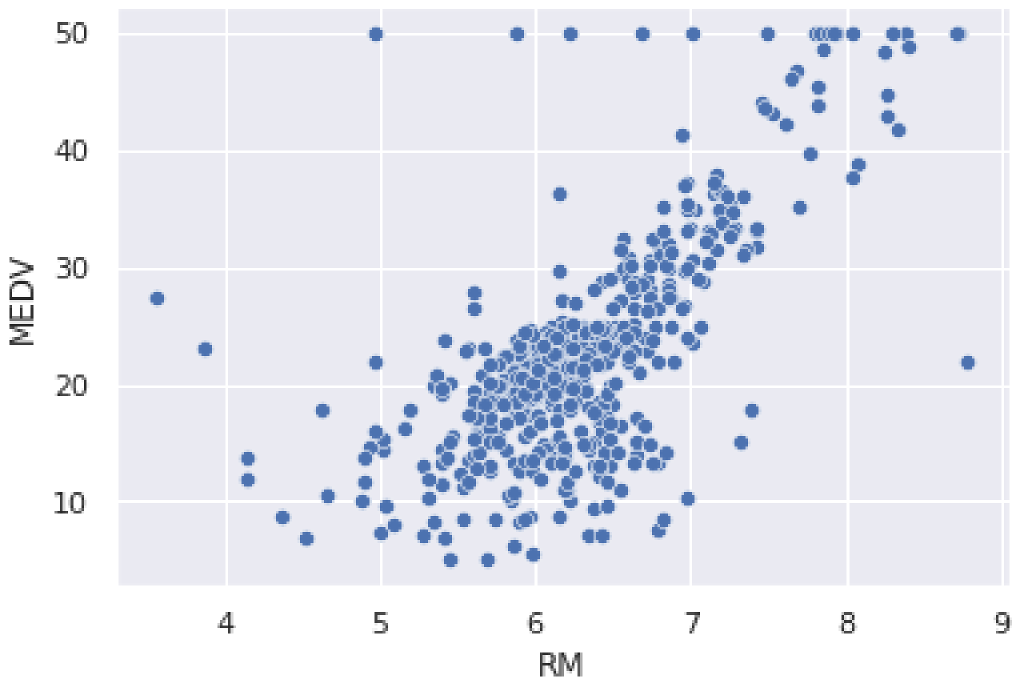


CS4. *Scatter plot:* Do the following plots.

- **Task 1:** Display scatter plot of "RM" versus "MEDV". Hint:
  - Use Seaborn default theme instead of matplotlib
  - Seaborn scatter plot
  - Use "RM" as x data
  - Use "MEDV" as y data

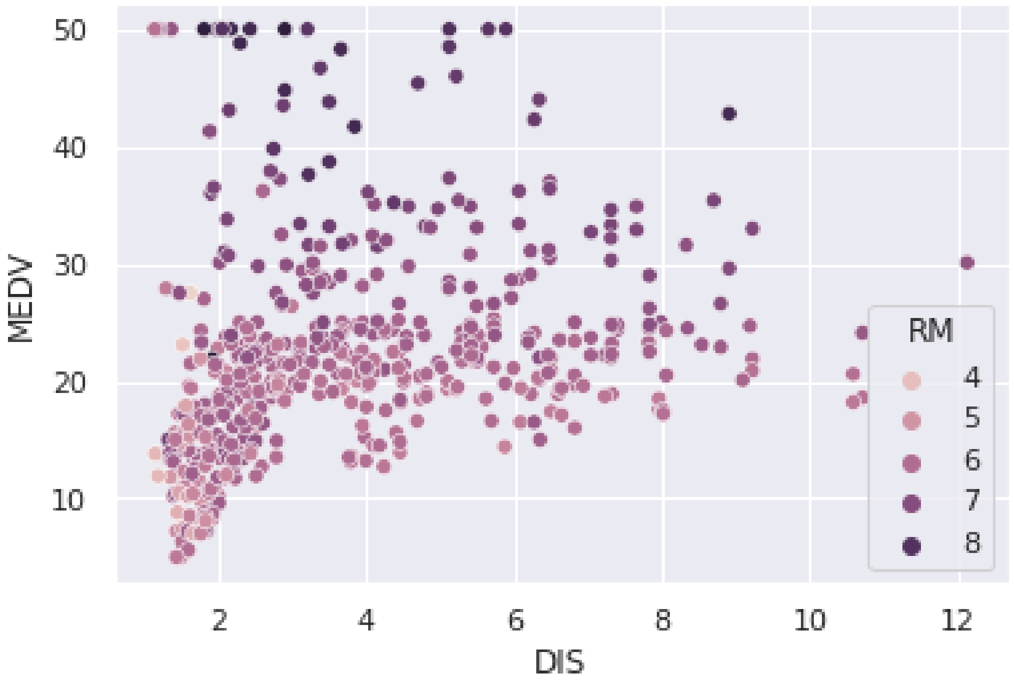
```
In [22]: # Set the theme parameter to Seaborn's default
         sns.set()
```

```
In [23]: # Task 5
         # plot scatter plot, replace the None
         myplot = sns.scatterplot(x='RM', y='MEDV', data=df)

         ###
         ### YOUR CODE HERE
         ###
```



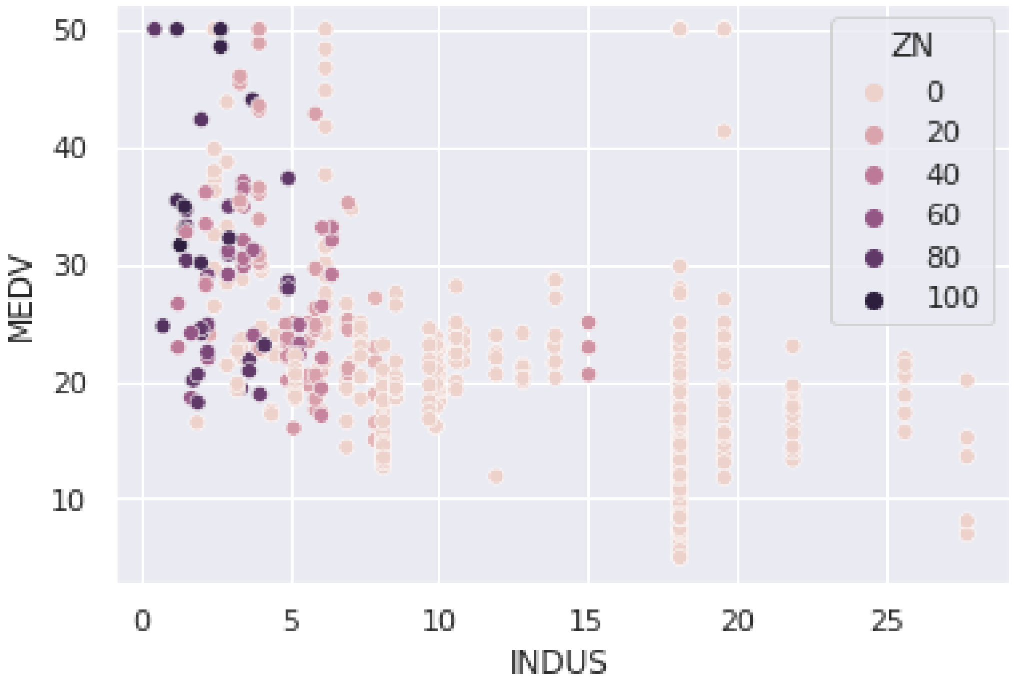
- **Task 2:** Display a scatter plot "weighted distances to ve Boston employment centers" versus "Median value of owner-occupied homes in $1000s". Use average number of rooms per dwelling as the hue data.

```
In [24]: # Task 6
         # plot using scatter plot
         # replace the None
         myplot = sns.scatterplot(x='DIS', y='MEDV', hue = 'RM', data=df)

         ###
         ### YOUR CODE HERE
         ###
```



- **Task 3:** Display a scatter plot "proportion of non-retail business acres per town" versus "Median value of owner-occupied homes in $1000s". Use "proportion of residential land zoned for lots over 25,000 sq.ft." as the hue data.

```
In [25]: # Task 3
         # plot using scatter plot and use "hue"
         # replace the None
         myplot = sns.scatterplot(x='INDUS', y='MEDV', hue="ZN", data=df)

         ###
         ### YOUR CODE HERE
         ###
```



Answer the following: - How many columns are there? - How many rows are there? - What are the names of the columns - What is the range of the median value in $1000 when the average number of room per dwelling is 8? - What's the relationship between distance to five Boston's employment centres and the median value in $1000? - What's the relationship between the proportion of non-retail business acres per town adn the median value in $1000?

CS5. *Standardization:* Write a function that takes in data frame where all the column are the features and normalize each column according to the following formula.

$$normalized = \frac{data - \mu}{\sigma}$$

where μ is the mean of the data and σ is the standard deviation of the data. **You need to normalize for each column respectively**. The function should return a new data frame.

Use the following functions from Pandas: - `df.mean(axis=0)`: This is to calculate the mean along the index axis. - `df.std(axis=0)`: This is to calculate the standard deviation along the index axis.

Note: Your function should be able to handle a numpy array as well as Panda's data frame. *Hint: use axis=0 argument when calculating mean and standard deviation.*

```
In [28]: def normalize_z(dfin):

             dfout = (dfin - dfin.mean(axis=0))/dfin.std(axis=0)

             return dfout
```

```
In [29]: data_norm = normalize_z(df_feature)
         stats = data_norm.describe()
```

```
        display(stats)
        assert np.isclose(stats.loc["mean", "RM"], 0.0) and \
               np.isclose(stats.loc["std", "RM"], 1.0, atol=1e-3)
        assert np.isclose(stats.loc["mean", "DIS"], 0.0) and \
               np.isclose(stats.loc["std", "DIS"], 1.0, atol=1e-3)
        assert np.isclose(stats.loc["mean", "INDUS"], 0.0) and \
               np.isclose(stats.loc["std", "INDUS"], 1.0, atol=1e-3)
```

```
                 RM           DIS         INDUS
count  5.060000e+02  5.060000e+02  5.060000e+02
mean  -1.148313e-14  7.161597e-16 -3.145486e-15
std    1.000000e+00  1.000000e+00  1.000000e+00
min   -3.876413e+00 -1.265817e+00 -1.556302e+00
25%   -5.680681e-01 -8.048913e-01 -8.668328e-01
50%   -1.083583e-01 -2.790473e-01 -2.108898e-01
75%    4.822906e-01  6.617161e-01  1.014995e+00
max    3.551530e+00  3.956602e+00  2.420170e+00
```

CS6. *Splitting Data Randomly:* Create a function to split the Data Frame randomly. The function should have the following arguments: - `df_feature`: which is the data frame for the features. - `df_target`: which is the data frame for the target. - `random_state`: which is the seed used to split randomly. - `test_size`: which is the fraction for the test data set (0 to 1), by default is set to 0.5

The output of the function is a tuple of four items: - `df_feature_train`: which is the train set for the features data frame - `df_feature_test`: which is the test set for the features data frame - `df_target_train`: which is the train set for the target data frame - `df_target_test`: which is the test set for the target data frame

Hint: - Use `numpy.random.choice()`

```
In [ ]:
```

```
In [ ]: def split_data(df_feature, df_target, random_state=None, test_size=0.5):
            np.random.seed(random_state)
            N = df_feature.shape[0]
            print(N)
            sample = int(test_size*N)

            train_idx = np.random.choice(N, N-sample,replace=False)

            df_feature_train = df_feature.iloc[train_idx]
            df_target_train = df_target.iloc[train_idx]

            test_idx = [idx for idx in range(N) if idx not in train_idx]

            df_feature_test = df_feature.iloc[test_idx]
            df_target_test = df_target.iloc[test_idx]

            return df_feature_train, df_feature_test, df_target_train, df_target_test
```

```
In [ ]: df_feature_train, df_feature_test, df_target_train, df_target_test = split_data(df_fea

        display(df_feature_train.describe())
        display(df_feature_test.describe())
        display(df_target_train.describe())
        display(df_target_test.describe())

        assert np.all(df_feature_train.count() == 355)
        assert np.all(df_feature_test.count() == 151)
        assert np.all(df_target_train.count() == 355)
        assert np.all(df_target_test.count() == 151)

        assert np.isclose(df_feature_train["RM"].mean(), 6.2968)
        assert np.isclose(df_feature_test["DIS"].std(), 2.2369)
        assert np.isclose(df_target_train["MEDV"].median(), 21.40)
        assert np.isclose(df_target_test["MEDV"].median(), 20.90)

In [ ]: df_feature_train.count()

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:
```

Week08_Homework

December 10, 2021

1 Week 8 Problem Set

1.1 Homeworks

```
In [4]: import pandas as pd
        import matplotlib.pyplot as plt
        import seaborn as sns
        import numpy as np
```

HW1. *Breast Cancer Data:* Read Breast Cancer data using Pandas library: - **Task 1:** Read the file: `breast_cancer_data.csv`: - Download the CSV file - Read Description of Data - use comma `","` as field separator - use `"utf-8"` as encoding field

```
In [5]: # Task 1
        # Read file, replace the None
        df = pd.read_csv("breast_cancer_data.csv", sep=",", encoding="utf-8")

        ###
        ### YOUR CODE HERE
        ###
        display(df)
```

```
           id diagnosis  radius_mean  texture_mean  perimeter_mean  area_mean  \
0      842302         M        17.99         10.38          122.80     1001.0
1      842517         M        20.57         17.77          132.90     1326.0
2    84300903         M        19.69         21.25          130.00     1203.0
3    84348301         M        11.42         20.38           77.58      386.1
4    84358402         M        20.29         14.34          135.10     1297.0
..        ...       ...          ...           ...             ...        ...
564    926424         M        21.56         22.39          142.00     1479.0
565    926682         M        20.13         28.25          131.20     1261.0
566    926954         M        16.60         28.08          108.30      858.1
567    927241         M        20.60         29.33          140.10     1265.0
568     92751         B         7.76         24.54           47.92      181.0

     smoothness_mean  compactness_mean  concavity_mean  concave points_mean  \
0            0.11840           0.27760         0.30010              0.14710
1            0.08474           0.07864         0.08690              0.07017
```

```
2           0.10960           0.15990         0.19740              0.12790
3           0.14250           0.28390         0.24140              0.10520
4           0.10030           0.13280         0.19800              0.10430
..              ...               ...             ...                  ...
564         0.11100           0.11590         0.24390              0.13890
565         0.09780           0.10340         0.14400              0.09791
566         0.08455           0.10230         0.09251              0.05302
567         0.11780           0.27700         0.35140              0.15200
568         0.05263           0.04362         0.00000              0.00000

     ...  radius_worst  texture_worst  perimeter_worst  area_worst  \
0    ...        25.380          17.33           184.60      2019.0
1    ...        24.990          23.41           158.80      1956.0
2    ...        23.570          25.53           152.50      1709.0
3    ...        14.910          26.50            98.87       567.7
4    ...        22.540          16.67           152.20      1575.0
..   ...           ...            ...              ...         ...
564  ...        25.450          26.40           166.10      2027.0
565  ...        23.690          38.25           155.00      1731.0
566  ...        18.980          34.12           126.70      1124.0
567  ...        25.740          39.42           184.60      1821.0
568  ...         9.456          30.37            59.16       268.6

     smoothness_worst  compactness_worst  concavity_worst  \
0             0.16220            0.66560           0.7119
1             0.12380            0.18660           0.2416
2             0.14440            0.42450           0.4504
3             0.20980            0.86630           0.6869
4             0.13740            0.20500           0.4000
..                ...                ...              ...
564           0.14100            0.21130           0.4107
565           0.11660            0.19220           0.3215
566           0.11390            0.30940           0.3403
567           0.16500            0.86810           0.9387
568           0.08996            0.06444           0.0000

     concave points_worst  symmetry_worst  fractal_dimension_worst
0                  0.2654          0.4601                  0.11890
1                  0.1860          0.2750                  0.08902
2                  0.2430          0.3613                  0.08758
3                  0.2575          0.6638                  0.17300
4                  0.1625          0.2364                  0.07678
..                    ...             ...                      ...
564                0.2216          0.2060                  0.07115
565                0.1628          0.2572                  0.06637
566                0.1418          0.2218                  0.07820
567                0.2650          0.4087                  0.12400
568                0.0000          0.2871                  0.07039
```

```
[569 rows x 32 columns]
```

```
In [6]: assert isinstance(df, pd.DataFrame)
        assert df.shape == (569, 32)
        assert df.columns[0] == 'id' and df.columns[-1] == 'fractal_dimension_worst'
```

- **Task 2:** Find the number of rows and columns.

```
In [7]: # Task 2
        # get the shape
        shape = df.shape

        # get rows and columns from shape
        row = shape[0]
        col = shape[1]

        ###
        ### YOUR CODE HERE
        ###
```

```
In [8]: assert shape == (569, 32)
        assert row == 569
        assert col == 32
```

- **Task 3:** Find the name of all the columns.

```
In [9]: # Task 3
        # display the name of all the columns
        names = df.columns

        ###
        ### YOUR CODE HERE
        ###

        print(names)
```

```
Index(['id', 'diagnosis', 'radius_mean', 'texture_mean', 'perimeter_mean',
       'area_mean', 'smoothness_mean', 'compactness_mean', 'concavity_mean',
       'concave points_mean', 'symmetry_mean', 'fractal_dimension_mean',
       'radius_se', 'texture_se', 'perimeter_se', 'area_se', 'smoothness_se',
       'compactness_se', 'concavity_se', 'concave points_se', 'symmetry_se',
       'fractal_dimension_se', 'radius_worst', 'texture_worst',
       'perimeter_worst', 'area_worst', 'smoothness_worst',
       'compactness_worst', 'concavity_worst', 'concave points_worst',
       'symmetry_worst', 'fractal_dimension_worst'],
      dtype='object')
```

```
In [10]: assert isinstance(names, pd.Index)
         assert np.all(names == pd.Index(['id', 'diagnosis', 'radius_mean', 'texture_mean', 'pe
                'area_mean', 'smoothness_mean', 'compactness_mean', 'concavity_mean',
                'concave points_mean', 'symmetry_mean', 'fractal_dimension_mean',
                'radius_se', 'texture_se', 'perimeter_se', 'area_se', 'smoothness_se',
                'compactness_se', 'concavity_se', 'concave points_se', 'symmetry_se',
                'fractal_dimension_se', 'radius_worst', 'texture_worst',
                'perimeter_worst', 'area_worst', 'smoothness_worst',
                'compactness_worst', 'concavity_worst', 'concave points_worst',
                'symmetry_worst', 'fractal_dimension_worst']))
```

- **Task 4:** Create a subset data set containing only the following features:
  - radius (mean of distances from center to points on the perimeter)
  - texture (standard deviation of gray-scale values)
  - perimeter
  - area
  - smoothness (local variation in radius lengths)
  - concavity (severity of concave portions of the contour)

Make sure the data type is `pd.DataFrame`.

```
In [11]: # Task 4
         # set the name of the columnbs for the subset of data
         columns = ['radius_mean','texture_mean','perimeter_mean','area_mean','smoothness_mean

         # extract the columns, replace the None
         df_features = df.loc[:,columns]

         ###
         ### YOUR CODE HERE
         ###

         df_features

Out[11]:      radius_mean  texture_mean  perimeter_mean  area_mean  smoothness_mean  \
         0          17.99         10.38          122.80     1001.0          0.11840
         1          20.57         17.77          132.90     1326.0          0.08474
         2          19.69         21.25          130.00     1203.0          0.10960
         3          11.42         20.38           77.58      386.1          0.14250
         4          20.29         14.34          135.10     1297.0          0.10030
         ..           ...           ...             ...        ...              ...
         564        21.56         22.39          142.00     1479.0          0.11100
         565        20.13         28.25          131.20     1261.0          0.09780
         566        16.60         28.08          108.30      858.1          0.08455
         567        20.60         29.33          140.10     1265.0          0.11780
         568         7.76         24.54           47.92      181.0          0.05263

              concavity_mean
```

```
0           0.30010
1           0.08690
2           0.19740
3           0.24140
4           0.19800
..              ...
564         0.24390
565         0.14400
566         0.09251
567         0.35140
568         0.00000

[569 rows x 6 columns]
```

```
In [12]: assert isinstance(df_features, pd.DataFrame)
         assert df_features.columns[0] == 'radius_mean'
         assert df_features.columns[-1] == 'concavity_mean'
         assert df_features.shape == (569, 6)
```

- **Task 5:** Create a subset data set containing only the target from the column "diagnosis".

Make sure the data type is `pd.DataFrame`.

```
In [13]: #5.
         # extract target
         df_target = df[["diagnosis"]]

         ###
         ### YOUR CODE HERE
         ###

         df_target
```

```
Out[13]:     diagnosis
         0           M
         1           M
         2           M
         3           M
         4           M
         ..        ...
         564         M
         565         M
         566         M
         567         M
         568         B

         [569 rows x 1 columns]
```

```
In [14]: assert isinstance(df_target, pd.DataFrame)
         assert df_target.shape == (569, 1)
         assert df_target.columns[0] == 'diagnosis'
```

- **Task 6:** Create a new Data Frame from the column "diagnosis", called "diagnosis_int" which is the integer representation of the column diagnosis. Copy the column into the original data frame so that it has a copy.
  - if the value in "diagnosis" column is "M", the value in the column "diagnosis_int" should be set to 1
  - otherwise, it should be set to 0

Hint: use `.apply()` *method.*

```
In [15]: # Task 6
         # creating diagnosis_int
         df_target["diagnosis_int"] = df_target["diagnosis"].apply(lambda x : 1 if x == "M" el
         # or we can use this instead
         # df.loc[:,"diagnosis_int"] = 1


         # copy the new column into the original data frame
         df["diagnosis_int"] = df_target["diagnosis_int"]

         ###
         ### YOUR CODE HERE
         ###

         display(df_target)
         display(df["diagnosis_int"])
```

```
/usr/lib/python3.7/site-packages/ipykernel_launcher.py:3: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/
  This is separate from the ipykernel package so we can avoid doing imports until
```

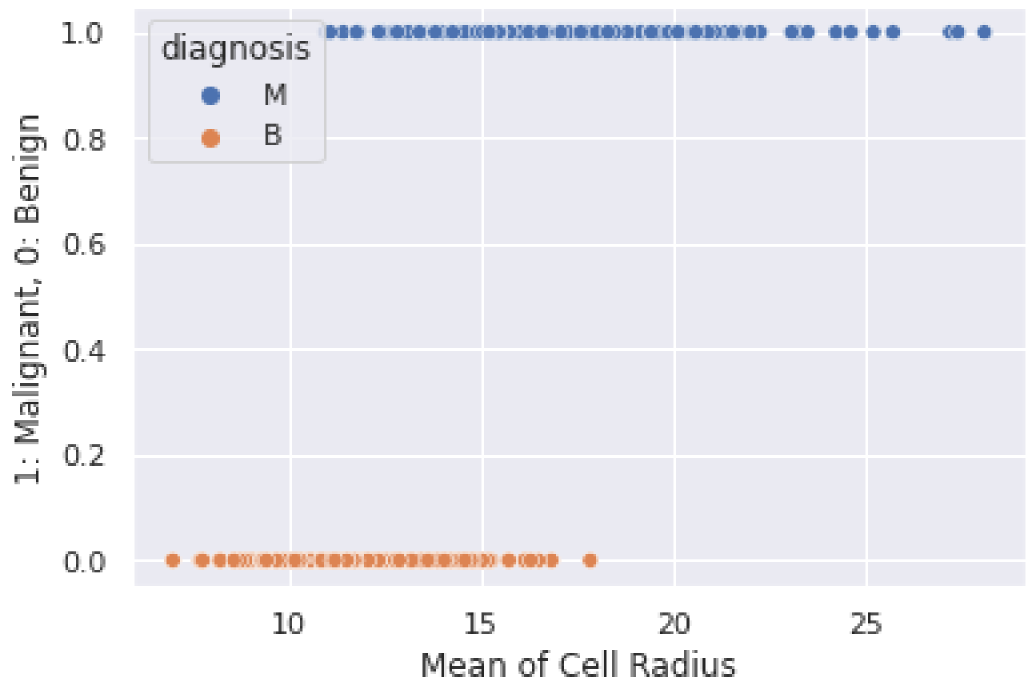
```
    diagnosis  diagnosis_int
0           M              1
1           M              1
2           M              1
3           M              1
4           M              1
..        ...            ...
564         M              1
565         M              1
566         M              1
567         M              1
568         B              0

[569 rows x 2 columns]
```

```
0      1
1      1
2      1
3      1
4      1
      ..
564    1
565    1
566    1
567    1
568    0
Name: diagnosis_int, Length: 569, dtype: int64
```

```
In [16]: assert isinstance(df_target, pd.DataFrame) and isinstance(df, pd.DataFrame)
         assert df_target.shape == (569, 2)
         assert np.all(df_target.columns == ["diagnosis", "diagnosis_int"])
         assert "diagnosis_int" in df.columns
```

- **Task 7:** Use scatter plot to see the relationship between "radius_mean" and "diagnosis_int".
  - Use the "diagnosis" column as the `hue` for the scatter plot.
  - Label the x axis as "Mean of Cell Radius"
  - Label the y axis as "1: Malignant, 0: Benign"

```
In [17]: # Task 7
         # set the default theme to use Seaborn
         sns.set()

         # display using scatter plot
         myplot = sns.scatterplot(data = df, x = "radius_mean",y= "diagnosis_int",hue = "diagno

         myplot.set(xlabel="Mean of Cell Radius", ylabel = "1: Malignant, 0: Benign")

Out[17]: [Text(0.5, 0, 'Mean of Cell Radius'), Text(0, 0.5, '1: Malignant, 0: Benign')]
```



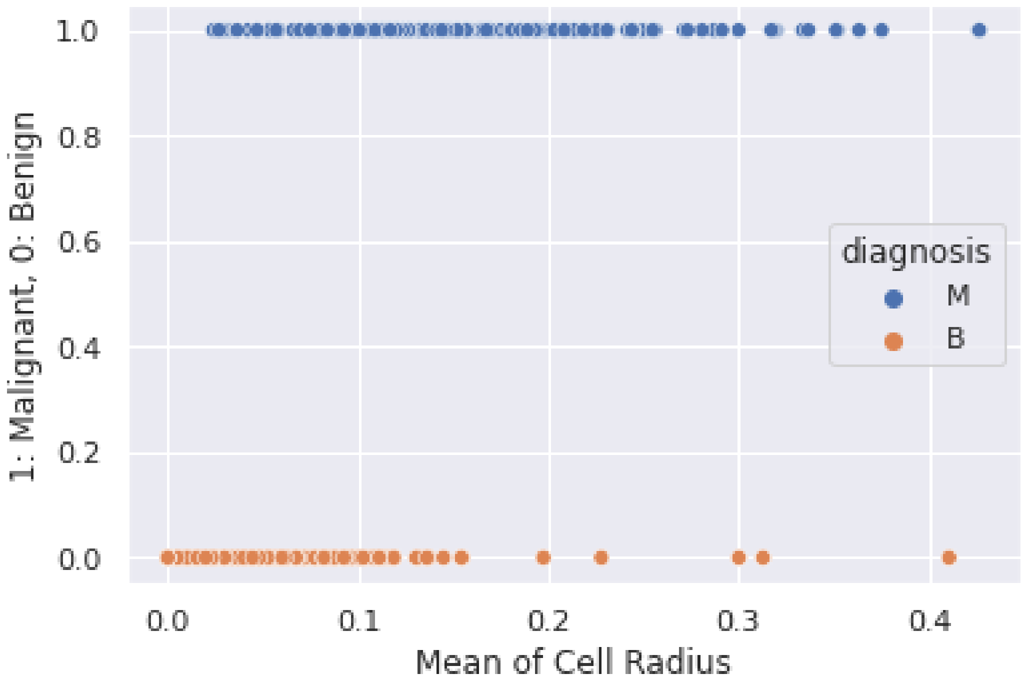
```
In [18]: df["radius_mean"]

Out[18]: 0      17.99
         1      20.57
         2      19.69
         3      11.42
         4      20.29
                ...
         564    21.56
         565    20.13
         566    16.60
         567    20.60
         568     7.76
         Name: radius_mean, Length: 569, dtype: float64
```

- **Task 8:** Use scatter plot to see the relationship between "concavity_mean" and "diagnosis_int".
  - Use the "diagnosis" column as the `hue` for the scatter plot.
  - Label the x axis as "Mean of Cell Concavity"
  - Label the y axis as "1: Malignant, 0: Benign"

```
In [19]: # Task 7
         # set the default theme to use Seaborn
         sns.set()
```

```
# display using scatter plot
myplot = sns.scatterplot(data = df, x = "concavity_mean",y= "diagnosis_int",hue = df['

myplot.set(xlabel="Mean of Cell Radius", ylabel = "1: Malignant, 0: Benign")
```

```
Out[19]: [Text(0.5, 0, 'Mean of Cell Radius'), Text(0, 0.5, '1: Malignant, 0: Benign')]
```



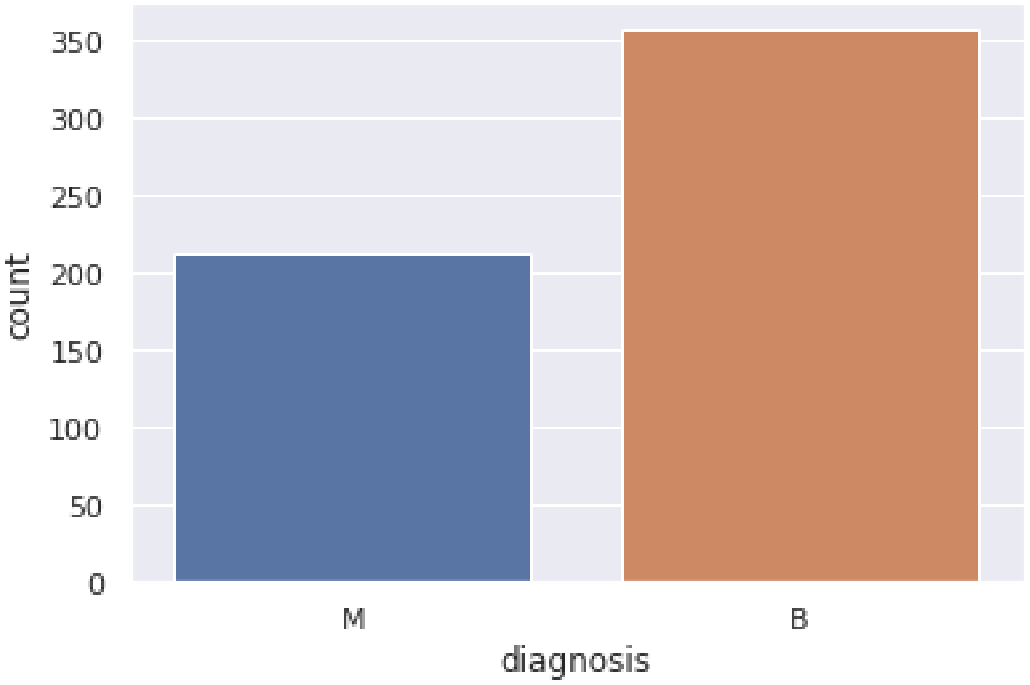
HW2. *Count Plot:* Create a function to count how many records are diagnosed as Malignant and Benign. The function should return a tuple: `(n_malignant, n_benign)`, where `n_malignant` is the number of records diagnosed as Malignant cell and `n_benign` is the number of records diagnosed as Benign.

Use Count plot to verify the answer.

Reference: - Count Plot

```
In [20]: def count_cell(target):
             n_malignant = 0
             n_benign = 0
             for item in target:
                 if item == "M":
                     n_malignant += 1
                 else:
                     n_benign += 1
             return (n_malignant, n_benign)
```

```
In [21]: result = count_cell(df_target["diagnosis"])
         assert result == (212, 357)

In [22]: result

Out[22]: (212, 357)

In [23]: # write the code to plot the count of the two classes
         myplot = sns.countplot(x="diagnosis",data = df)


         ###
         ### YOUR CODE HERE
         ###
```



HW3. *Normalization:* Create a function that takes in Data Frame as the input and returns the normalized Data Frame as the output. **Each column** is normalized separately using **min-max normalization**. The function should return a new data frame instead of modifying the input data frame.

$$normalized = \frac{data - min}{max - min}$$

Use the following functions from Pandas or Numpy: - `df.copy()`: This is to create a new copy of the data frame. - `df.min(axis=0)`: This is to get the minimum along the index axis. - `df.max(axis=0)`: This is to get the maximum along the index axis.

Note: Your function should be able to handle a numpy array as well as Panda's data frame.

```
In [1]: def normalize_minmax(dfin):

            dfout = (dfin - dfin.min(axis=0))/(dfin.max(axis=0)-dfin.min(axis=0))

            return dfout
```

```
In [2]: data_norm = normalize_minmax(df_features.to_numpy())
        assert data_norm[:,0].max() == 1.0 and \
               data_norm[:,0].min() == 0 and \
               np.isclose(data_norm[:,0].mean(), 0.338222)
        assert data_norm[:,1].max() == 1.0 and \
               data_norm[:,1].min() == 0 and\
               np.isclose(data_norm[:,1].mean(), 0.323965)


        ---------------------------------------------------------------------------

        NameError                                 Traceback (most recent call last)

        <ipython-input-2-c65c2397dfc5> in <module>
    ----> 1 data_norm = normalize_minmax(df_features.to_numpy())
          2 assert data_norm[:,0].max() == 1.0 and \
          3        data_norm[:,0].min() == 0 and \
          4        np.isclose(data_norm[:,0].mean(), 0.338222)
          5 assert data_norm[:,1].max() == 1.0 and \


        NameError: name 'df_features' is not defined
```

```
In [26]: data_norm = normalize_minmax(df_features)
         stats = data_norm.describe()
         display(stats)
         assert stats.loc["max", "radius_mean"] == 1.0 and \
                stats.loc["min", "radius_mean"] == 0 and \
                np.isclose(stats.loc["mean", "radius_mean"], 0.338222)
         assert stats.loc["max", "texture_mean"] == 1.0 and \
                stats.loc["min", "texture_mean"] == 0 and\
                np.isclose(stats.loc["mean", "texture_mean"], 0.323965)
```

```
       radius_mean  texture_mean  perimeter_mean   area_mean  smoothness_mean  \
count   569.000000    569.000000      569.000000  569.000000       569.000000
mean      0.338222      0.323965        0.332935    0.216920         0.394785
std       0.166787      0.145453        0.167915    0.149274         0.126967
min       0.000000      0.000000        0.000000    0.000000         0.000000
25%       0.223342      0.218465        0.216847    0.117413         0.304595
50%       0.302381      0.308759        0.293345    0.172895         0.390358
75%       0.416442      0.408860        0.416765    0.271135         0.475490
max       1.000000      1.000000        1.000000    1.000000         1.000000
```

```
       concavity_mean
count      569.000000
mean         0.208058
std          0.186785
min          0.000000
25%          0.069260
50%          0.144189
75%          0.306232
max          1.000000
```

```
In [27]: data_norm = normalize_minmax(df_features.to_numpy())
         assert data_norm[:,0].max() == 1.0 and \
                data_norm[:,0].min() == 0 and \
                np.isclose(data_norm[:,0].mean(), 0.338222)
         assert data_norm[:,1].max() == 1.0 and \
                data_norm[:,1].min() == 0 and\
                np.isclose(data_norm[:,1].mean(), 0.323965)
```

HW4. *Splitting the Data:* Use the function to split the breast cancer data set into a training data set and a testing data set. Use `random_state=100` and `test_size=0.3` and the normalized features data set from the previous exercise.

```
In [31]: def split_data(df_feature, df_target, random_state=None, test_size=0.5):
             np.random.seed(random_state)
             N = df_feature.shape[0]
             print(N)
             sample = int(test_size*N)

             train_idx = np.random.choice(N, N-sample,replace=False)

             df_feature_train = df_feature.iloc[train_idx]
             df_target_train = df_target.iloc[train_idx]

             test_idx = [idx for idx in range(N) if idx not in train_idx]
             print(len(test_idx))

             df_feature_test = df_feature.iloc[test_idx]
             df_target_test = df_target.iloc[test_idx]

             return df_feature_train, df_feature_test, df_target_train, df_target_test

In [35]: # call normalize_minmax() to normalize df_features
         data_norm = data_norm = normalize_minmax(df_features)


         ###
         ### YOUR CODE HERE
```

```
    ###

    # call split_data() with seed of 100 and test size of 30%
    datasets_tupple = split_data(data_norm, df_target, 100, 0.3)

    ###
    ### YOUR CODE HERE
    ###

569
170
```

```
In [36]: df_features_train = datasets_tupple[0]
         df_features_test = datasets_tupple[1]
         df_target_train = datasets_tupple[2]
         df_target_test = datasets_tupple[3]

         display(df_features_train.describe())
         display(df_features_test.describe())
         display(df_target_train.describe())
         display(df_target_test.describe())
```

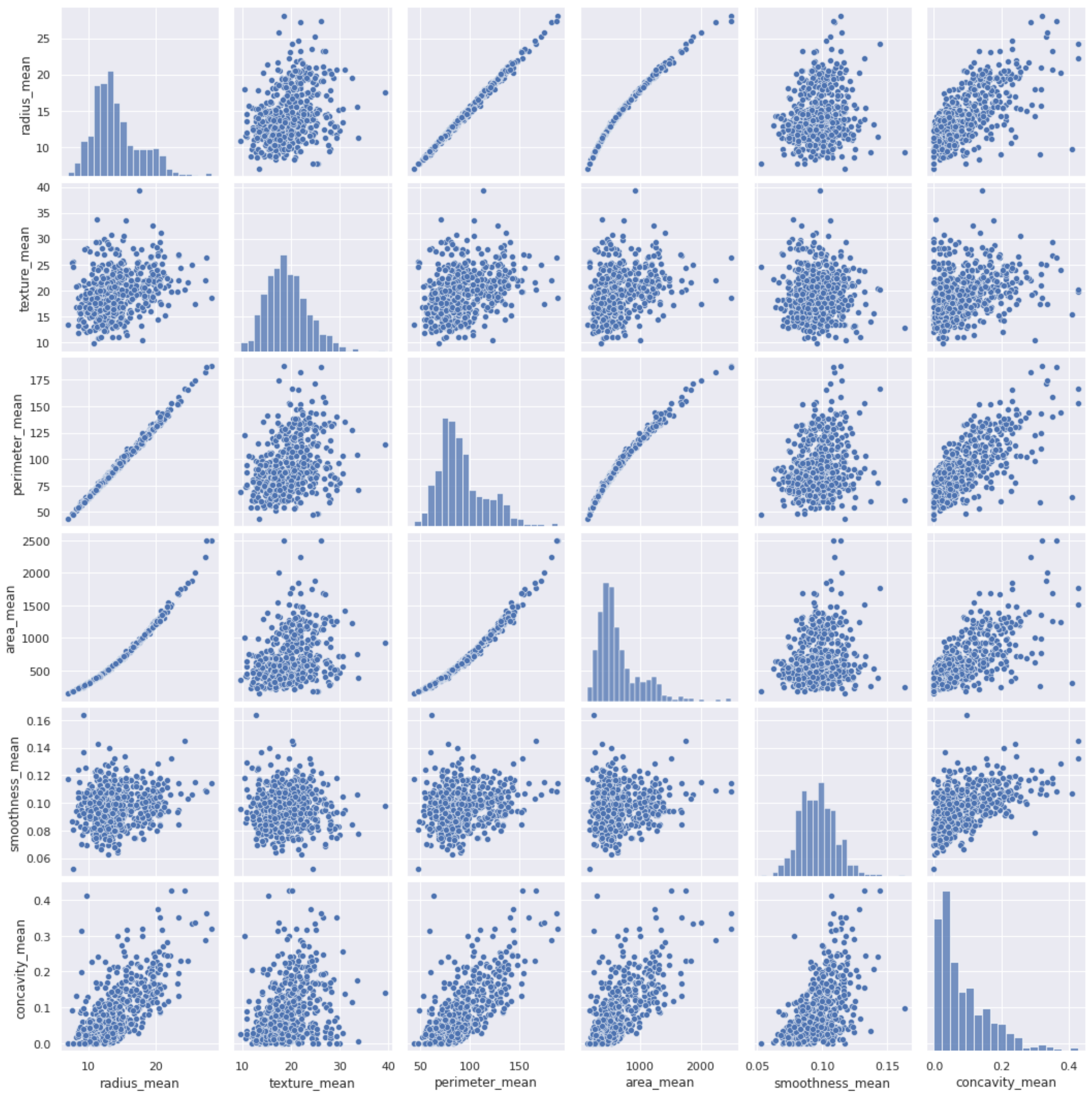
```
       radius_mean  texture_mean  perimeter_mean   area_mean  smoothness_mean  \
count   399.000000    399.000000      399.000000  399.000000       399.000000
mean      0.333567      0.324109        0.328153    0.212882         0.388423
std       0.163565      0.146559        0.164450    0.145811         0.124730
min       0.000000      0.000000        0.000000    0.000000         0.089194
25%       0.219319      0.225059        0.213358    0.114125         0.299675
50%       0.294335      0.308083        0.289545    0.167508         0.379706
75%       0.406503      0.405140        0.408127    0.258494         0.467365
max       0.967343      1.000000        0.988943    1.000000         0.811321

       concavity_mean
count      399.000000
mean         0.205096
std          0.187089
min          0.000000
25%          0.065628
50%          0.138051
75%          0.288308
max          0.999063


       radius_mean  texture_mean  perimeter_mean   area_mean  smoothness_mean  \
count   170.000000    170.000000      170.000000  170.000000       170.000000
mean      0.349148      0.323627        0.344158    0.226399         0.409716
std       0.174122      0.143249        0.175766    0.157139         0.131232
min       0.036869      0.022658        0.028540    0.015907         0.000000
```

```
25%      0.238251      0.206713       0.232292    0.127126          0.312336
50%      0.317526      0.310112       0.306060    0.183351          0.406157
75%      0.437385      0.411989       0.445443    0.283372          0.487000
max      1.000000      0.815015       1.000000    0.999152          1.000000

       concavity_mean
count      170.000000
mean         0.215012
std          0.186435
min          0.000000
25%          0.080325
50%          0.155155
75%          0.315194
max          1.000000


       diagnosis_int
count     399.000000
mean        0.353383
std         0.478621
min         0.000000
25%         0.000000
50%         0.000000
75%         1.000000
max         1.000000


       diagnosis_int
count     170.000000
mean        0.417647
std         0.494628
min         0.000000
25%         0.000000
50%         0.000000
75%         1.000000
max         1.000000
```

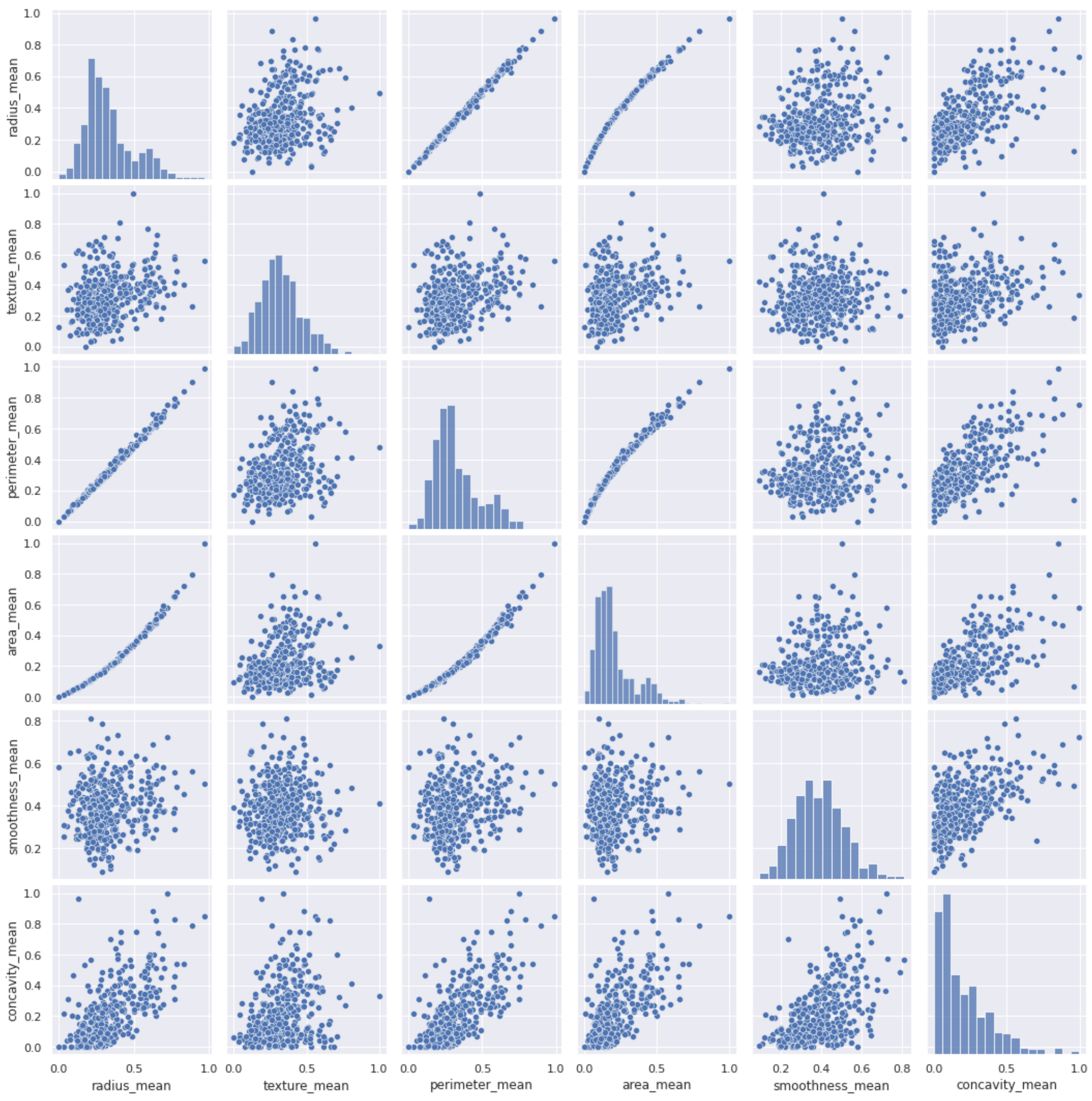
```
In [ ]: assert isinstance(df_features_train, pd.DataFrame)
        assert isinstance(df_features_test, pd.DataFrame)
        assert isinstance(df_target_train, pd.DataFrame)
        assert isinstance(df_target_test, pd.DataFrame)

        assert df_features_train.shape == (399, 6)
        assert df_features_test.shape == (170, 6)
        assert df_target_train.shape == (399, 2)
        assert df_target_test.shape == (170, 2)
        assert np.isclose(df_features_train.mean().mean(), 0.29844)
        assert np.isclose(df_features_test.mean().mean(), 0.311966)
        assert np.isclose(df_target_train.mean().mean(), 0.358396)
        assert np.isclose(df_target_test.mean().mean(), 0.40588)
```

HW5. *Pair Plot:* Use pair plot to find out the relationship between different columns in df_features. Ensure that similar relationship exists in both the training and the test datasets.

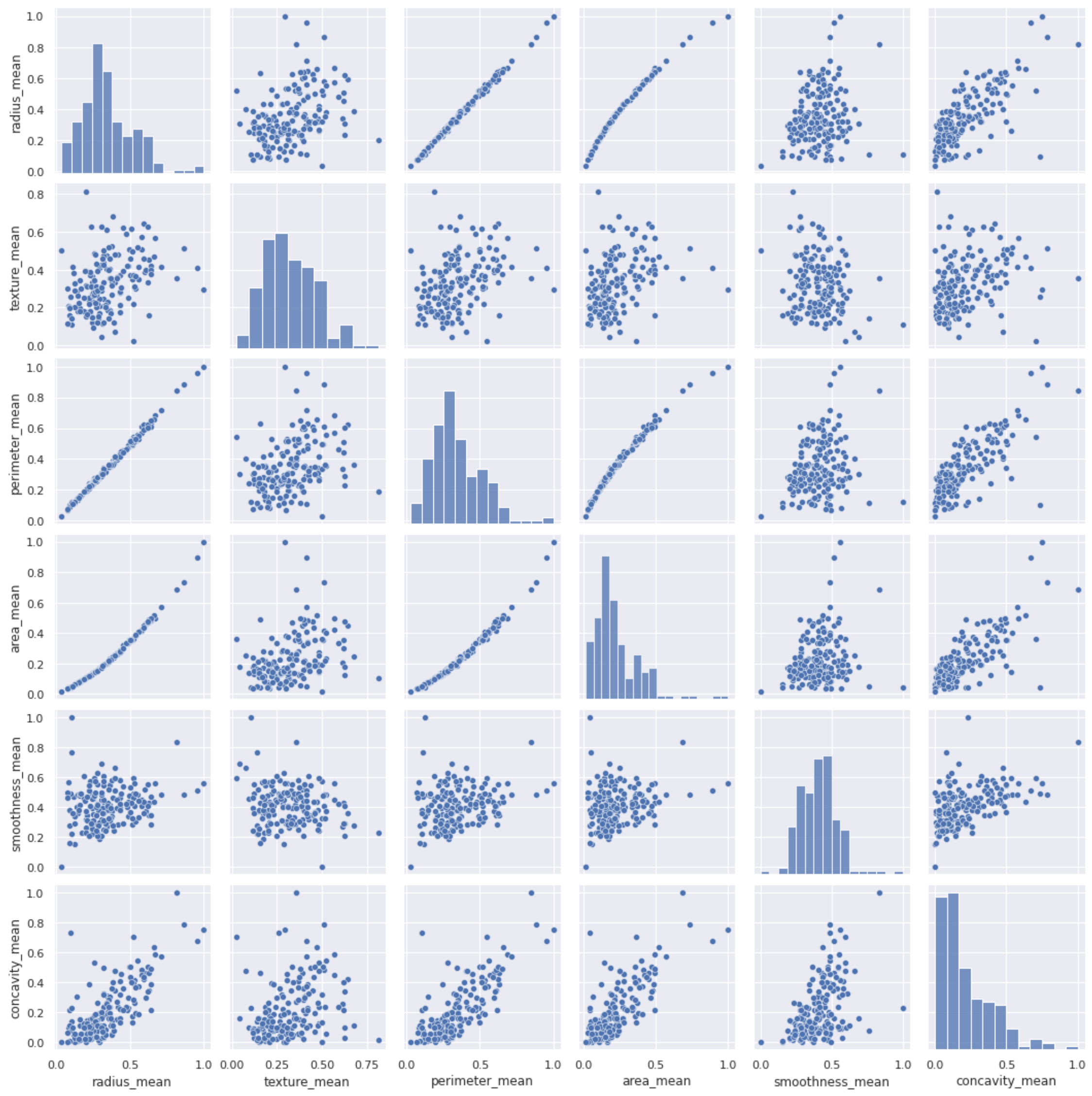
```
In [37]: # write your code below to plot for df_features

         myplot = sns.pairplot(data=df_features)
```



```
In [38]: # write your code below to plot for df_features_train
         myplot = sns.pairplot(data=df_features_train)
```



```
In [39]: # write your code below to plot for df_features_test
         myplot = sns.pairplot(data=df_features_test)
```



radius_mean
texture_mean
perimeter_mean
area_mean
smoothness_mean
concavity_mean

```
In [ ]:
```

Week09_Cohort

December 10, 2021

1 Week 9 Problem Set

1.1 Cohort Session

```
In [54]: import numpy as np
         import pandas as pd
         import matplotlib.pyplot as plt
         import seaborn as sns
```

CS0. *Plot:* Read data for Boston Housing Prices and write a function `get_features_targets()` to get the columns for the features and the targets from the input argument data frame. The function should take in Pandas' dataframe and two lists. The first list is for the feature names and the other list is for the target names.

We will use the following columns for our test cases: - x data: RM column - use z normalization (standardization) - y data: MEDV column

Make sure you return a new data frame for both the features and the targets.

We will normalize the feature using z normalization. Plot the data using scatter plot.

```
In [55]: def normalize_z(dfin):

             dfout = (dfin - dfin.mean(axis=0))/dfin.std(axis=0)

             return dfout
```

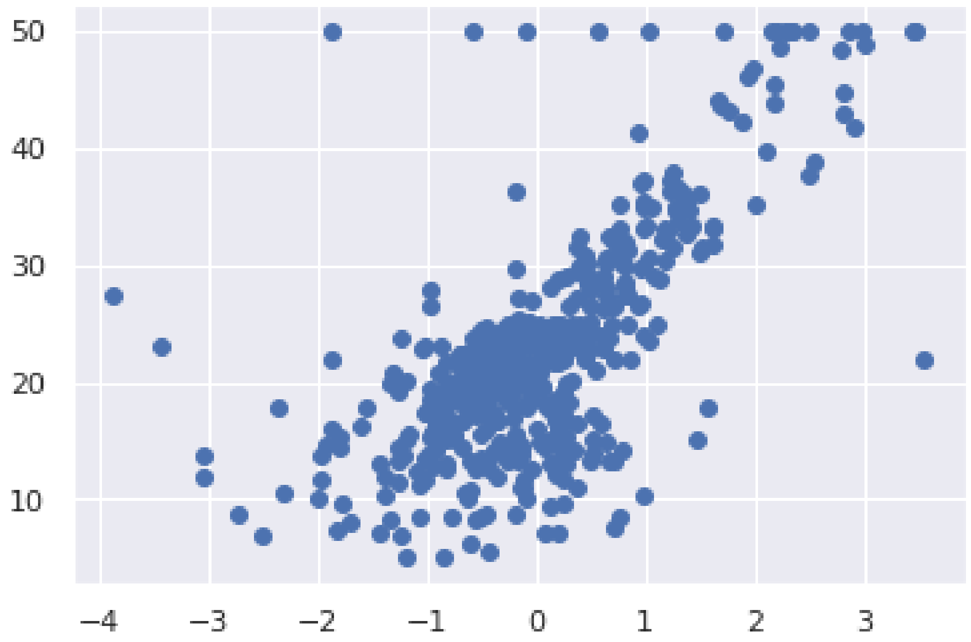
```
In [56]: def get_features_targets(df, feature_names, target_names):
             df_feature = df[feature_names]
             df_target = df[target_names]
             return df_feature, df_target
```

```
In [57]: df = pd.read_csv("housing_processed.csv")
         df_feature, df_target = get_features_targets(df,["RM"],["MEDV"])
         df_feature = normalize_z(df_feature)

         assert isinstance(df_feature, pd.DataFrame)
         assert isinstance(df_target, pd.DataFrame)
         assert np.isclose(df_feature.mean(), 0.0)
         assert np.isclose(df_feature.std(), 1.0)
         assert np.isclose(df_target.mean(), 22.532806)
         assert np.isclose(df_target.std(), 9.1971)
```

```
In [58]: ###
         ### AUTOGRADER TEST - DO NOT REMOVE
         ###
```

```
In [59]: sns.set()
         plt.scatter(df_feature, df_target)
```

```
Out[59]: <matplotlib.collections.PathCollection at 0x7f32aa631690>
```



CS1. *Cost Function:* Write a function `compute_cost()` to compute the cost function of a linear regression model. The function should take in two 2-D numpy arrays. The first one is the matrix of the linear equation and the second one is the actual target value.

Recall that:

$$J(\hat{\beta}_0, \hat{\beta}_1) = \frac{1}{2m}\Sigma_{i=1}^m \left(\hat{y}(x^i) - y^i\right)^2$$

where

$$\hat{y}(x^i) = \hat{\beta}_0 + \hat{\beta}_1 x^i$$

The function should receive a numpy array, so we will need to convert to numpy array and change the shape. To do this, we will create two other functions: - `prepare_feature(df)`: which takes in a data frame for the feature. The function should convert the data frame to a numpy array and change it into a column vector. The function should also add a column of constant 1s in the first column. - `prepare_target(df)`: which takes in a data frame for the target. The function should simply convert the data frame to a numpy array and change it into column vectors. **The function should be able to handle if the data frame has more than one column.**

You can use the following methods in your code: - `df.to_numpy()`: which is to convert a Pandas data frame to Numpy array. - `np.reshape(row, col)`: which is to reshape the numpy array to a particular shape. - `np.concatenate((array1, array2), axis)`: which is to join a sequence of arrays along an existing axis. - `np.matmul(array1, array2)`: which is to do matrix multiplication on two Numpy arrays.

```
In [60]: def compute_cost(X, y, beta):
             m = X.shape[0]
             y_pred = np.matmul(X,beta)
             error = y_pred - y

             # Matrix multiplcation does both sum and square
             J = (1/(2*m))*np.matmul(error.T,error)

             return J[0][0] # Extract a scalar value from the 1 x 1 matrix

In [61]: def prepare_feature(df_feature):
             n = df_feature.shape[0]
             ones = np.ones(n).reshape(n,1)
             return np.concatenate((ones,df_feature.to_numpy()),axis = 1)



         # # df_feature is a 506 rows x 1 columns dataframe that gives the average number of r
         # # Column of constants 1 is on the left
         # # Column on the right is the feature data
         # # This function is to get the hypothesis matrix which is an 506 x 2 matrix
         # def prepare_feature(df_feature):
         #     cols = len(df_target.columns)
         #     feature = df_feature.to_numpy().reshape(-1, cols)
         #     ones = np.ones((feature.shape[0],1))
         #     X = np.concatenate((ones, feature), axis=1)
         # #     X = np.concatenate((np.ones((feature.shape[0],1)),feature), axis=1)
         #     return X

In [62]: # def prepare_target(df_feature):
         #     return df_feature.to_numpy()

         # Converts a dataframe to a numpy array
         # One shape dimension can be -1. In this case, the value is inferred from the length
         def prepare_target(df_target):
             cols = len(df_target.columns)
             target = df_target.to_numpy().reshape(-1, cols)
             return target

In [63]: X = prepare_feature(df_feature)
         target = prepare_target(df_target)

         assert isinstance(X, np.ndarray)
         assert isinstance(target, np.ndarray)
```

```
        assert X.shape == (506, 2)
        assert target.shape == (506, 1)
```

```
In [64]: ###
         ### AUTOGRADER TEST - DO NOT REMOVE
         ###
```

```
In [65]: # print(X)
         beta = np.zeros((2,1))
         J = compute_cost(X, target, beta)
         print(J)
         assert np.isclose(J, 296.0735)

         beta = np.ones((2,1))
         J = compute_cost(X, target, beta)
         print(J)
         assert np.isclose(J, 268.157)

         beta = np.array([-1, 2]).reshape((2,1))
         J = compute_cost(X, target, beta)
         print(J)
         assert np.isclose(J, 308.337)
```

```
296.07345849802374
268.1570051486897
308.33699448710274
```

```
In [66]: ###
         ### AUTOGRADER TEST - DO NOT REMOVE
         ###
```

```
In [67]: bx = beta.transpose()
```

```
In [68]: bx.shape
```

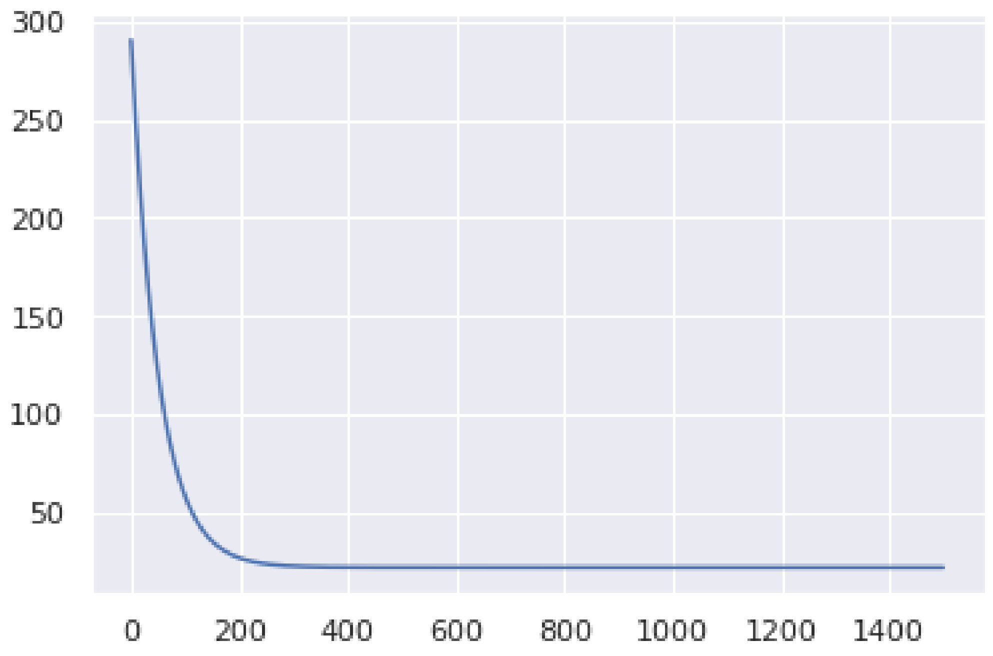
```
Out[68]: (1, 2)
```

CS2. *Gradient Descent:* Write a function called `gradient_descent()` that takes in these parameters: - `X`: is a 2-D numpy array for the features - `y`: is a vector array for the target - `beta`: is a column vector for the initial guess of the parameters - `alpha`: is the learning rate - `num_iters`: is the number of iteration to perform

The function should return two numpy arrays: - `beta`: is coefficient at the end of the iteration - `J_storage`: is the array that stores the cost value at each iteration

You can use some of the following functions: - `np.matmul(array1, array2)`: which is to do matrix multiplication on two Numpy arrays. - `compute_cost()`: which the function you created in the previous problem set to compute the cost.

```
In [69]: # 1. guess beta
         # 2. Decide direction
         # 3. updadte beta

         def gradient_descent(X, y, beta, alpha, num_iters):
             m = X.shape[0]
             J_storage = np.zeros(num_iters)

             for i in range(num_iters):
                 yp = np.matmul(X,beta)
                 error = yp - y
                 beta = beta - (alpha/m) * np.matmul(X.T,error)
         #         beta = beta - (alpha/m)*np.matmul(X.T,np.matmul(X,beta)-y)
                 cost = compute_cost(X,y,beta)
         #         print(cost)
                 J_storage[i] = cost
         #         print(J_storage[i])


             return beta, J_storage

In [70]: iterations = 1500
         alpha = 0.01
         beta = np.zeros((2,1))

         beta, J_storage = gradient_descent(X, target, beta, alpha, iterations)
         print(beta)
         assert np.isclose(beta[0], 22.5328)
         assert np.isclose(beta[1], 6.3953)

[[22.53279993]
 [ 6.39529594]]


In [71]: ###
         ### AUTOGRADER TEST - DO NOT REMOVE
         ###

In [72]: plt.plot(J_storage)

Out[72]: [<matplotlib.lines.Line2D at 0x7f32aa5b5310>]
```



CS3. *Predict:* Write two functions `predict()` and `predict_norm()` that calculate the straight line equation given the features and its coefficient. - `predict()`: this function should standardize the feature using z normalization, change it to a Numpy array, and add a column of constant 1s. You should use `prepare_feature()` for this purpose. Lastly, this function should also call `predict_norm()` to get the predicted y values. - `predict_norm()`: this function should calculate the straight line equation after standardization and adding of column for constant 1.

You can use some of the following functions: - `np.matmul(array1, array2)`: which is to do matrix multiplication on two Numpy arrays. - `normalize_z(df)`: which is to do z normalization on the data frame.

```
In [73]: def normalize_z(dfin):

             dfout = (dfin - dfin.mean(axis=0))/dfin.std(axis=0)
             return dfout

         def predict_norm(X, beta):
             return np.matmul(X,beta)


In [74]: def predict(df_feature, beta):
             df_feature = normalize_z(df_feature)
             X = prepare_feature(df_feature)

             return predict_norm(X, beta)

In [75]: df_feature, buf = get_features_targets(df, ["RM"], [])
         beta = [[22.53279993],[ 6.39529594]] # from previous result
         pred = predict(df_feature, beta)

         assert isinstance(pred, np.ndarray)
         assert pred.shape == (506, 1)
         assert np.isclose(pred.mean(), 22.5328)
         assert np.isclose(pred.std(), 6.38897)
```

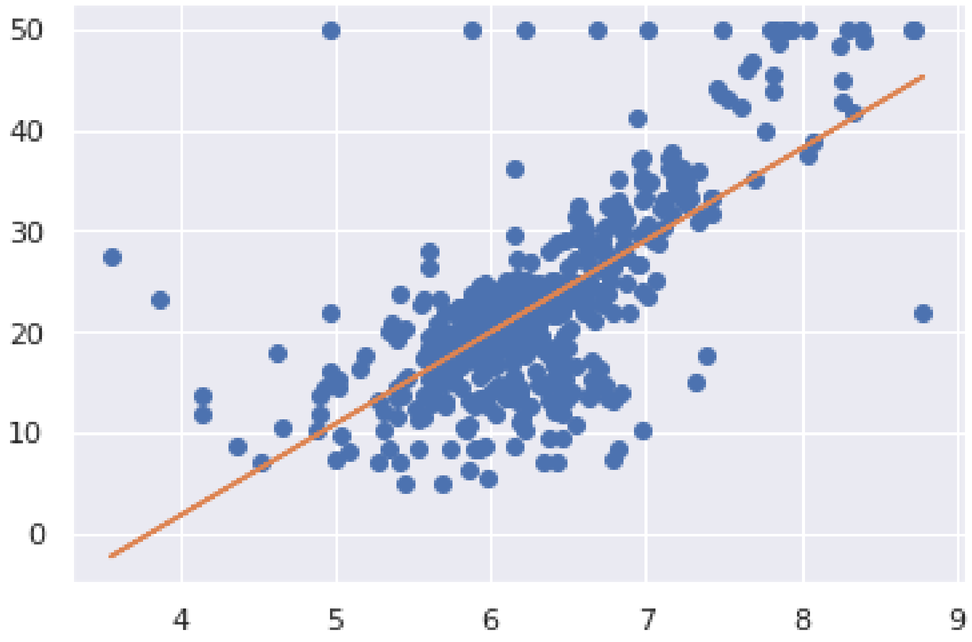
```
In [76]: df_feature, buf = get_features_targets(df, ["RM"], [])
         beta = [[22.53279993],[ 6.39529594]] # from previous result
         df_feature.shape
```

```
Out[76]: (506, 1)
```

```
In [77]: ###
         ### AUTOGRADER TEST - DO NOT REMOVE
         ###
```

```
In [78]: plt.plot(df_feature["RM"],target,'o')
         plt.plot(df_feature["RM"],pred,'-')
```

```
Out[78]: [<matplotlib.lines.Line2D at 0x7f32aa53b710>]
```



CS4. *Splitting data:* Do the following tasks: - Read RM as the feature and MEDV as the target from the data frame. - Use Week 9's function `split_data()` to split the data into train and test using `random_state=100` and `test_size=0.3`. - Normalize and prepare the features and the target. - Use the training data set and call `gradient_descent()` to obtain the `theta`. - Use the test data set to get the predicted values.

You need to replace the `None` in the code below with other a function call or any other Python expressions.

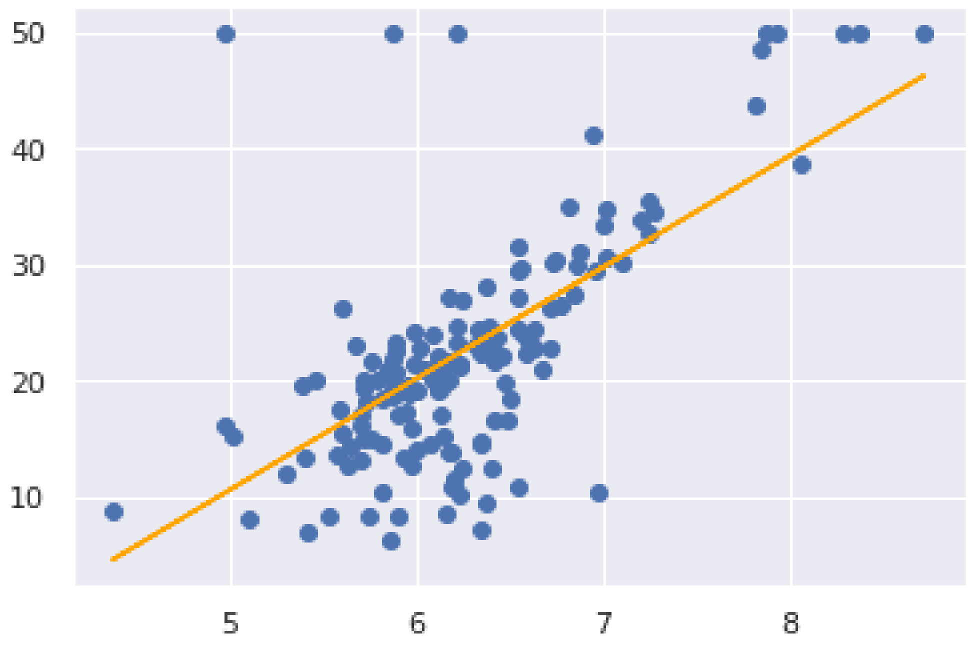
In [97]:
```
# def split_data(df_feature, df_target, random_state=None, test_size=0.5):
#     np.random.seed(random_state)
#     N = df_feature.shape[0]
#     sample = int(test_size*N)

#     train_idx = np.random.choice(N, N-sample,replace=False)

#     df_feature_train = df_feature.iloc[train_idx]
#     df_target_train = df_target.iloc[train_idx]

#     test_idx = [idx for idx in range(N) if idx not in train_idx]

#     df_feature_test = df_feature.iloc[test_idx]
#     df_target_test = df_target.iloc[test_idx]

#     return df_feature_train, df_feature_test, df_target_train, df_target_test


def split_data(df_feature, df_target, random_state=None, test_size=0.5):
    np.random.seed(random_state)
    TestSize = int(test_size*len(df_feature))
    testchoice = np.random.choice(len(df_feature),size = TestSize, replace = False)
#     print(TestSize)
#     print(testchoice)
    remainder = []
    for i in df_feature.index:
        if i not in testchoice:
            remainder.append(i)
#     print(remainder)
    trainchoice = np.random.choice(remainder, size = len(remainder),replace = False)
#     print(list(trainchoice))

    df_feature_train = df_feature.iloc[trainchoice]
    df_target_train = df_target.iloc[trainchoice]
    df_feature_test = df_feature.iloc[testchoice]
    df_target_test = df_target.iloc[testchoice]


    return df_feature_train, df_feature_test, df_target_train, df_target_test
```

In [98]:
```
# get features and targets from data frame
df_feature, df_target = get_features_targets(df,["RM"],["MEDV"])

# split the data into training and test data sets
df_feature_train, df_feature_test, df_target_train, df_target_test = split_data(df_fea

# normalize the feature using z normalization
```

```
df_feature_train_z = normalize_z(df_feature_train)

X = prepare_feature(df_feature_train_z) # concatenating for the y intercept
target = prepare_target(df_target_train)

iterations = 1500
alpha = 0.01
beta = np.zeros((2,1)) #[0,0]

# call the gradient_descent function
beta, J_storage = gradient_descent(X, target, beta, alpha, iterations)

# call the predict method to get the predicted values
df_feature_test_z = normalize_z(df_feature_test)
pred = predict(df_feature_test_z,beta)
print(beta)
###
### YOUR CODE HERE
###
```

```
[[22.66816258]
 [ 6.27808747]]
```

```
In [99]: J_storage

Out[99]: array([290.6613454 , 285.26808904, 279.98213717, ...,  19.58945173,
                 19.58945173,  19.58945173])

In [100]: assert isinstance(pred, np.ndarray)
          assert pred.shape == (151, 1)
          assert np.isclose(pred.mean(), 22.66816)
          assert np.isclose(pred.std(), 6.257265)

In [101]: ###
          ### AUTOGRADER TEST - DO NOT REMOVE
          ###

In [102]: plt.scatter(df_feature_test, df_target_test)
          plt.plot(df_feature_test, pred, color="orange")

Out[102]: [<matplotlib.lines.Line2D at 0x7f32aa4c7d10>]
```



CS5. *R2 Coefficient of Determination:* Write a function to calculate the coefficient of determination as given by the following equations.

$$r^2 = 1 - \frac{SS_{res}}{SS_{tot}}$$

where

$$SS_{res} = \Sigma_{i=1}^{n}(y_i - \hat{y}_i)^2$$

where y_i is the actual target value and $\hat{y}_i$ is the predicted target value.

$$SS_{tot} = \Sigma_{i=1}^{n}(y_i - \overline{y})^2$$

where

$$\overline{y} = \frac{1}{n}\Sigma_{i=1}^{n}y_i$$

and n is the number of target values.

You can use the following functions in your code: - `np.mean(array)`: which is to get the mean of the array. You can also call it using `array.mean()`. - `np.sum(array)`: which is to sum the array along a default axis. You can specify which axis to perform the summation.

```
In [103]: y = np.array([1,2,3])
          y_mean = np.mean(y)

In [104]: np.sum(y)

Out[104]: 6
```

```
In [105]: np.sum(np.square(y))

Out[105]: 14

In [106]: np.matmul(y.T,y)

Out[106]: 14

In [111]: def r2_score(y, ypred):
              rss = np.sum((ypred - y) ** 2)
              tss = np.sum((y-y.mean()) ** 2)

              r2 = 1 - (rss / tss)

              return r2

In [112]: target = prepare_target(df_target_test)
          r2 = r2_score(target, pred)
          assert np.isclose(r2, 0.45398)


        ---------------------------------------------------------------------------

        IndexError                                Traceback (most recent call last)

        <ipython-input-112-7601a9c51c7e> in <module>
          1 target = prepare_target(df_target_test)
    ----> 2 r2 = r2_score(target, pred)
          3 assert np.isclose(r2, 0.45398)


        <ipython-input-111-341873669eea> in r2_score(y, ypred)
          2     y_mean = np.mean(y)
          3     ss_res = np.matmul((y-ypred).T,(y-ypred))[0][0]
    ----> 4     ss_total = np.sum(np.matmul((y-y_mean).T,(y-y_mean)))[0][0]
          5     return 1 - (ss_res/ss_total)
          6


        IndexError: invalid index to scalar variable.
```

CS6. *Mean Squared Error:* Create a function to calculate the MSE as given below.

$$MSE = \frac{1}{n}\Sigma_{i=1}^{n}(y^i - \hat{y}^i)^2$$

```
In [109]: def mean_squared_error(target, pred):
              n = target.shape[0]
              error = target-pred
```

```
        mse = np.matmul(error.T,error)/n

        return mse
```

```
In [110]: mse = mean_squared_error(target, pred)
          assert np.isclose(mse, 53.6375)
```

CS8. *Optional:* Redo the above tasks using Sci-kit learn libraries. You will need to use the following: - LinearRegression - train_test_split - r2_score - mean_squared_error

```
In [ ]: from sklearn.linear_model import LinearRegression
        from sklearn.model_selection import train_test_split
        from sklearn.metrics import r2_score, mean_squared_error
```

```
In [ ]: # Read the CSV and extract the features
        df = None
        df_feature, df_target = None, None
        # normalize
        df_feature = None

        ###
        ### YOUR CODE HERE
        ###
```

```
In [ ]: # Split the data into training and test data set using scikit-learn function
        df_feature_train, df_feature_test, df_target_train, df_target_test = None, None, None,

        # Instantiate LinearRegression() object
        model = None

        # Call the fit() method
        pass

        ###
        ### YOUR CODE HERE
        ###

        print(model.coef_, model.intercept_)
        assert np.isclose(model.coef_,[6.05090511])
        assert np.isclose(model.intercept_, 22.52999668)
```

```
In [ ]: # Call the predict() method
        pred = None

        ###
        ### YOUR CODE HERE
        ###

        print(type(pred), pred.mean(), pred.std())
        assert isinstance(pred, np.ndarray)
        assert np.isclose(pred.mean(), 22.361699)
        assert np.isclose(pred.std(), 5.7011267)

In [ ]: plt.scatter(df_feature_test, df_target_test)
        plt.plot(df_feature_test, pred, color="orange")

In [ ]: r2 = r2_score(df_target_test, pred)
        print(r2)
        assert np.isclose(r2, 0.457647)

In [ ]: mse = mean_squared_error(df_target_test, pred)
        print(mse)
        assert np.isclose(mse, 54.93216)
```

Week09_Homework

December 10, 2021

1 Week 9 Problem Set

1.1 Homeworks

```
In [41]: import numpy as np
         import pandas as pd
         import matplotlib.pyplot as plt
         import seaborn as sns
```

HW0. Copy and paste some of the functions from you cohort sessions and previous exercises that you will need in this homework. See below template and the list here: - `normalize_z()` - `get_features_targets()` - `prepare_feature()` - `prepare_target()` - `predict()` - `predict_norm()` - `split_data()` - `r2_score()` - `mean_squared_error()`

Then do the following: - Read the CSV file `housing_processed.csv` and extract the following columns: - x data: RM, DIS, and INDUS columns - y data: MEDV column - Normalize the features using z normalization. - Plot the data using scatter plot. Use the following columns:

```
In [42]: def normalize_z(dfin):

             dfout = (dfin - dfin.mean(axis=0))/dfin.std(axis=0)

             return dfout


         def get_features_targets(df, feature_names, target_names):
             df_feature = df.loc[:,feature_names]
             df_target = df.loc[:,target_names]

             return df_feature, df_target

         def prepare_feature(df_feature):
             n = df_feature.shape[0]
             ones = np.ones(n).reshape(n,1)

             return np.concatenate((ones,df_feature.to_numpy()),axis = 1)

         def prepare_target(df_feature):
             return df_feature.to_numpy()
```

```python
def predict(df_feature, beta):
    df_feature = normalize_z(df_feature)
    X = prepare_feature(df_feature)

    return predict_norm(X, beta)

def predict_norm(X, beta):
    return np.matmul(X,beta)


def split_data(df_feature, df_target, random_state=None, test_size=0.5):
    np.random.seed(random_state)
    TestSize = int(test_size*len(df_feature))
    testchoice = np.random.choice(len(df_feature),size = TestSize, replace = False)
    remainder = []
    for i in df_feature.index:
        if i not in testchoice:
            remainder.append(i)
    trainchoice = np.random.choice(remainder, size = len(remainder),replace = False)

    df_feature_train = df_feature.iloc[trainchoice]
    df_target_train = df_target.iloc[trainchoice]
    df_feature_test = df_feature.iloc[testchoice]
    df_target_test = df_target.iloc[testchoice]


    return df_feature_train, df_feature_test, df_target_train, df_target_test

def r2_score(y, ypred):
#     y_mean = np.mean(y)
#     ss_res = np.matmul((y-ypred).T,(y-ypred))[0][0]
#     ss_total = np.sum(np.matmul((y-y_mean).T,(y-y_mean)))[0][0]
#     return 1 - (ss_res/ss_total)

    rss = np.sum((ypred - y) ** 2)
    tss = np.sum((y-y.mean()) ** 2)

    r2 = 1 - (rss / tss)

    return r2

def mean_squared_error(target, pred):
    n = target.shape[0]
    error = target-pred
    mse = np.matmul(error.T,error)/n

    return mse
```

```
In [43]: # Read the CSV file
         df = pd.read_csv('housing_processed.csv')

         # Extract the features and the targets
         df_features, df_target = get_features_targets(df,["RM", "DIS","INDUS"],["MEDV"])

         # Normalize using z normalization
         df_features = normalize_z(df_features)
```

```
In [44]: ###
         ### AUTOGRADER TEST - DO NOT REMOVE
         ###
```

```
In [45]: display(df_features.describe())
         display(df_target.describe())
         assert np.isclose(df_features['RM'].mean(), 0)
         assert np.isclose(df_features['DIS'].mean(), 0)
         assert np.isclose(df_features['INDUS'].mean(), 0)

         assert np.isclose(df_features['RM'].std(), 1)
         assert np.isclose(df_features['DIS'].std(), 1)
         assert np.isclose(df_features['INDUS'].std(), 1)

         assert np.isclose(df_target['MEDV'].mean(), 22.532806)
         assert np.isclose(df_target['MEDV'].std(), 9.197104)

         assert np.isclose(df_features['RM'].median(), -0.1083583)
         assert np.isclose(df_features['DIS'].median(), -0.2790473)
         assert np.isclose(df_features['INDUS'].median(), -0.2108898)
```

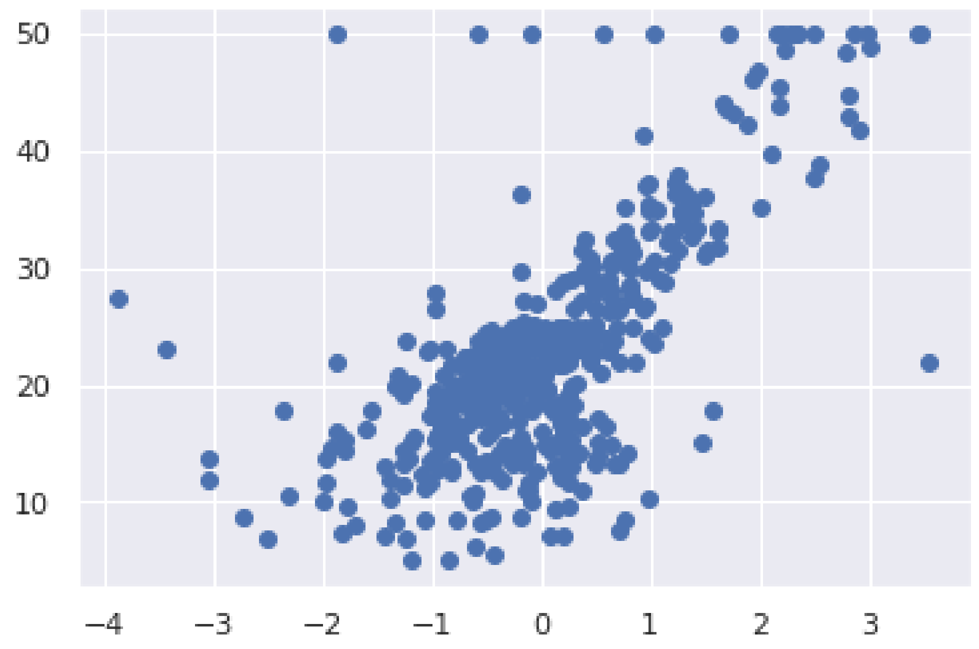
```
                 RM           DIS         INDUS
count  5.060000e+02  5.060000e+02  5.060000e+02
mean  -9.478584e-17 -1.404235e-16  3.089316e-16
std    1.000000e+00  1.000000e+00  1.000000e+00
min   -3.876413e+00 -1.265817e+00 -1.556302e+00
25%   -5.680681e-01 -8.048913e-01 -8.668328e-01
50%   -1.083583e-01 -2.790473e-01 -2.108898e-01
75%    4.822906e-01  6.617161e-01  1.014995e+00
max    3.551530e+00  3.956602e+00  2.420170e+00


             MEDV
count  506.000000
mean    22.532806
std      9.197104
min      5.000000
25%     17.025000
50%     21.200000
75%     25.000000
```

```
max      50.000000
```

```
In [46]: ###
         ### AUTOGRADER TEST - DO NOT REMOVE
         ###
```

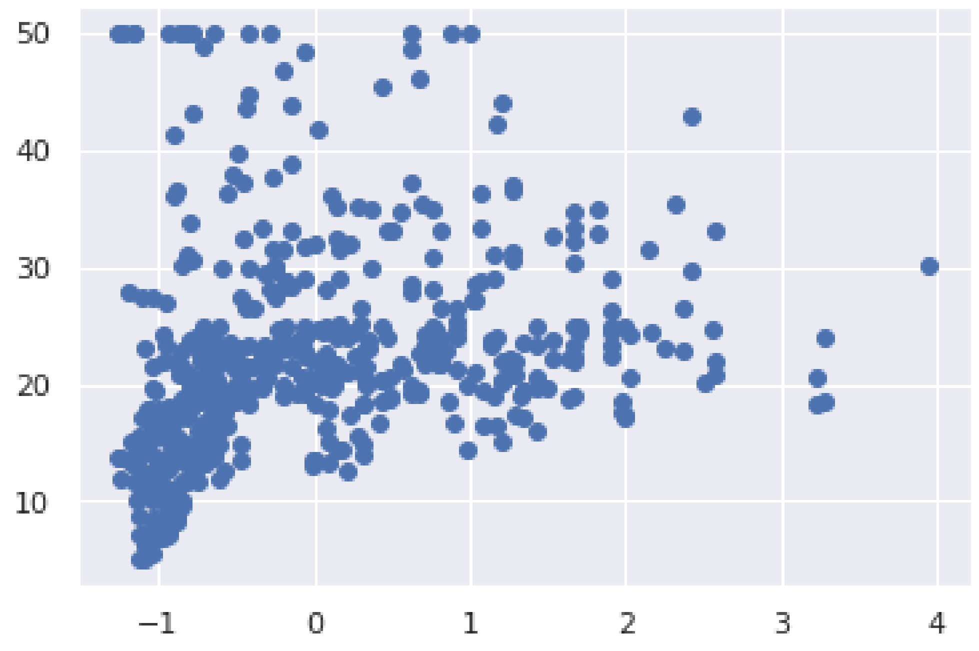
```
In [47]: sns.set()
         plt.scatter(df_features["RM"], df_target)
```

```
Out[47]: <matplotlib.collections.PathCollection at 0x7fb728001610>
```



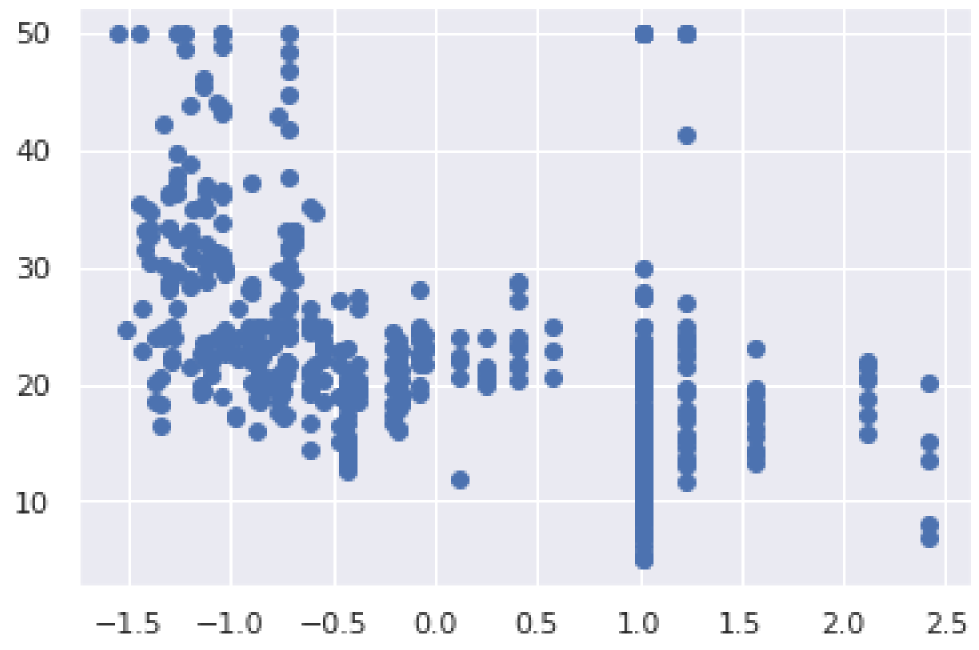
```
In [48]: plt.scatter(df_features["DIS"], df_target)
```

```
Out[48]: <matplotlib.collections.PathCollection at 0x7fb727f7da10>
```



```
In [49]: plt.scatter(df_features["INDUS"], df_target)
```

```
Out[49]: <matplotlib.collections.PathCollection at 0x7fb727ef73d0>
```



HW1. *Multiple variables cost function:* Write a function `compute_cost_multi()` to compute the cost function of a linear regression model. The function should take in two 2-D numpy arrays. The first one is the matrix of the linear equation and the second one is the actual target value.

Recall that:

$$J(\hat{\beta}_0, \hat{\beta}_1) = \frac{1}{2m}\Sigma_{i=1}^{m}\left(\hat{y}(x^i) - y^i\right)^2$$

where

$$\hat{y}(x) = \hat{\beta}_0 + \hat{\beta}_1 x_1 + \hat{\beta}_2 x_2 + \ldots + \hat{\beta}_n x_n$$

The function should receive three Numpy arrays: - `X`: is the feature 2D Numpy array - `y`: is the target 2D Numpy array - `beta`: is the parameter 2D Numpy array

The function should return the cost which is a float.

You can use the following function in your code: - `np.matmul(array1, array2)`

Note that if you wrote your Cohort session's `compute_cost()` using proper Matrix operations to do the square and the summation, the code will be exactly the same here.

```
In [50]: def compute_cost(X, y, beta):
             m = X.shape[0]
             y_pred = np.matmul(X,beta)
             error = y_pred - y
             J = (1/(2*m))*np.matmul(error.T,error)

             return J[0][0]
```

```
In [51]: X = prepare_feature(df_features)
         target = prepare_target(df_target)

         assert isinstance(X, np.ndarray)
         assert isinstance(target, np.ndarray)
         assert X.shape == (506, 4)
         assert target.shape == (506, 1)
```

```
In [52]: ###
         ### AUTOGRADER TEST - DO NOT REMOVE
         ###
```

```
In [53]: beta = np.zeros((4,1))
         J = compute_cost(X, target, beta)
         print(J)
         assert np.isclose(J, 296.0734)

         beta = np.ones((4,1))
         J = compute_cost(X, target, beta)
         print(J)
         assert np.isclose(J, 270.4083)

         beta = np.array([-1, 2, 1, 2]).reshape((4,1))
         J = compute_cost(X, target, beta)
         print(J)
         assert np.isclose(J, 314.8510)
```

```
296.07345849802374
270.40834049507566
314.8509513115608
```

```
In [54]: ###
         ### AUTOGRADER TEST - DO NOT REMOVE
         ###
```

HW2. *Gradient Descent:* Write a function called `gradient_descent_multi()` that takes in four parameters: - `X`: is a 2-D numpy array for the features - `y`: is a vector array for the target - `alpha`: is the learning rate - `num_iters`: is the number of iteration to perform

The function should return two arrays: - `beta`: is coefficient at the end of the iteration - `J_storage`: is the array that stores the cost value at each iteration

You can use some of the following functions: - `np.matmul(array1, array2)`: which is to do matrix multiplication on two Numpy arrays. - `compute_cost()`: which the function you created in the previous problem set to compute the cost.

Note that if you use proper matrix operations in your cohort sessions for the gradient descent function, the code will be the same here.

```
In [55]: def gradient_descent(X, y, beta, alpha, num_iters):
             m = X.shape[0]
             J_storage = np.zeros(num_iters)

             for i in range(num_iters):
                 yp = np.matmul(X,beta)
                 error = yp - y
                 beta = beta - (alpha/m) * np.matmul(X.T,error)
                 cost = compute_cost(X,y,beta)
                 J_storage[i] = cost


             return beta, J_storage
```

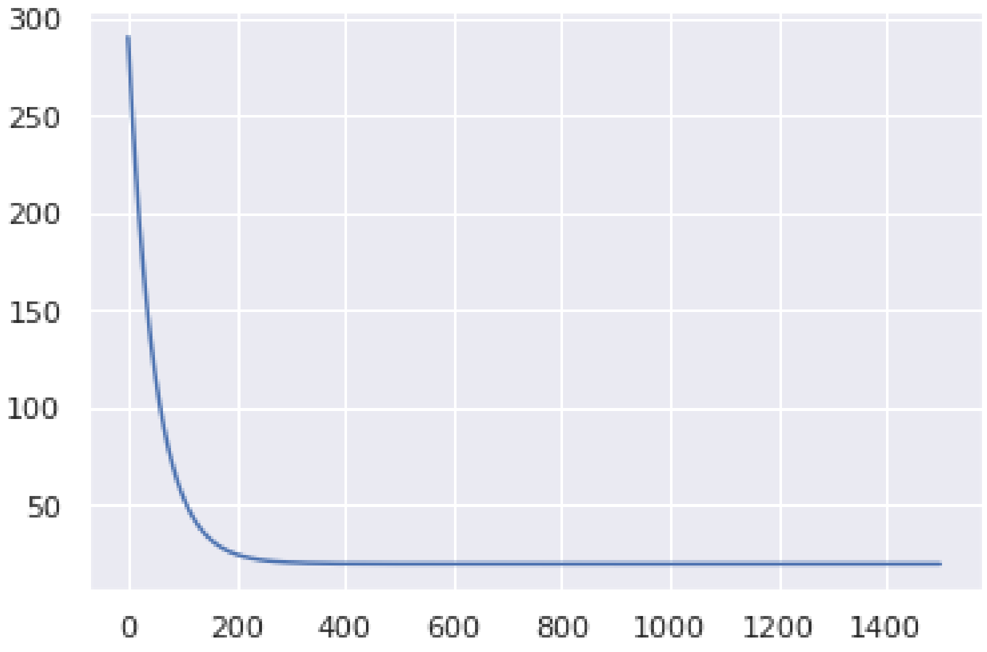
```
In [56]: iterations = 1500
         alpha = 0.01
         beta = np.zeros((4,1))

         beta, J_storage = gradient_descent(X, target, beta, alpha, iterations)
         print(beta)
         assert np.isclose(beta[0], 22.5328)
         assert np.isclose(beta[1], 5.4239)
         assert np.isclose(beta[2], -0.90367)
         assert np.isclose(beta[3], -2.95818)
```

```
[[22.53279993]
 [ 5.42386374]
 [-0.90367449]
 [-2.95818095]]
```

```
In [57]: ###
         ### AUTOGRADER TEST - DO NOT REMOVE
         ###
```

```
In [58]: plt.plot(J_storage)
```

```
Out[58]: [<matplotlib.lines.Line2D at 0x7fb727e83810>]
```



HW3. Do the following tasks: - Get the features and the targets. - features: RM, DIS, INDUS - target: MEDV - Use the previous functions called `predict()` to calculated the predicted values given the features and the model. - Create a target numpy array from the data frame.

```
In [59]: # This is from the previous result
         beta = np.array([[22.53279993],
                 [ 5.42386374],
                 [-0.90367449],
                 [-2.95818095]])

         # Extract the feature and the target
         df_features, df_target = get_features_targets(df,["RM", "DIS", "INDUS"],["MEDV"])
```

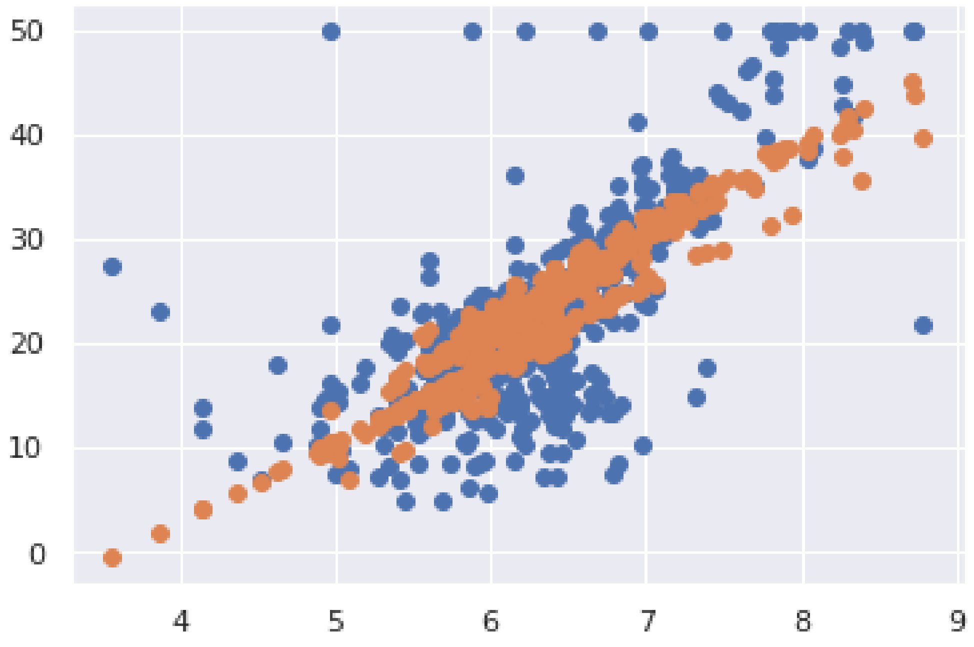
```
        # Call predict()
        pred = predict(df_features,beta)

        # Change target to numpy array
        target = df_target.to_numpy()

        ###
        ### YOUR CODE HERE
        ###
```

```
In [60]: assert isinstance(pred, np.ndarray)
         assert np.isclose(pred.mean(), 22.5328)
         assert np.isclose(pred.std(), 6.7577)
```

```
In [61]: ###
         ### AUTOGRADER TEST - DO NOT REMOVE
         ###
```

```
In [62]: plt.scatter(df_features["RM"],target)
         plt.scatter(df_features["RM"],pred)
         ### END HIDDEN TESTS
```

```
Out[62]: <matplotlib.collections.PathCollection at 0x7fb727e09c10>
```



```
In [63]: ###
         ### AUTOGRADER TEST - DO NOT REMOVE
         ###
```

```
In [64]: ###
         ### AUTOGRADER TEST - DO NOT REMOVE
         ###
```

HW4. *Splitting data:* Do the following tasks: - Extract the following: - features: RM, DIS, and INDUS - target: MEDV - Use Week 9's function `split_data()` to split the data into train and test using `random_state=100` and `test_size=0.3`. - Normalize and prepare the features and the target. - Use the training data set and call `gradient_descent()` to obtain the `theta`. - Use the test data set to get the predicted values.

You need to replace the `None` in the code below with other a function call or any other Python expressions.

```
In [65]: # Extract the features and the target
         df_features, df_target = get_features_targets(df,["RM", "DIS", "INDUS"],["MEDV"])

         # Split the data set into training and test
         df_features_train, df_features_test, df_target_train, df_target_test = split_data(df_

         # Normalize the features using z normalization
         df_features_train_z = normalize_z(df_features_train)


         # Change the features and the target to numpy array using the prepare functions
         X = prepare_feature(df_features_train_z) # concatenating for the y intercept
         target = prepare_target(df_target_train)

         iterations = 1500
         alpha = 0.01
         beta = np.zeros((4,1))

         # call the gradient_descent function
         beta, J_storage = gradient_descent(X, target, beta, alpha, iterations)

         # call the predict() method
         pred = predict(df_features_test,beta)

         print(beta)
```

```
[[22.66816258]
 [ 5.20934787]
 [-0.96089997]
 [-3.22203154]]
```

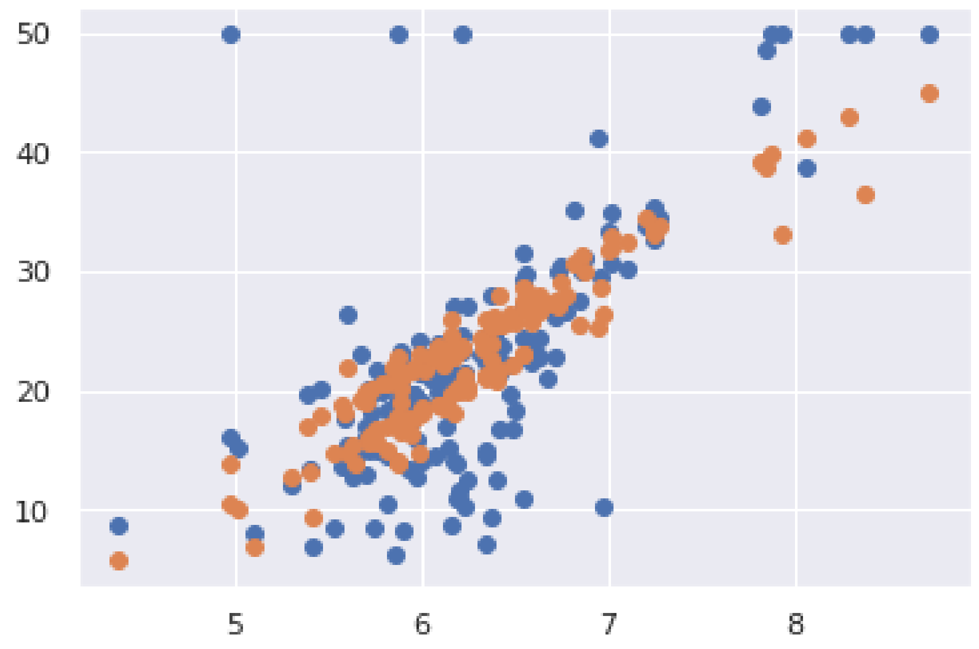
```
In [66]: ###
         ### AUTOGRADER TEST - DO NOT REMOVE
         ###

In [67]: assert isinstance(pred, np.ndarray)
         assert pred.shape == (151, 1)
```

```
        assert np.isclose(pred.mean(), 22.66816)
        assert np.isclose(pred.std(), 6.67324)
```

```
In [68]: ###
         ### AUTOGRADER TEST - DO NOT REMOVE
         ###
```

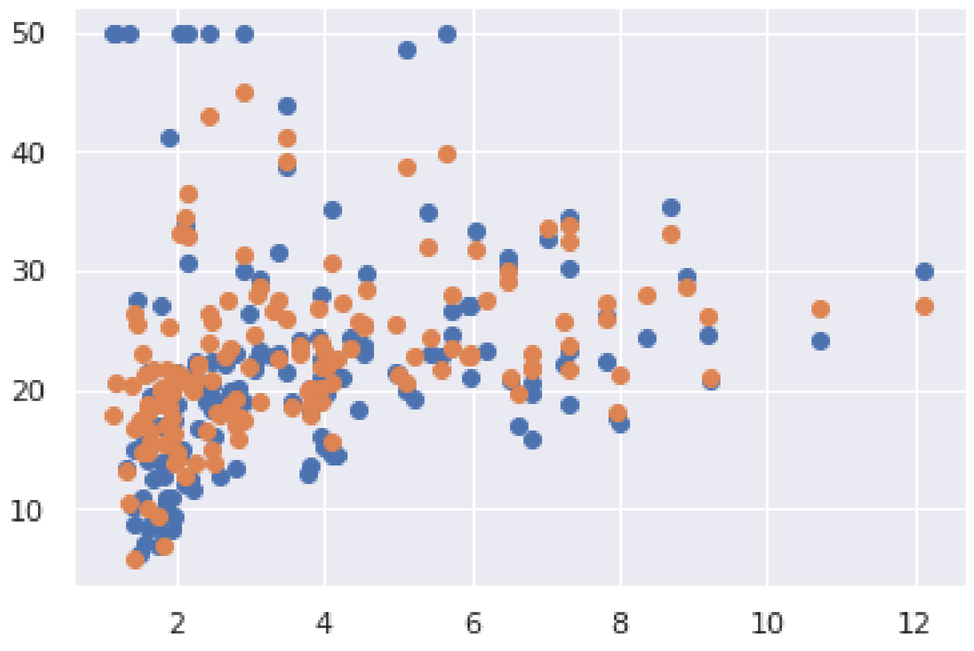
```
In [69]: plt.scatter(df_features_test["RM"], df_target_test)
         plt.scatter(df_features_test["RM"], pred)
```

```
Out[69]: <matplotlib.collections.PathCollection at 0x7fb727e962d0>
```

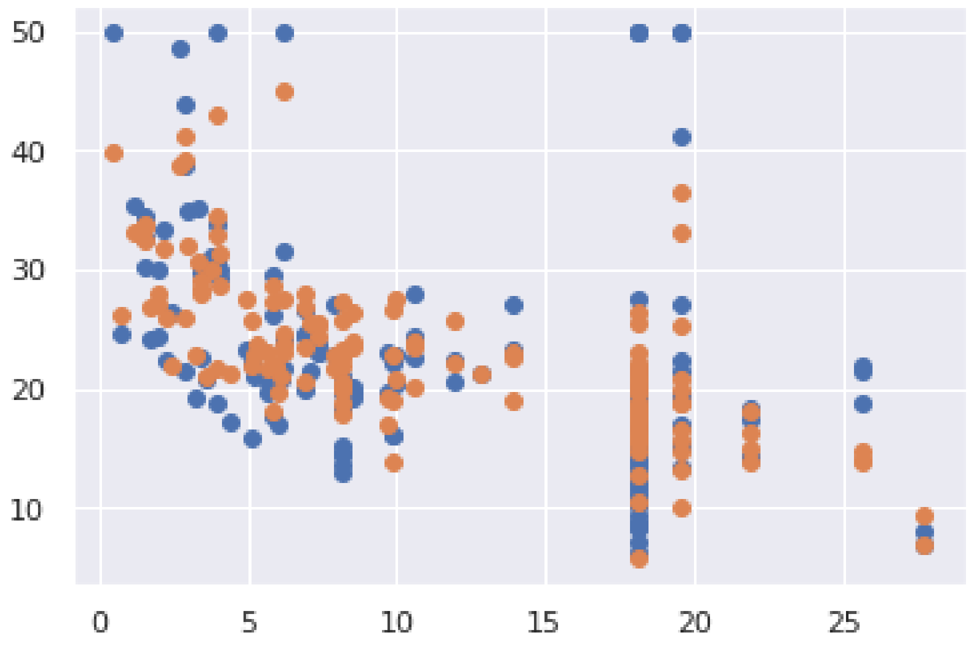


```
In [70]: plt.scatter(df_features_test["DIS"], df_target_test)
         plt.scatter(df_features_test["DIS"], pred)
```

```
Out[70]: <matplotlib.collections.PathCollection at 0x7fb727cefe10>
```



```
In [71]: plt.scatter(df_features_test["INDUS"], df_target_test)
         plt.scatter(df_features_test["INDUS"], pred)

Out[71]: <matplotlib.collections.PathCollection at 0x7fb727c6ab50>
```



```
In [74]: target.shape

Out[74]: (506, 1)
```

HW5. Calculate the coefficient of determination, r^2.

```
In [75]: # change target test set to a numpy array
         target = df_target.to_numpy()

         # Calculate r2 score by calling a function
         r2 = r2_score(df_target_test, pred)


         print(r2)

MEDV    0.477135
dtype: float64


In [76]: assert np.isclose(r2, 0.47713)

In [77]: ###
         ### AUTOGRADER TEST - DO NOT REMOVE
         ###
```

HW6. Calculate the mean squared error.

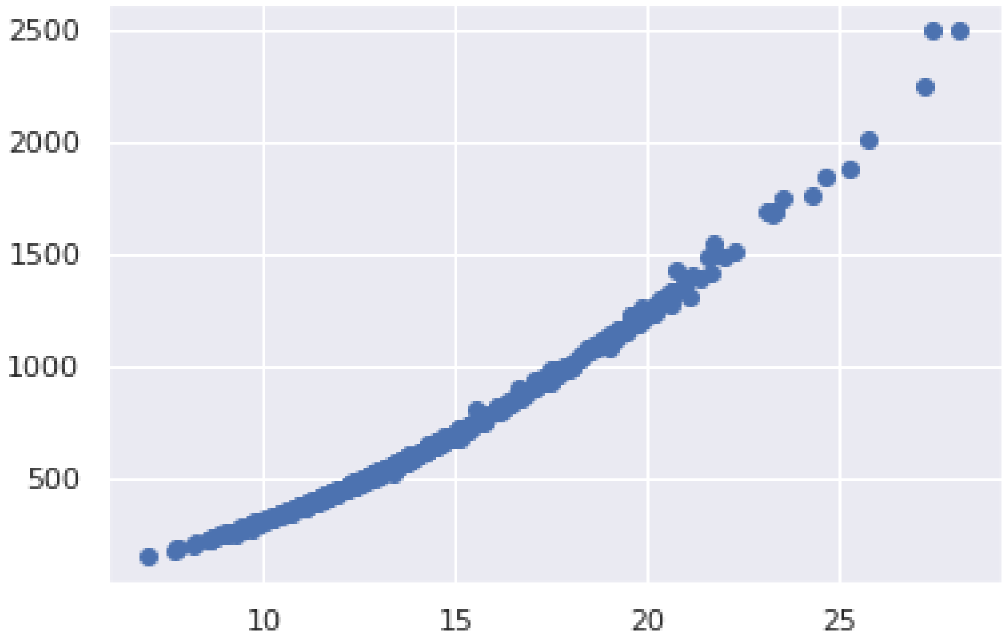
```
In [79]: # Calculate the mse
         mse = mean_squared_error(df_target_test, pred)

         ###
         ### YOUR CODE HERE
         ###
         print(mse)

             0
MEDV  51.362988


/usr/lib/python3.7/site-packages/ipykernel_launcher.py:67: FutureWarning: Calling a ufunc on no
Convert one of the arguments to a NumPy array (eg 'ufunc(df1, np.asarray(df2)') to keep the cu


In [80]: assert np.isclose(mse, 51.363)

In [34]: ###
         ### AUTOGRADER TEST - DO NOT REMOVE
         ###
```

HW7. *Polynomial Transformation:* Redo the steps for breast cancer data but this time we will use quadratic model. Use the following columns: - x data: radius_mean - y data: area_mean

We will create a quadratic hypothesis for this x and y data. To do that write a function `transform_features(df, colname, colname_transformed)` that takes in a dataframe for the features, the original column name, and the transformed column name. The function should add another column with the name `colname_transformed` with the value of column in `colname` transformed to its quadratic value.

```
In [84]: # Read from breast_cancer_data.csv file
         df = pd.read_csv("breast_cancer_data.csv")

         # Extract feature and target
         df_feature, df_target = get_features_targets(df,["radius_mean"],["area_mean"])

         plt.scatter(df_feature, df_target)
```

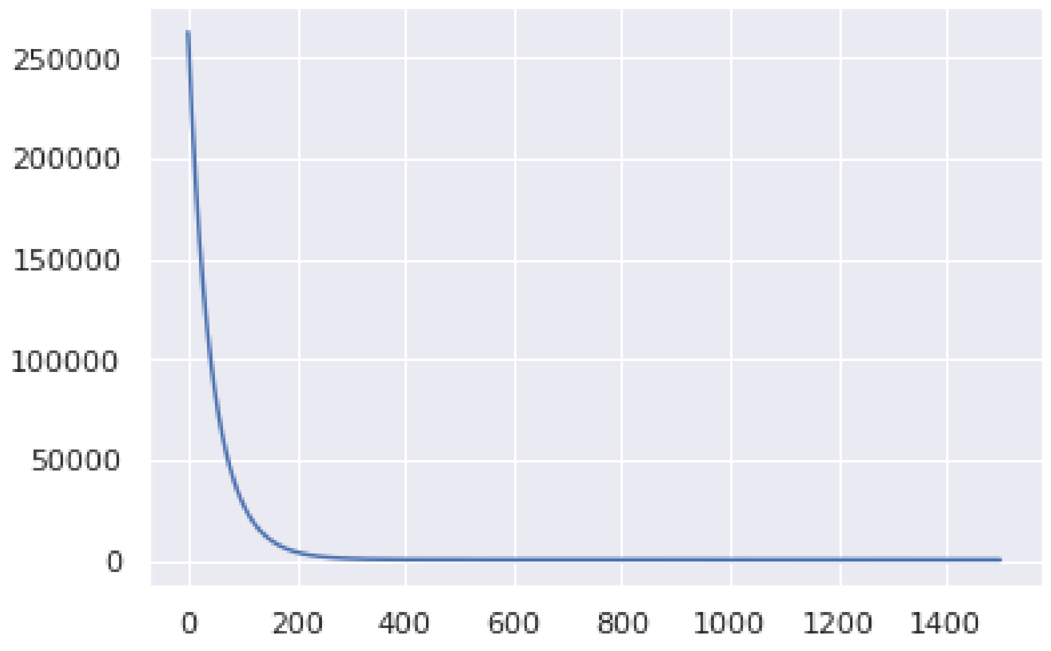
```
Out[84]: <matplotlib.collections.PathCollection at 0x7fb727af3fd0>
```



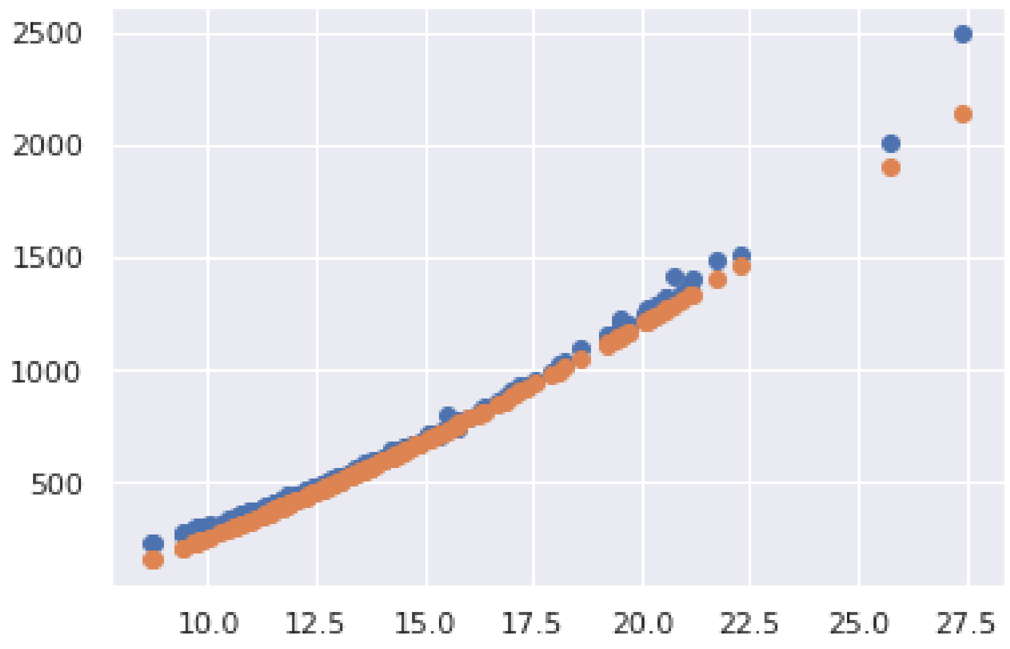
```
In [36]: ###
         ### AUTOGRADER TEST - DO NOT REMOVE
         ###
```

```
In [85]: # write your function to create a quadratic feature of x

         def transform_features(df_feature, colname, colname_transformed):
             col = df_feature[colname]
             df_feature[colname_transformed] = np.square(col)
             return df_feature
```

```
In [86]: df_features = transform_features(df_feature, "radius_mean", "radius_mean^2")

         assert (df_features.loc[:,"radius_mean^2"] == df_features.loc[:,"radius_mean"] ** 2).a

In [39]: ###
         ### AUTOGRADER TEST - DO NOT REMOVE
         ###

In [88]: # split data using random_state = 100 and 30% test size
         df_features_train, df_features_test, df_target_train, df_target_test = split_data(df_

         # normalize features
         df_feature_train_z = normalize_z(df_features_train)

         # change to numpy array and append column for feature
         X = prepare_feature(df_feature_train_z) # concatenating for the y intercept
         target = prepare_target(df_target_train)


         iterations = 1500
         alpha = 0.01
         beta = np.zeros((3,1))

         # call gradient_descent() function
         beta, J_storage = gradient_descent(X, target, beta, alpha, iterations)

In [ ]:

In [89]: ###
         ### AUTOGRADER TEST - DO NOT REMOVE
         ###

In [90]: assert np.isclose(beta[0], 646.0787)
         assert np.isclose(beta[1], 146.4801)
         assert np.isclose(beta[2], 201.9803)

In [91]: ###
         ### AUTOGRADER TEST - DO NOT REMOVE
         ###

In [92]: plt.plot(J_storage)

Out[92]: [<matplotlib.lines.Line2D at 0x7fb727abbe90>]
```



```
In [97]: type(df_target_test)

Out[97]: pandas.core.frame.DataFrame

In [99]: # change target to numpy array
         beta = np.array([[646.0787641 ], [146.4800792 ], [201.98031254]])

         target = df_target_test
         pred = predict(df_features_test,beta)

         # get predicted values
         ###
         ### YOUR CODE HERE
         ###

In [100]: ###
          ### AUTOGRADER TEST - DO NOT REMOVE
          ###

In [101]: plt.scatter(df_features_test["radius_mean"], target)
          plt.scatter(df_features_test["radius_mean"], pred)

Out[101]: <matplotlib.collections.PathCollection at 0x7fb7279e3ed0>
```



```
In [48]: r2 = r2_score(df_target_test, pred)
         print(r2)
         assert np.isclose(r2, 0.985095)

In [49]: ###
         ### AUTOGRADER TEST - DO NOT REMOVE
         ###

In [50]: mse = mean_squared_error(df_target_test, pred)
         print(mse)
         assert np.isclose(mse, 1919.164)

In [51]: ###
         ### AUTOGRADER TEST - DO NOT REMOVE
         ###
```

HW8. *Optional:* Redo the above tasks using Sci-kit learn libraries. You will need to use the following: - LinearRegression - train_test_split - r2_score - mean_squared_error - PolynomialFeatures

```
In [52]: from sklearn.linear_model import LinearRegression
         from sklearn.model_selection import train_test_split
         from sklearn.metrics import r2_score, mean_squared_error
         from sklearn.preprocessing import PolynomialFeatures
```

Redo HW 4 using Scikit Learn

```
In [53]: # Read the housing_processed.csv file
         df = None

         # extract the features from ["RM", "DIS", "INDUS"] and target from []"MEDV"]
         df_features, df_target = None, None
         # normalize
         df_features = None

         ###
         ### YOUR CODE HERE
         ###

In [54]: # Split the data into training and test data set using scikit-learn function
         df_features_train, df_features_test, df_target_train, df_target_test = None, None, No

         # Instantiate LinearRegression() object
         model = None

         # Call the fit() method
         pass

         ###
         ### YOUR CODE HERE
         ###
         print(model.coef_, model.intercept_)
         assert np.isclose(model.coef_, [ 5.01417104, -1.00878266, -3.27301726]).all()
         assert np.isclose(model.intercept_, 22.45962454)

In [55]: # Call the predict() method
         pred = None

         ###
         ### YOUR CODE HERE
         ###

In [56]: plt.scatter(df_features_test["RM"], df_target_test)
         plt.scatter(df_features_test["RM"], pred)

In [57]: plt.scatter(df_features_test["DIS"], df_target_test)
         plt.scatter(df_features_test["DIS"], pred)

In [58]: plt.scatter(df_features_test["INDUS"], df_target_test)
         plt.scatter(df_features_test["INDUS"], pred)

In [59]: r2 = r2_score(df_target_test, pred)
         print(r2)
         assert np.isclose(r2, 0.48250)

In [60]: mse = mean_squared_error(df_target_test, pred)
         print(mse)
         assert np.isclose(mse, 52.41451)
```

Redo HW7 Using Scikit Learn

```
In [61]: # Read the file breast_cancer_data.csv
         df = None
         # extract feature from "radius_mean" and target from "area_mean"
         df_feature, df_target = None, None

         ###
         ### YOUR CODE HERE
         ###
```

```
In [62]: # instantiate a PolynomialFeatures object with degree = 2
         poly = None

         # call its fit_transform() method
         df_features = None

         # call train_test_split() to split the data
         df_features_train, df_features_test, df_target_train, df_target_test = None, None, Nor

         # instantiate LinearRegression() object
         model = None

         # call its fit() method
         pass


         ### BEGIN SOLUTON
         poly = PolynomialFeatures(2)
         df_features = poly.fit_transform(df_feature)
         df_features_train, df_features_test, df_target_train, df_target_test = train_test_spli
         model = LinearRegression()
         model.fit(df_features_train, df_target_train)
         ### END SOLUTION
         print(model.coef_, model.intercept_)
         assert np.isclose(model.coef_, [0., 3.69735512, 2.9925278 ]).all()
         assert np.isclose(model.intercept_, -32.3684598)
```

```
In [63]: # Call the predict() method
         pred = None

         ###
         ### YOUR CODE HERE
         ###

         print(type(pred), pred.mean(), pred.std())
         assert isinstance(pred, np.ndarray)
         assert np.isclose(pred.mean(), 672.508465)
         assert np.isclose(pred.std(), 351.50271)
```

```
In [64]: plt.scatter(df_features_test[:,1], df_target_test)
         plt.scatter(df_features_test[:,1], pred)
```

```
In [65]: r2 = r2_score(df_target_test, pred)
         print(r2)
         assert np.isclose(r2, 0.99729)
```

```
In [66]: mse = mean_squared_error(df_target_test, pred)
         print(mse)
         assert np.isclose(mse, 346.79479)
```

Week10_Cohort

December 10, 2021

1 Week 10 Problem Set

1.1 Cohort Session

```
In [1]: import pandas as pd
        import numpy as np
        import matplotlib.pyplot as plt
```

CS0. Do the following tasks before you start with the first cohort session.
Task 1. Paste the following functions from your previous work: - `get_features_targets()` - `normalize_z()` - `prepare_feature()` - `prepare_target()` - `split_data()`

```
In [2]: def normalize_z(df):
            dfout = (df - df.mean(axis=0)) / df.std(axis=0)
            return dfout

        def get_features_targets(df, feature_names, target_names):
            df_feature = df[feature_names]
            df_target = df[target_names]
            return df_feature, df_target

        def prepare_feature(df_feature): # creates X matrix
            feature = df_feature.to_numpy()
            array1 = np.ones((feature.shape[0], 1))
        #     array1 = np.ones(feature.shape) --> works too
            X = np.concatenate((array1, feature), axis=1)
            return X

        def prepare_target(df_target): # creates numpy array for y (target)
            return df_target.to_numpy()

        def split_data(df_feature, df_target, random_state=None, test_size=0.5):
            # assuming that index is consistent between features and target
            indices = df_target.index
            if random_state != None:
                np.random.seed(random_state)

            # k is the no. of rows in the test set
```

```
    num_rows = len(indices)
    k = int(test_size * num_rows)
    # randomly choose indices for test set
    test_indices = np.random.choice(indices, k, replace=False)

    indices = set(indices)
    test_indices = set(test_indices)
    train_indices = indices - test_indices

    df_feature_train = df_feature.loc[train_indices, :]
    df_feature_test = df_feature.loc[test_indices, :]
    df_target_train = df_target.loc[train_indices, :]
    df_target_test = df_target.loc[test_indices, :]

    return df_feature_train, df_feature_test, df_target_train, df_target_test
```

Task 2. Load the breast cancer data from `breast_cancer_data.csv` into a Data Frame.

```
In [3]: # read breast_cancer_data.csv
        df = pd.read_csv("breast_cancer_data.csv")

        df
```

```
Out[3]:            id diagnosis  radius_mean  texture_mean  perimeter_mean  area_mean  \
        0      842302         M        17.99         10.38          122.80     1001.0
        1      842517         M        20.57         17.77          132.90     1326.0
        2    84300903         M        19.69         21.25          130.00     1203.0
        3    84348301         M        11.42         20.38           77.58      386.1
        4    84358402         M        20.29         14.34          135.10     1297.0
        ..        ...       ...          ...           ...             ...        ...
        564    926424         M        21.56         22.39          142.00     1479.0
        565    926682         M        20.13         28.25          131.20     1261.0
        566    926954         M        16.60         28.08          108.30      858.1
        567    927241         M        20.60         29.33          140.10     1265.0
        568     92751         B         7.76         24.54           47.92      181.0

             smoothness_mean  compactness_mean  concavity_mean  concave points_mean  \
        0            0.11840           0.27760         0.30010              0.14710
        1            0.08474           0.07864         0.08690              0.07017
        2            0.10960           0.15990         0.19740              0.12790
        3            0.14250           0.28390         0.24140              0.10520
        4            0.10030           0.13280         0.19800              0.10430
        ..               ...               ...             ...                  ...
        564          0.11100           0.11590         0.24390              0.13890
        565          0.09780           0.10340         0.14400              0.09791
        566          0.08455           0.10230         0.09251              0.05302
        567          0.11780           0.27700         0.35140              0.15200
```

```
568         0.05263          0.04362        0.00000             0.00000

     ...  radius_worst  texture_worst  perimeter_worst  area_worst  \
0    ...        25.380          17.33           184.60      2019.0
1    ...        24.990          23.41           158.80      1956.0
2    ...        23.570          25.53           152.50      1709.0
3    ...        14.910          26.50            98.87       567.7
4    ...        22.540          16.67           152.20      1575.0
..   ...           ...            ...              ...         ...
564  ...        25.450          26.40           166.10      2027.0
565  ...        23.690          38.25           155.00      1731.0
566  ...        18.980          34.12           126.70      1124.0
567  ...        25.740          39.42           184.60      1821.0
568  ...         9.456          30.37            59.16       268.6

     smoothness_worst  compactness_worst  concavity_worst  \
0             0.16220            0.66560           0.7119
1             0.12380            0.18660           0.2416
2             0.14440            0.42450           0.4504
3             0.20980            0.86630           0.6869
4             0.13740            0.20500           0.4000
..                ...                ...              ...
564           0.14100            0.21130           0.4107
565           0.11660            0.19220           0.3215
566           0.11390            0.30940           0.3403
567           0.16500            0.86810           0.9387
568           0.08996            0.06444           0.0000

     concave points_worst  symmetry_worst  fractal_dimension_worst
0                  0.2654          0.4601                  0.11890
1                  0.1860          0.2750                  0.08902
2                  0.2430          0.3613                  0.08758
3                  0.2575          0.6638                  0.17300
4                  0.1625          0.2364                  0.07678
..                    ...             ...                      ...
564                0.2216          0.2060                  0.07115
565                0.1628          0.2572                  0.06637
566                0.1418          0.2218                  0.07820
567                0.2650          0.4087                  0.12400
568                0.0000          0.2871                  0.07039

[569 rows x 32 columns]
```

Task 3. Do the following tasks.

- Read the following columns
  - feature: `radius_mean`
  - target: `diagnosis`

- Normalize the feature column using z normalization.

```
In [4]: # extract the feature and the target
        df_feature, df_target = get_features_targets(df, ["radius_mean"], ["diagnosis"])

        # normalize the feature
        df_feature = normalize_z(df_feature)
```

Task 4. Write a function `replace_target()` to replace the `diagnosis` column with the following mapping: - `M`: `1`, this means that malignant cell are indicated as `1` in our new column. - `B`: `0`, this means that benign cell are indicated as `0` in our new column.

The function should takes in the following:

- `df_target`: the target data frame
- `target_name`: which is the column name of the target data frame
- `map`: which is a dictionary containing the map

It should returns a new data frame with the same column name but with its values changed according to the mapping.

```
In [5]: def replace_target(df_target, target_name, map_vals):
            df_out = df_target.copy()
            df_out.loc[:, target_name] = df_target[target_name].apply(lambda val: map_vals[val]

            return df_out
```

```
In [6]: df_target = replace_target(df_target, "diagnosis", {'M': 1, 'B': 0})
        df_target
```

```
Out[6]:      diagnosis
        0            1
        1            1
        2            1
        3            1
        4            1
        ..         ...
        564          1
        565          1
        566          1
        567          1
        568          0

        [569 rows x 1 columns]
```

Task 5. Do the following tasks. - Change feature to Numpy array and append constant 1 column. - Change target to Numpy array

```
In [7]: # change feature data frame to numpy array and append column 1
        feature = prepare_feature(df_feature)

        # change target data frame to numpy array
        target = prepare_target(df_target)
```

CS1. *Logistic function:* Write a function to calculate the hypothesis using a logistic function. Recall that the hypothesis for a logistic regression model is written as:

$$\mathbf{p}(x) = \frac{1}{1 + e^{-\mathbf{X}\mathbf{b}}}$$

The shape of the input is as follows: - **b**: is a column vector for the parameters - **X**: is a matrix where the number of rows are the number of data points and the the number of columns must the same as the number of parameters in **b**.

Note that you need to ensure that the output is a **column vector**.

You can use the following functions: - `np.matmul(array1, array2)`: which is to perform matrix multiplication on the two numpy arrays. - `np.exp()`: which is to calculate the function e^x

```
In [8]: # logistic function transforms x to z (p against x -> z against x (linear) -> p agains
        # z is a linear combination that transforms from x to z, p is the hypothesis

        def log_regression(beta, X):
            z = np.matmul(X, beta)
            return 1 / (1 + np.exp(-z))
```

```
In [9]: beta = np.array([0])
        x = np.array([0])
        ans = log_regression(beta, x)
        assert ans == 0.5

        beta = np.array([2])
        x = np.array([40])
        ans = log_regression(beta, x)
        assert np.isclose(ans, 1.0)

        beta = np.array([2])
        x = np.array([-40])
        ans = log_regression(beta, x)
        assert np.isclose(ans, 0.0)

        beta = np.array([[1, 2, 3]])
        x = np.array([[3, 2, 1]])
        ans = log_regression(beta.T, x)
        assert np.isclose(ans.all(), 1.0)

        beta = np.array([[1, 2, 3]])
        x = np.array([[3, 2, 1], [3, 2, 1]])
        ans = log_regression(beta.T, x)
        assert ans.shape == (2, 1)
        assert np.isclose(ans.all(), 1.0)
```

```
In [10]: ###
         ### AUTOGRADER TEST - DO NOT REMOVE
         ###
```

CS2. *Cost Function:* Write a function to calculate the cost function for logistic regression. Recall that the cost function for logistic regression is given by:

$$J(\beta) = -\frac{1}{m}\left[\Sigma_{i=1}^{m} y^i \log(p(x^i)) + (1-y^i)\log(1-p(x^i))\right]$$

You can use the following function in your code: - `np.where(condition, then_expression, else_expression)`

```
In [11]: # cost function: small penalty when y is close to 0

         def compute_cost_logreg(beta, X, y):
             np.seterr(divide = 'ignore')
             p = log_regression(beta, X)
             m = X.shape[0]
             J = (-1 / m) * np.sum(np.where(y == 1, np.log(p), np.log(1 - p)))
             np.seterr(divide = 'warn')
             return J
```

```
In [12]: y = np.array([[1]])
         X = np.array([[10, 40]])
         beta = np.array([1, 1]).T
         ans = compute_cost_logreg(beta, X, y)
         print(ans)
         assert np.isclose(ans, 0)

         y = np.array([[0]])
         X = np.array([[10, 40]])
         beta = np.array([[-1, -1]]).T
         ans = compute_cost_logreg(beta, X, y)
         print(ans)
         assert np.isclose(ans, 0)
```

```
-0.0
-0.0
```

```
In [13]: ###
         ### AUTOGRADER TEST - DO NOT REMOVE
         ###
```

CS3. *Gradient Descent:* Recall that the update functions can be written as a matrix multiplication.

$$\mathbf{b} = \mathbf{b} - \alpha\frac{1}{m}\mathbf{X}^T(\mathbf{p}-\mathbf{y})$$

Write a function called `gradient_descent_logreg()` that takes in five parameters: - `X`: is a 2-D numpy array for the features - `y`: is a vector array for the target - `beta`: is a column vector for the initial guess of the parameters - `alpha`: is the learning rate - `num_iters`: is the number of iteration to perform

The function should return two arrays: - `beta`: is coefficient at the end of the iteration - `J_storage`: is the array that stores the cost value at each iteration

The solution is similar to Linear Regression gradient descent function with two differences: - you need to use `log_regression()` to calculate the hypothesis - you need to use `compute_cost_logreg()` to calculate the cost

```
In [14]: def gradient_descent_logreg(X, y, beta, alpha, num_iters):
             m = X.shape[0]
             J_storage = np.zeros(num_iters)
             for n in range(num_iters):
                 p = log_regression(beta, X)
                 error = p - y
                 delta = np.matmul(X.T, error)
                 beta = beta - (alpha / m) * delta
                 # change the J value from 0 to sum cost (can also use an empty list and appen
                 J_storage[n] = compute_cost_logreg(beta, X, y)
             return beta, J_storage
```

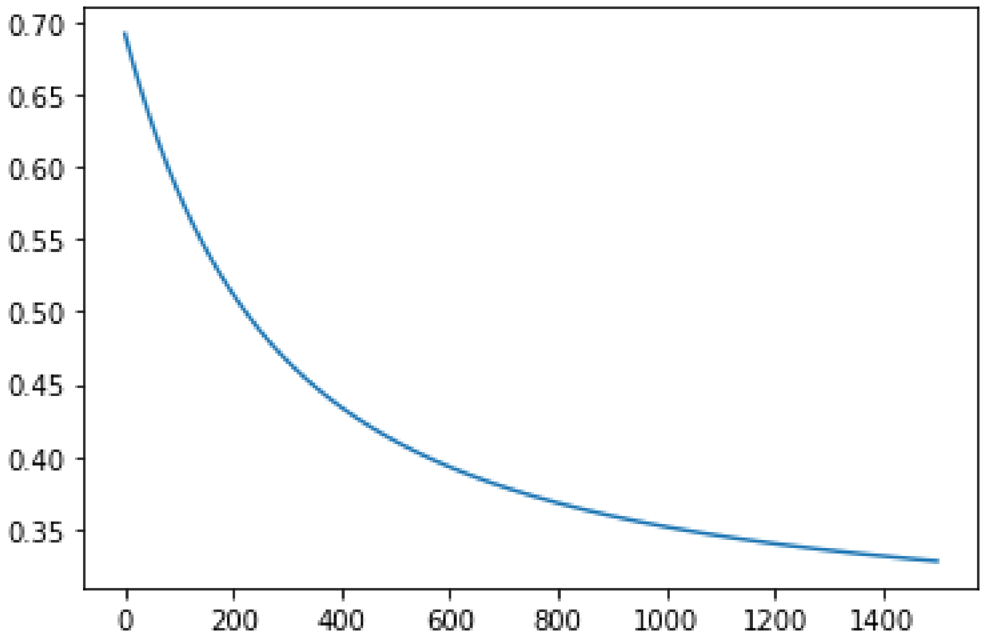
```
In [15]: def gradient_descent_lin_reg(X, y, beta, alpha, num_iters):
             m = X.shape[0]
             J_storage = np.zeros(num_iters)

             for i in range(num_iters):
                 yp = np.matmul(X,beta)
                 error = yp - y
                 beta = beta - (alpha/m) * np.matmul(X.T,error)
                 cost = compute_cost(X,y,beta)
                 J_storage[i] = cost


             return beta, J_storage
```

```
In [16]: iterations = 1500
         alpha = 0.01
         beta = np.zeros((2,1))
         beta, J_storage = gradient_descent_logreg(feature, target, beta, alpha, iterations)

         print(beta)
         assert beta.shape == (2, 1)
         assert np.isclose(beta[0][0], -0.56630)
         assert np.isclose(beta[1][0], 1.93764)
```

```
[[-0.56630289]
 [ 1.93763591]]
```

```
In [17]: ###
         ### AUTOGRADER TEST - DO NOT REMOVE
         ###
```

```
In [18]: plt.plot(J_storage)

Out[18]: [<matplotlib.lines.Line2D at 0x7fc9a6cafcd0>]
```



CS4. *Predict:* Write two functions `predict()` and `predict_norm()` that calculate the straight line equation given the features and its coefficient. - `predict()`: this function should standardize the feature using z normalization, change it to a Numpy array, and add a column of constant 1s. You should use `prepare_feature()` for this purpose. Lastly, this function should also call `predict_norm()` to get the predicted y values. - `predict_norm()`: this function should calculate the straight line equation after standardization and adding of column for constant 1.

You can use the following function in your code: - `np.where()`

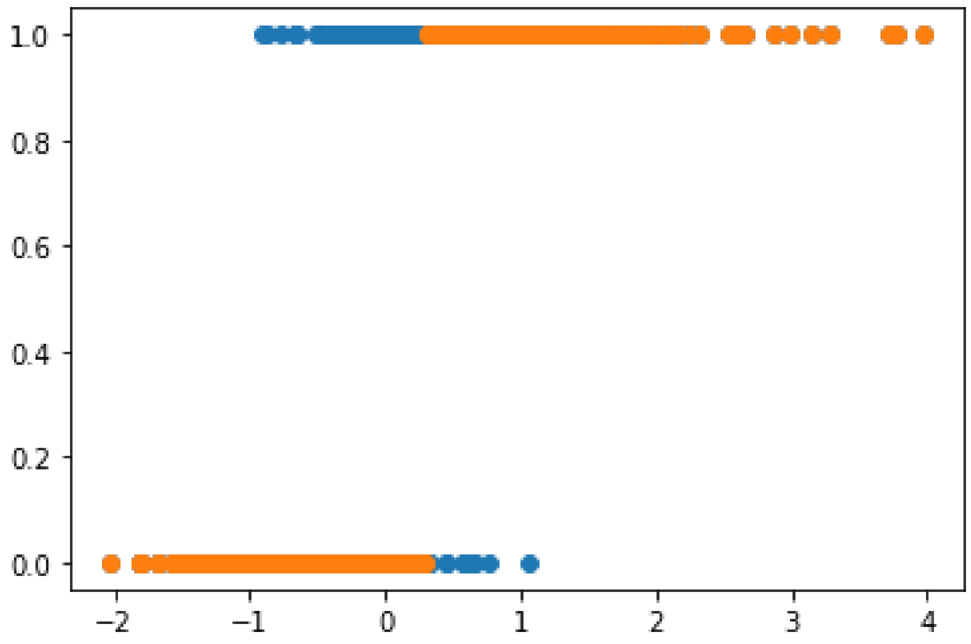
```
In [19]: def predict_norm(X, beta):
             p = log_regression(beta, X)
             return np.where(p >= 0.5, 1, 0)

In [20]: def predict(df_feature, beta):
             feature_z = normalize_z(df_feature)
             X = prepare_feature(feature_z)
             return predict_norm(X, beta)

In [21]: pred = predict(df_feature, beta)
         print(pred.mean(), pred.std())
         assert isinstance(pred, np.ndarray)
         assert np.isclose(pred.mean(), 0.28998)
         assert np.isclose(pred.std(), 0.45375)
```

```
0.28998242530755713 0.4537539182423709
```

```
In [22]: ###
         ### AUTOGRADER TEST - DO NOT REMOVE
         ###
```

```
In [23]: plt.scatter(df_feature, df_target)
         plt.scatter(df_feature, pred)
```

```
Out[23]: <matplotlib.collections.PathCollection at 0x7fc9a6b5fd90>
```



CS5. *Multiple features and splitting of data set:*

Do the following task in the code below: - Read the following column names as the features: `"radius_mean"`, `"texture_mean"`, `"perimeter_mean"`, `"area_mean"`, `"smoothness_mean"`, `"compactness_mean"`, `"concavity_mean"` - Read the column `diagnosis` as the target. Change the value from `M` and `B` to `1` and `0` respectively. - Split the data set with 30% test size and `random_state = 100`. - Normalize the training feature data set using `normalize_z()` function. - Convert to numpy array both the target and the features using `prepare_feature()` and `prepare_target()` functions. - Call `gradient_descent()` function to get the parameters using the training data set. - Call `predict()` function on the test data set to get the predicted values.

```
In [24]: columns = ["radius_mean", "texture_mean", "perimeter_mean", "area_mean", "smoothness_

         # extract the features and the target columns
         df_features, df_target = get_features_targets(df, columns, ["diagnosis"])
```

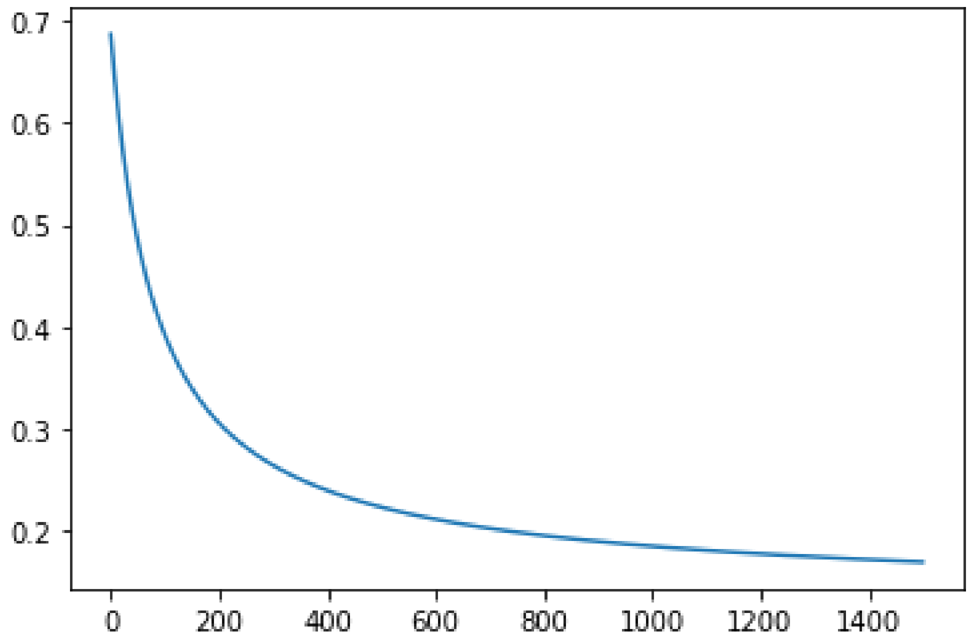
```
        # replace the target values using from string to integer 0 and 1
        df_target = replace_target(df_target, "diagnosis", {'M': 1, 'B': 0})

        # split the data with random_state = 100 and 30% test size
        df_features_train, df_features_test, df_target_train, df_target_test = split_data(df_

        # normalize the features
        df_features_train_z = normalize_z(df_features_train)

        # change the feature columns to numpy array and append column of 1s
        features = prepare_feature(df_features_train_z)

        # change the target column to numpy array
        target = prepare_target(df_target_train)

        iterations = 1500
        alpha = 0.01

        # provide initial guess for theta
        beta = np.zeros((features.shape[1],1))

        # call the gradient descent method
        beta, J_storage = gradient_descent_logreg(features, target, beta, alpha, iterations)

        print(beta)
```

```
[[-0.6139379 ]
 [ 0.82529554]
 [ 0.72746485]
 [ 0.8236603 ]
 [ 0.81647937]
 [ 0.5057749 ]
 [ 0.44176466]
 [ 0.78736842]]
```

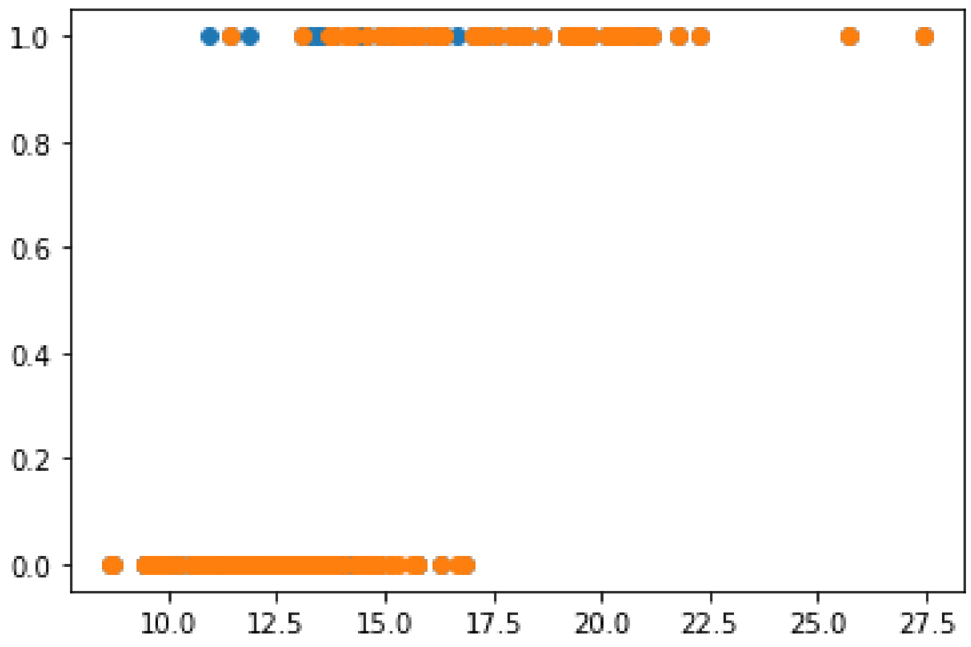
```
In [25]: assert beta.shape == (8, 1)
         ans = np.array([[-0.6139379 ],
                         [ 0.82529554],
                         [ 0.72746485],
                         [ 0.8236603 ],
                         [ 0.81647937],
                         [ 0.5057749 ],
                         [ 0.44176466],
                         [ 0.78736842]])
         assert np.isclose(beta, ans).all()

In [26]: ###
```

```
### AUTOGRADER TEST - DO NOT REMOVE
###
```

```
In [27]: plt.plot(J_storage)

Out[27]: [<matplotlib.lines.Line2D at 0x7fc9a6a87410>]
```

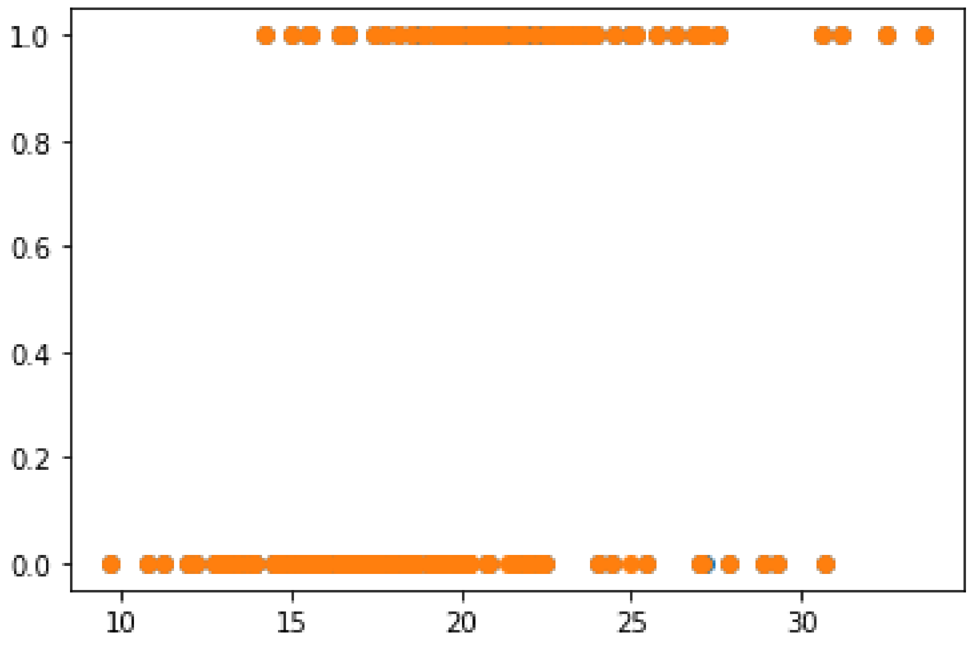


```
In [28]: # call predict() to get the predicted values
         pred = predict(df_features_test, beta)

In [29]: plt.scatter(df_features_test["radius_mean"], df_target_test)
         plt.scatter(df_features_test["radius_mean"], pred)

Out[29]: <matplotlib.collections.PathCollection at 0x7fc9a6a77d90>
```

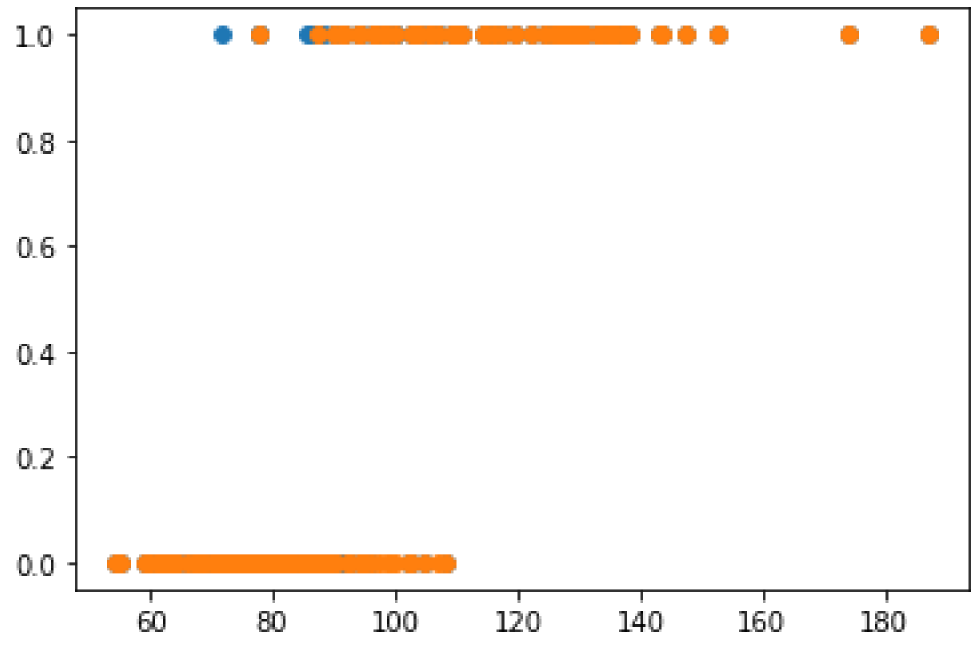


```
In [30]: plt.scatter(df_features_test["texture_mean"], df_target_test)
         plt.scatter(df_features_test["texture_mean"], pred)
```

```
Out[30]: <matplotlib.collections.PathCollection at 0x7fc9a6c7f6d0>
```



```
In [31]: plt.scatter(df_features_test["perimeter_mean"], df_target_test)
         plt.scatter(df_features_test["perimeter_mean"], pred)

Out[31]: <matplotlib.collections.PathCollection at 0x7fc9a6953ed0>
```



CS6. *Confusion Matrix:* Write a function `confusion_matrix()` that takes in: - `ytrue`: which is the true target values - `ypred`: which is the predicted target values - `labels`: which is a list of the category. In the above case it will be `[1, 0]`. Put the positive case as the first element of the list.

The function should return a dictionary containing the matrix with the following format.

	predicted positive (1)	predicted negative (0)
actual positive (1)	correct positive (1, 1)	false negative (1, 0)
actual negative (0)	false positive (0, 1)	correct negative (0, 0)

The keys to the dictionary are the indices: `(0, 0), (0, 1), (1, 0), (1, 1)`.

You can use the following function in your code: - `itertools.product()`: this is to create a combination of all the labels.

1.1.1 Usage of itertools

- allows us to create different combinations of the numbers in array

```
In [32]: import itertools

         print(list(itertools.product([0,1],repeat = 1)))
         print(list(itertools.product([0,1],repeat = 2)))
         print(list(itertools.product([0,1],repeat = 3)))
```

```
[(0,), (1,)]
[(0, 0), (0, 1), (1, 0), (1, 1)]
[(0, 0, 0), (0, 0, 1), (0, 1, 0), (0, 1, 1), (1, 0, 0), (1, 0, 1), (1, 1, 0), (1, 1, 1)]
```

```
In [33]: import itertools
         def confusion_matrix(ytrue, ypred, labels):
             output = {i : 0 for i in list(itertools.product(labels,repeat = 2))}

             for idx in range(ytrue.shape[0]):
                 actual = ytrue[idx,0]
                 pred = ypred[idx,0] # or using a list index ypred[idx][0]
                 output[(actual,pred)] +=1

             return output
```

```
In [34]: result = confusion_matrix(df_target_test.values, pred, [1,0])
         print(result)
         assert result == {(0, 0): 100, (0, 1): 1, (1, 0): 12, (1, 1): 57}
```

```
{(0, 0): 100, (0, 1): 1, (1, 0): 12, (1, 1): 57}
```

CS7. *Metrics:* Write a function `calc_accuracy()` that takes in a Confusion Matrix array and output a dictionary with the following keys and values: - `accuracy`: total number of correct predictions / total number of records - `sensitivity`: total correct positive cases / total positive cases - `specificity`: total false positives / total negative cases - `precision`: total of correct positive cases / total predicted positive cases

```
In [35]: def calc_accuracy(cm):
             tp = cm[(1, 1)]
             tn = cm[(0,0)]
             fp = cm[(0,1)]
             fn = cm[(1,0)]
             total = tp + tn + fp + fn

             accuracy = (tp + tn)/total
             sensitivity = tp/(tp+fn)
             specificity = tn/(fp+tn)
             precision = tp/(tp+fp)
             result = {'accuracy': accuracy, 'sensitivity': sensitivity,
                       'specificity': specificity, 'precision': precision}
             return result
```

```
In [36]: ans = calc_accuracy(result)
         expected = {'accuracy': 0.9235294117647059, 'sensitivity': 0.8260869565217391, 'speci
         assert np.isclose(ans['accuracy'], expected['accuracy'])
         assert np.isclose(ans['sensitivity'], expected['sensitivity'])
         assert np.isclose(ans['specificity'], expected['specificity'])
```

```
        assert np.isclose(ans['precision'], expected['precision'])
```

CS8. *Optional:* Redo the above tasks using Scikit Learn libraries. You will need to use the following: - LogisticRegression - train_test_split - confusion_matrix

```
In [37]: from sklearn.linear_model import LogisticRegression
         from sklearn.model_selection import train_test_split
         from sklearn.metrics import confusion_matrix

In [38]: columns = ["radius_mean", "texture_mean", "perimeter_mean", "area_mean", "smoothness_r
         # get the features and the columns
         df_features = None

         # replace target values with 0 and 1
         df_target = None

         ###
         ### YOUR CODE HERE
         ###

In [39]: # split data set using random_state = 100 and 30% test size
         df_features_train, df_features_test, df_target_train, df_target_test = None, None, Nor

         # change feature to numpy array and append column of 1s
         feature = None

         # change target to numpy array
         target = None

         ###
         ### YOUR CODE HERE
         ###

In [40]: # create LogisticRegression object instance, use newton-cg solver
         model = None

         # build model
         pass

         # get predicted value
         pred = None

         ###
         ### YOUR CODE HERE
         ###

In [41]: # calculate confusion matrix
         cm = None
         ###
         ### YOUR CODE HERE
         ###

In [42]: expected = np.array([[58,  11], [6, 96]])
         assert (cm == expected).all()


        ---------------------------------------------------------------------------

        AssertionError                            Traceback (most recent call last)

        <ipython-input-42-5489459f6fd4> in <module>
          1 expected = np.array([[58,  11], [6, 96]])
    ----> 2 assert (cm == expected).all()


        AssertionError:


In [ ]: plt.scatter(df_features_test["radius_mean"], df_target_test)
        plt.scatter(df_features_test["radius_mean"], pred)

In [ ]: plt.scatter(df_features_test["texture_mean"], df_target_test)
        plt.scatter(df_features_test["texture_mean"], pred)

In [ ]: plt.scatter(df_features_test["perimeter_mean"], df_target_test)
        plt.scatter(df_features_test["perimeter_mean"], pred)
```

Week10_Homework

December 10, 2021

1 Week 10 Problem Set

1.1 Homeworks

```
In [3]: import pandas as pd
        import numpy as np
        import matplotlib.pyplot as plt
```

HW0. Do the following before starting the homework questions.

Task 1. Paste the following functions from your cohort sessions: - `get_features_target()` - `normalize_z()` - `normalize_minmax()` - `replace_target()` - `split_data()` - `prepare_feature()` - `prepare_target()` - `log_regression()` - `compute_cost_logreg()` - `gradient_descent_logreg()` - `predict_norm()` - `predict()` - `confusion_matrix()`

```
In [4]: def normalize_z(df):
            dfout = (df - df.mean(axis=0)) / df.std(axis=0)
            return dfout

        def get_features_targets(df, feature_names, target_names):
            df_feature = df[feature_names]
            df_target = df[target_names]
            return df_feature, df_target

        def prepare_feature(df_feature): # creates X matrix
            feature = df_feature.to_numpy()
            array1 = np.ones((feature.shape[0], 1))
        #     array1 = np.ones(feature.shape) --> works too
            X = np.concatenate((array1, feature), axis=1)
            return X

        def prepare_target(df_target): # creates numpy array for y (target)
            return df_target.to_numpy()

        def split_data(df_feature, df_target, random_state=None, test_size=0.5):
            # assuming that index is consistent between features and target
            indices = df_target.index
            if random_state != None:
                np.random.seed(random_state)
```

```
    # k is the no. of rows in the test set
    num_rows = len(indices)
    k = int(test_size * num_rows)
    # randomly choose indices for test set
    test_indices = np.random.choice(indices, k, replace=False)

    indices = set(indices)
    test_indices = set(test_indices)
    train_indices = indices - test_indices

    df_feature_train = df_feature.loc[train_indices, :]
    df_feature_test = df_feature.loc[test_indices, :]
    df_target_train = df_target.loc[train_indices, :]
    df_target_test = df_target.loc[test_indices, :]

    return df_feature_train, df_feature_test, df_target_train, df_target_test

def replace_target(df_target, target_name, map_vals):
    df_out = df_target.copy()
    df_out.loc[:, target_name] = df_target[target_name].apply(lambda val: map_vals[val]

    return df_out

def normalize_minmax(dfin):

    dfout = (dfin - dfin.min(axis=0))/(dfin.max(axis=0)-dfin.min(axis=0))

    return dfout

def log_regression(beta, X):
    z = np.matmul(X, beta)
    hypothesis = 1 / (1 + np.exp(-z))
    return hypothesis

def compute_cost_logreg(beta, X, y):
    np.seterr(divide = 'ignore')
    p = log_regression(beta, X)
    m = X.shape[0]
    J = (-1 / m) * np.sum(np.where(y == 1, np.log(p), np.log(1 - p)))
    np.seterr(divide = 'warn')
    return J

def gradient_descent_logreg(X, y, beta, alpha, num_iters):
    m = X.shape[0]
    J_storage = np.zeros(num_iters)
    for n in range(num_iters):
        p = log_regression(beta, X)
        error = p - y
        delta = np.matmul(X.T, error)
        beta = beta - (alpha / m) * delta
        # change the J value from 0 to sum cost (can also use an empty list and append
        J_storage[n] = compute_cost_logreg(beta, X, y)
    return beta, J_storage

def predict_norm(X, beta):
    p = log_regression(beta, X)
    return np.where(p >= 0.5, 1, 0)

def predict(df_feature, beta):
    feature_z = normalize_z(df_feature)
    X = prepare_feature(feature_z)
    return predict_norm(X, beta)

import itertools
def confusion_matrix(ytrue, ypred, labels):
    output = {i : 0 for i in list(itertools.product([0,1],repeat = 2))}

    for idx in range(ytrue.shape[0]):
        actual = ytrue[idx,0]
        pred = ypred[idx,0] # or using a list index ypred[idx][0]
        output[(actual,pred)] +=1

    return output
```

Task 2. Load the Iris data set from `iris_data.csv` into a Data Frame.

```
In [5]: # read iris_data.csv
        df = pd.read_csv("iris_data.csv")
        df.head()
```

```
Out[5]:    sepal_length  sepal_width  petal_length  petal_width      species
        0           5.1          3.5           1.4          0.2  Iris-setosa
        1           4.9          3.0           1.4          0.2  Iris-setosa
        2           4.7          3.2           1.3          0.2  Iris-setosa
        3           4.6          3.1           1.5          0.2  Iris-setosa
        4           5.0          3.6           1.4          0.2  Iris-setosa
```

Task 3. Do the following tasks.

- Read the following columns for the features: `'sepal_length'`, `'sepal_width'`, `'petal_length'`, `'petal_width'`.
- Read the column species for the target.
- Replace the species column with the following mapping:
  - Iris-setosa: 0
  - Iris-versicolor: 1

– Iris-virginica: 2

```
In [6]: columns = ['sepal_length', 'sepal_width', 'petal_length', 'petal_width']
        mapping = {'Iris-setosa': 0, 'Iris-versicolor': 1, 'Iris-virginica':2}

        # extract the features and the target
        df_features, df_target = get_features_targets(df, columns, ["species"])

        # replace the target using the mapping
        df_target = replace_target(df_target, "species", mapping)
```

```
In [7]: result = np.unique(df_target['species'], return_counts=True)
        assert (result[0] == [0, 1, 2]).all()
        assert (result[1] == [50, 50, 50]).all()
```

HW1. *One-vs-All target:* Write a function that takes in a target data frame and returns a new dataframe where the size of the column is the same as the number of category. The function makes use of `replace_target()` function to create one-vs-all target values.

For example, if we have three categories of class, the columns of the returned data frame will be as follows: - column target: this is the original target column - column 0: the target with values of 0 will be set to 1 while the rest will be replaced with 0. - column 1: the target with values of 1 will be set to 1 while the rest will be replaced with 0. - column 2: the target with values of 2 will be set to 1 while while the rest will be replaced with 0.

```
In [8]: def create_onevsall_columns(df_target, col):
            dfout = df_target.copy()
            num_classes = df_target[col].nunique()
            for i in range(num_classes):
                dfout[i] = dfout[col].apply(lambda y : np.where(y == i,1,0))

            return dfout
```

```
In [9]: df_targets = create_onevsall_columns(df_target, 'species')
        print(df_targets)
        result = np.unique(df_targets['species'], return_counts=True)
        assert (result[0] == [0, 1, 2]).all()
        assert (result[1] == [50, 50, 50]).all()
        result = np.unique(df_targets[0], return_counts=True)
        assert (result[0] == [0, 1]).all()
        assert (result[1] == [100, 50]).all()
        result = np.unique(df_targets[1], return_counts=True)
        assert (result[0] == [0, 1]).all()
        assert (result[1] == [100, 50]).all()
        result = np.unique(df_targets[2], return_counts=True)
        assert (result[0] == [0, 1]).all()
        assert (result[1] == [100, 50]).all()
```

```
     species  0  1  2
0          0  1  0  0
```

```
1          0  1  0  0
2          0  1  0  0
3          0  1  0  0
4          0  1  0  0
..       ... .. .. ..
145        2  0  0  1
146        2  0  0  1
147        2  0  0  1
148        2  0  0  1
149        2  0  0  1

[150 rows x 4 columns]
```

HW2. *Multiple features and splitting of data set:* Do the following task in the code below: - Read the following columns for the features: `sepal_length`,`sepal_width`, `petal_length`, `petal_width` normalize it using `normalize_z()`. - Read `species` as the target column and use `create_onevsall_columns()` to create the additional target columns to do multi class classification. - Split the data set with 30% test size and `random_state = 100`. - Normalize the training feature data set using `normalize_z()` function. - Convert to numpy array both the target and the features using `prepare_feature()` and `prepare_target()` functions. - Call `gradient_descent()` function to get the parameters using the training data set.

```
In [10]: columns = ['sepal_length', 'sepal_width', 'petal_length', 'petal_width']
         mapping = {'Iris-setosa': 0, 'Iris-versicolor': 1, 'Iris-virginica':2}

         # extract the features and the target
         df_features, df_target = get_features_targets(df, columns, ["species"])

         # change target values to integer using mapping
         df_target = replace_target(df_target, "species", mapping)

         # create one vs all columns for the target
         df_targets = create_onevsall_columns(df_target, 'species')

         # split the data using random_state = 100 and 30% test size
         df_features_train, df_features_test, df_targets_train, df_targets_test = split_data(d

         # normalize the training feature
         df_features_train_z = normalize_z(df_features_train)
```

```
In [11]: assert df_features_train_z.shape == (105, 4)

         assert np.isclose(df_features_train_z.min(), -2.52349).any()
         assert np.isclose(df_features_train_z.max(), 2.73284).any()
         assert np.isclose(df_features_train_z['sepal_width'].mean(), 0)
         assert np.isclose(df_features_train_z['sepal_width'].std(), 1, atol=0.01)
         assert (np.unique(df_targets_train['species']) == [0, 1, 2]).all()
         assert (np.unique(df_targets_train[0]) == [0, 1]).all()
         assert (np.unique(df_targets_train[1]) == [0, 1]).all()
         assert (np.unique(df_targets_train[2]) == [0, 1]).all()
```

HW3. *Build Multi-class Model:* Write a function `build_model_multiclass()` which takes in the following arguments: - `df_features`: which is a Pandas data framecontaining the features. - `df_targets`: which is a Pandas data frame containing the target for one vs all classification. - `col_target`: the name of the column target in the original data frame which is also the key of the dictionary containing the original target numpy array. - `iterations`: the number of iterations to perform the gradient descent. By default it is set to 1500. - `alpha`: the learning rate in the gradient descent algorithm. By default it is set to 0.01.

The function should return a dictionary of dictionary. The output dictionary has the following key and values: - key: the keys are the categories or the labels in the target. - values: the values are another dictionary for that particular label. This dictionary has two keys: `beta` and `J_storage`, which gives the parameter value for that particular label and its cost minimization values at every iteration.

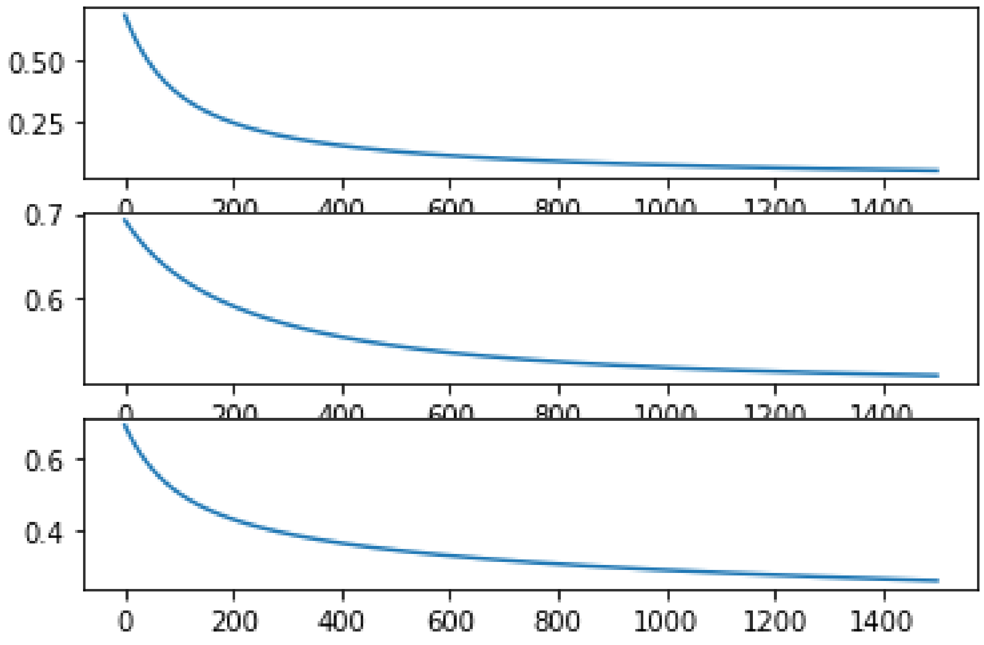
Hint: - you need to call `prepare_feature()` and `prepare_target()` to change the Pandas data frame to Numpy arrays. - in order to create a data frame instead of a series when accessing a column, use `df[[c]]` (will output data frame) instead of `df[c]` (will output series). - You need to use `normalize_minmax()` on your target before passing it on to `gradient_descent_logreg()` because the function logistic regression has the normalized value of 0 to 1 in the y axis.

```
In [12]: def build_model_multiclass(df_features, df_targets, col_target, iterations=1500, alpha
             output = {}

             # change the feature columns to numpy array and append column of 1s
             features = prepare_feature(df_features)

             # change the target column to numpy array

             # prepare target for every column in targets
             for column in df_targets.columns:
                 if column == col_target:
                     continue

                 values = {}
                 target = prepare_target(df_targets[[column]])

                 target = normalize_minmax(target)

                 # provide initial guess for theta
                 beta = np.zeros((features.shape[1],1))

                 # call the gradient descent method
                 beta, J_storage = gradient_descent_logreg(features, target, beta, alpha, itera
                 values["beta"] = beta
                 values["J_storage"] = J_storage
                 output[column] = values

             return output
```

```
In [13]: output = build_model_multiclass(df_features_train_z, df_targets_train, 'species')

         assert isinstance(output, dict)
         expected = np.array([[ -1.0198841], [ -0.69883077], [  1.0774116], [-1.17170999], [-1
         assert np.isclose(output[0]['beta'], expected).all()
         expected = np.array([[ -0.63304937], [ 0.11684857], [-1.15346071], [ 0.18746937], [-0
         assert np.isclose(output[1]['beta'], expected).all()
         expected = np.array([[-1.31740148 ], [0.42271871], [0.18526839], [ 0.8831822], [1.1792
         assert np.isclose(output[2]['beta'], expected).all()
```

```
In [14]: fig, axes = plt.subplots(len(output), 1)
         idx = 0
         for c in output:
             print(f'class model = {c:}', output[c]['beta'])
             axes[idx].plot(output[c]['J_storage'])
             idx += 1
```

```
class model = 0 [[-1.0198841 ]
 [-0.69883077]
 [ 1.0774116 ]
 [-1.17170999]
 [-1.12846826]]
class model = 1 [[-0.63304937]
 [ 0.11684857]
 [-1.15346071]
 [ 0.18746937]
 [-0.14534827]]
class model = 2 [[-1.31740148]
 [ 0.42271871]
 [ 0.18526839]
 [ 0.8831822 ]
 [ 1.17929455]]
```



HW4. *Predict Multi-class:* Write a function `predict_multiclass()` that takes in the data frame for the features and the parameters for the multi-class classification and return a Numpy array for the predicted target.

Recall that you need to do the following steps: - Normalize the features and change to numpy array - For each of the class, calculate the probability by using `log_regression()` function. - For each record, find the class that gives the maximum probability. - Returns a Numpy array with the predicted target values

You can use the following function in your code: - `np.argmax()` to find the column name with the maximum value - `df.apply(func, axis=1)`: which is to apply some function on a particular axis. Setting axis=1 means that the function is to be applied accross the columns of the data frame instead of the index or the rows.

```
In [15]: output

Out[15]: {0: {'beta': array([[-1.0198841 ],
                [-0.69883077],
                [ 1.0774116 ],
                [-1.17170999],
                [-1.12846826]]),
           'J_storage': array([0.68739949, 0.68172973, 0.67613683, ..., 0.05490399, 0.05487285
                0.05484174])},
          1: {'beta': array([[-0.63304937],
                [ 0.11684857],
                [-1.15346071],
                [ 0.18746937],
                [-0.14534827]]),
           'J_storage': array([0.69217867, 0.69121867, 0.6902671 , ..., 0.50952092, 0.50950849
                0.50949608])},
        2: {'beta': array([[-1.31740148],
                [ 0.42271871],
                [ 0.18526839],
                [ 0.8831822 ],
                [ 1.17929455]]),
         'J_storage': array([0.68991177, 0.68671789, 0.68356495, ..., 0.26207724, 0.26203146
                0.26198572])}}
```

```
In [16]: def predict_multiclass(df_features, multi_beta):
             # normalize the training feature
             df_features_z = normalize_z(df_features)

             features = prepare_feature(df_features_z)

             num_features = len(multi_beta)
             pred = pd.DataFrame()
             for i in range(num_features):
                 beta = multi_beta[i]["beta"]
                 y_pred = log_regression(beta,features).ravel()
                 pred[i] = pd.Series(y_pred)

             pred['final'] = pred.apply(lambda x : np.argmax(x),axis = 1)


             return pred['final'].to_numpy().reshape(45,1)
```

```
In [17]: pred = predict_multiclass(df_features_test, output)
```

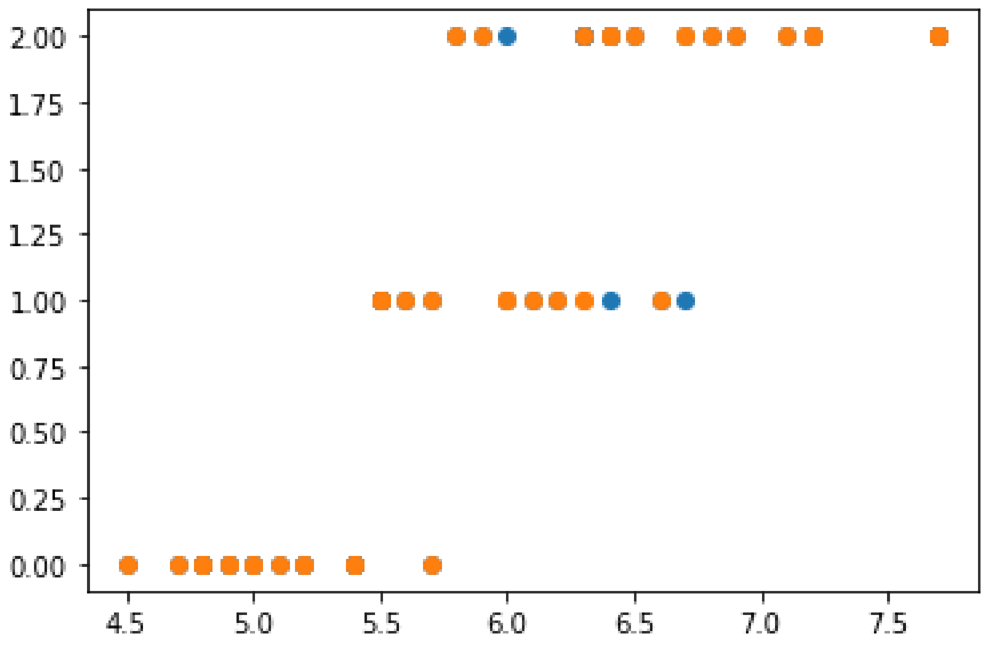
```
In [18]: pred.shape
```

```
Out[18]: (45, 1)
```

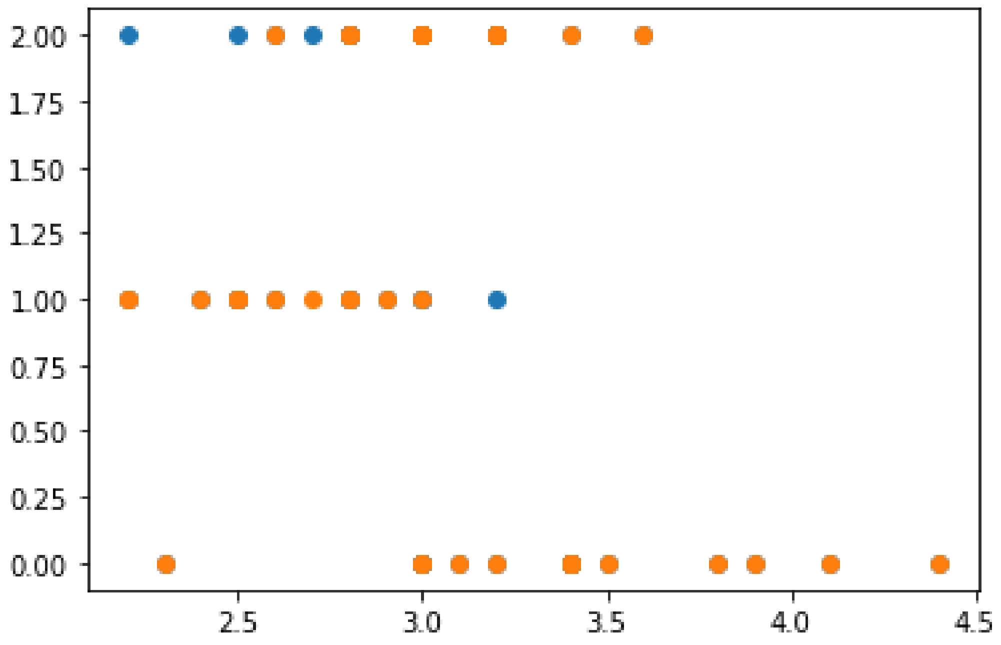
```
In [19]: pred = predict_multiclass(df_features_test, output)

         assert isinstance(pred, np.ndarray)
         assert pred.shape == (45, 1)
         assert pred.min() == 0
         assert pred.max() == 2
         assert np.median(pred) == 1
```

```
In [20]: plt.scatter(df_features_test['sepal_length'], df_targets_test['species'])
         plt.scatter(df_features_test['sepal_length'], pred)
```

```
Out[20]: <matplotlib.collections.PathCollection at 0x7f4cd7a91610>
```

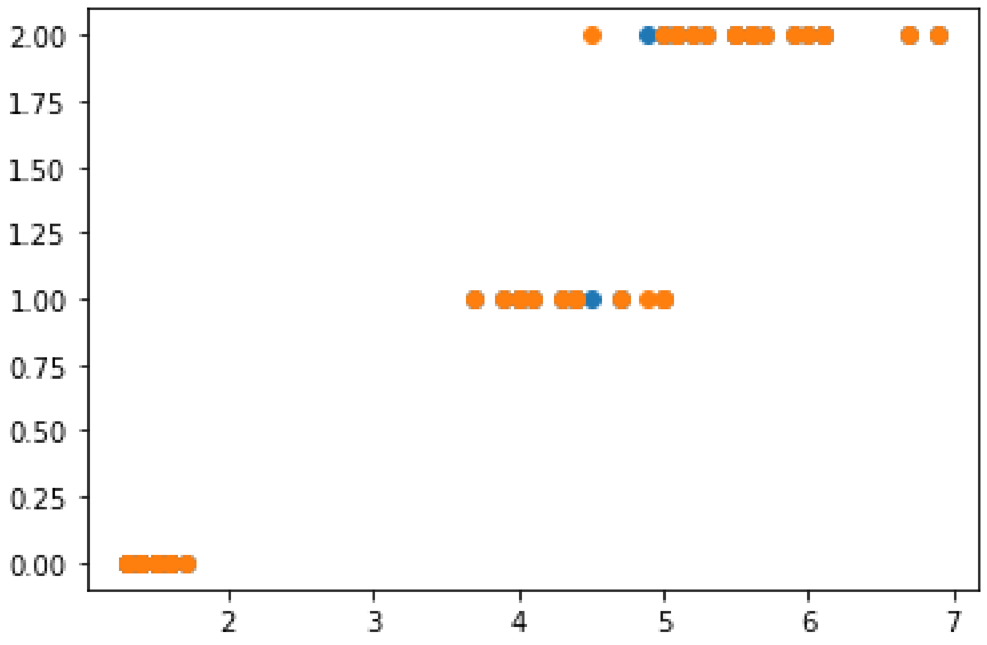


```
In [21]: plt.scatter(df_features_test['sepal_width'], df_targets_test['species'])
         plt.scatter(df_features_test['sepal_width'], pred)

Out[21]: <matplotlib.collections.PathCollection at 0x7f4cd53c3a10>
```



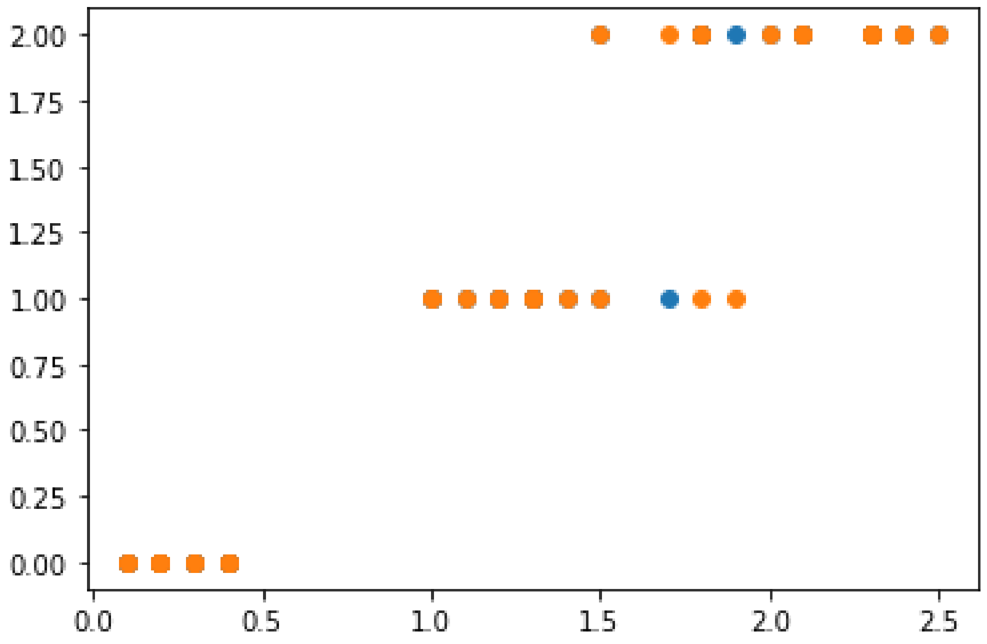
```
In [22]: plt.scatter(df_features_test['petal_length'], df_targets_test['species'])
         plt.scatter(df_features_test['petal_length'], pred)
```

```
Out[22]: <matplotlib.collections.PathCollection at 0x7f4cd5344e50>
```



```
In [23]: plt.scatter(df_features_test['petal_width'], df_targets_test['species'])
         plt.scatter(df_features_test['petal_width'], pred)
```

```
Out[23]: <matplotlib.collections.PathCollection at 0x7f4cd52c9c50>
```



HW5. *Confusion Matrix:* Write a function to calculate the confusion matrix for multi-class label. If you write the solution in the Cohort session properly, the solution will be the same as in the Cohort session.

Make sure that you can output a dictionary where the keys are all the combinations of all the classes: `(0, 0)`, `(0, 1)`, `(0, 2)`, `(1, 0)`, `(1, 1)`, `(1, 2)`, `(2, 0)`, `(2, 1)`, `(2, 2)`.

```
In [24]: import itertools
         def confusion_matrix(ytrue, ypred, labels):
             output = {i : 0 for i in list(itertools.product(labels,repeat = 2))}

             for idx in range(ytrue.shape[0]):
                 actual = ytrue[idx,0]
                 pred = ypred[idx,0] # or using a list index ypred[idx][0]
                 output[(actual,pred)] +=1

             return output
```

```
In [25]: cm = confusion_matrix(df_targets_test.values, pred, [0, 1, 2])
         print(cm)
         assert cm == {(0, 0): 16, (0, 1): 0, (0, 2): 0, (1, 0): 0, (1, 1): 9, (1, 2): 2, (2,
```

```
{(0, 0): 16, (0, 1): 0, (0, 2): 0, (1, 0): 0, (1, 1): 9, (1, 2): 2, (2, 0): 0, (2, 1): 3, (2,
```

HW6. *Metrics:* Write a function `calc_accuracy()` that takes in a Confusion Matrix array and output a dictionary with the following keys and values: - `accuracy`: total number of correct predictions / total number of records - `sensitivity`: total correct positive cases / total positive cases - `precision`: total of correct positive cases / total predicted positive cases

For multiple classes, we can also calculate *sensitivity* and *precision* for each of the class. For example, to calculate the sensitivity for class i, we use:

$$\text{sensitivity}_i = \frac{M_{ii}}{\sum_j M_{ij}}$$

This means that we get the value at row i and columnn i which is the total correct case for class i and the sum over all the columns in row i which is the total cases for class i.

Similarly, we can calculate the precision for class i using:

$$\text{precision}_i = \frac{M_{ii}}{\sum_j M_{ji}}$$

Notice that the indices are swapped for the denominator. For precision, we instead of summing over all the columns, we sum over all the rows in column i which is the total cases when class i is *predicted*.

The output is a dictionary with one of the keys called `accuracy` and the rest of the keys are the label for the different classes, i.e. 0, 1, and 2 in our example here. The value for `accuracy` key is a float. On the other hand, the values for the other label keys is another dictionary that has `sensitivity` and `precision` as the keys.

```
In [40]: cm

Out[40]: {(0, 0): 16,
          (0, 1): 0,
          (0, 2): 0,
          (1, 0): 0,
          (1, 1): 9,
          (1, 2): 2,
          (2, 0): 0,
          (2, 1): 3,
          (2, 2): 15}
```

```
In [41]: def calc_accuracy(cm, labels):
             output = {'accuracy': 0}

             correct_predictions = 0
             total_predictions = 0
             for k,v in cm.items():
                 if k[0] == k[1]:
                     correct_predictions += v
                 total_predictions += v
             output['accuracy'] = correct_predictions/total_predictions

             for l in labels:
                 tp = cm[(l, l)]
                 denom_sensititivity = 0
                 denom_precision = 0
                 for j in labels:
                     # getting sum of the row
                     denom_sensititivity += cm[(l, j)]
                     # getting sum of the column
                     denom_precision += cm[(j, l)]
                 sensitivity = tp/denom_sensititivity
                 precision = tp/denom_precision
                 output[l] = {'sensitivity':sensitivity,'precision':precision}

             return output
```

```
In [42]: metrics = calc_accuracy(cm, [0,1,2])
         print(metrics)
         assert np.isclose(metrics['accuracy'], 0.88888)
         assert metrics[0] == {'sensitivity': 1.0, 'precision': 1.0}
         assert np.isclose(metrics[0]['sensitivity'], 1.0)
         assert np.isclose(metrics[0]['precision'], 1.0)
         assert np.isclose(metrics[1]['sensitivity'], 0.8181818)
         assert np.isclose(metrics[1]['precision'], 0.75)
         assert np.isclose(metrics[2]['sensitivity'], 0.833333)
         assert np.isclose(metrics[2]['precision'], 0.88235)
```

```
0 0
16
0 1
0
0 2
0
1 0
0
1 1
9
1 2
2
2 0
0
2 1
3
2 2
15
{'accuracy': 0.8888888888888888, 0: {'sensitivity': 1.0, 'precision': 1.0}, 1: {'sensitivity':
```

HW7. *Optional:* Redo the above tasks using Scikit Learn libraries. You will need to use the following: - LogisticRegression - train_test_split - confusion_matrix

You can refer to the followign discussion on the different minimization solver for `LogisticRegression()` class. - Stack overflow - logistic regression python solvers' defintions

```
In [ ]: from sklearn.linear_model import LogisticRegression
        from sklearn.model_selection import train_test_split
        from sklearn.metrics import confusion_matrix
```

```
In [ ]: columns = ['sepal_length', 'sepal_width', 'petal_length', 'petal_width']
        mapping = {'Iris-setosa': 0, 'Iris-versicolor': 1, 'Iris-virginica':2}

        # get the features and the columns
        df_features = None

        # replace target values with integers using the mapping
        df_target = None


        ###
        ### YOUR CODE HERE
        ###
```

```
In [ ]: # split data set using random_state = 100 and 30% test size
        df_features_train, df_features_test, df_target_train, df_target_test = None, None, Non

        # change feature to numpy array and append column of 1s
        feature = None
        # change target to numpy array
        target = None

        ###
        ### YOUR CODE HERE
        ###

In [ ]: # create LogisticRegression object instance
        # set solver to 'newton-cg' and multi_class to 'auto'
        model = None

        # build model
        pass

        # get predicted value
        pred = None

        ###
        ### YOUR CODE HERE
        ###

In [ ]: # calculate confusion matrix
        cm = None

        ###
        ### YOUR CODE HERE
        ###
        print(cm)

In [ ]: expected = np.array([[16,  0,  0], [ 0, 11,  0], [ 0,  1, 17]])
        assert (cm == expected).all()

In [ ]: plt.scatter(df_features_test["sepal_width"], df_target_test)
        plt.scatter(df_features_test["sepal_width"], pred)

In [ ]: plt.scatter(df_features_test["sepal_length"], df_target_test)
        plt.scatter(df_features_test["sepal_length"], pred)

In [ ]: plt.scatter(df_features_test["petal_width"], df_target_test)
        plt.scatter(df_features_test["petal_width"], pred)

In [ ]: plt.scatter(df_features_test["petal_length"], df_target_test)
        plt.scatter(df_features_test["petal_length"], pred)
```

Week12_Homework

December 10, 2021

1 Week 12 Problem Set

1.1 Homework

HW1. *Comments:* Write a state machine whose inputs are the characters of a string. The string contains the code for a computer program. The output of the state machine are either: - the input character if it is part of a comment, or - `None`, otherwise.

Comment starts with a `#` character and continue to the end of the current line. If you want to create a string that contains a new line character, you can use `\n`.

For example,

```
inpstr = "def func(x): # comment\n    return 1"
m = CommentsSM()
print(m.transduce(inpstr))
```

The expected output is:

```
[None, None, None, None, None, None, None, None, None, None, None, None, None, "#", " ", "c", 
```

You should start by drawing a state transition diagram indicating the states and what inputs cause transition to which other states. Use the test case above to determine if your state transition diagram is correct. You should begin writing your program only when you are confident that your diagram is correct.

```
In [17]: from abc import ABC, abstractmethod

         class StateMachine(ABC):
             def __init__(self):
                 self.state = None

             def start(self):
                 self.state = self.start_state

             def step(self, inp):
                 self.state,out = self.get_next_values(self.state,inp)
                 return out

             def transduce(self, inp_list):
```

```
            output_list = []
            self.start()
            for inp in inp_list:
                if not self.is_done():
                    out = self.step(inp)
                    output_list.append(out)
            print(output_list)
            return output_list

        @abstractmethod
        def get_next_values(self, state, inp):
            pass

        def done(self, state):
            return False

        def is_done(self):
            return self.done(self.state)
```

```
In [18]: class CommentsSM(StateMachine):

             def __init__(self):
                 self.start_state = 0

             # @ args
             # state : current state
             def get_next_values(self, state, inp):
                 next_state = state
                 output = None

                 if self.state == 0 and inp == "#":
                     next_state = 1
                     output = inp

                 elif self.state == 1 and inp == "\n":
                     next_state = 0

                 elif self.state == 1 and inp != "\n":
                     output = inp


                 return next_state, output
```

```
In [19]: inpstr = "def func(x): # comment\n    return 1"
         m = CommentsSM()
         out = m.transduce(inpstr)
         assert out == [None, None, None, None, None, None, None, None, None, None, None, None
```

```
[None, None, None, None, None, None, None, None, None, None, None, None, None, '#', ' ', 'c',
In [20]: ###
         ### AUTOGRADER TEST - DO NOT REMOVE
         ###
```

HW2. *First Word:* Write a state machine whose inputs are the characters of a string and which outputs either: - the input character if it is part of the first word on a line, or - `None`, otherwise

For the purposes here, a word is any sequence of consecutive characters that does not contain spaces or end-of-line characters. In this problem, comments have no special status. This means that if the line begins with `#`, then the first word is `#`.

For example,

```
inpstr = "def func(x): # comment\n    return 1"
m = FirstWordSM()
print(m.transduce( inpstr))
```

The expected output is:

```
["d", "e", "f", None, None, None, None, None, None, None, None, None, None, None, None, None, 
```

```
In [47]: class FirstWordSM(StateMachine):

             def __init__(self):
                 self.start_state = 2

             def get_next_values(self, state, inp):
                 next_state = state
                 output = None
                 t = []

                 if self.state == 2:
                     if inp == " ":
                         next_state = 0

                     elif inp == '\n':
                         next_state = 1

                     elif inp != " ":
                         output = inp

                 elif self.state == 0 and inp == "\n":
                     next_state = 1

                 elif self.state == 1 and inp != " ":
                     next_state = 2
                     output = inp

                 return next_state, output
```

```
In [48]: inpstr = "def func(x, y): # comment\n pass\n    return 1"
         m = FirstWordSM()
         out = m.transduce(inpstr)
         assert out == ["d", "e", "f", None, None, None, None, None, None, None, None, None, No

['d', 'e', 'f', None, None, None, None, None, None, None, None, None, None, None, None, None, N


        ---------------------------------------------------------------------------

        AssertionError                            Traceback (most recent call last)

        <ipython-input-48-b0b23635bf50> in <module>
          2 m = FirstWordSM()
          3 out = m.transduce(inpstr)
    ----> 4 assert out == ["d", "e", "f", None, None, None, None, None, None, None, None, None


        AssertionError:


In [49]: inpstr = "def func(x): # comment\n    return 1"
         m = FirstWordSM()
         out = m.transduce(inpstr)
         assert out == ["d", "e", "f", None, None, None, None, None, None, None, None, None, No

['d', 'e', 'f', None, None, None, None, None, None, None, None, None, None, None, None, None, N


In [ ]: ###
        ### AUTOGRADER TEST - DO NOT REMOVE
        ###
```

HW3. *Robot:* Write a State Machine class that represent a robot. The dimension of the world and the robot initial position should be specified during the class instantiation. The robot can take in the following input: - "left" - "right" - "up" - "down"

The initial position of the robot is specified during the object instantiation and the input should modify the position of the robot. The robot position must not change if it exceed the boundary. At each step, the robot should output the updated position.

```
In [ ]: class Position:

            def __init__(self, x=0, y=0):
                self.x = x
                self.y = y

            def __str__(self):
                return f"({self.x:}, {self.y:})"
```

```
        class Dimension:

            def __init__(self, width=0, height=0):
                self.width = width
                self.height = height

            def __str__(self):
                return f"width: {self.width:}, height: {self.height:}"

In [ ]: class RobotSM(StateMachine):

            # @args
            # init_pos : Position
            # dimension : Dimension
            def __init__(self, init_pos, dimension):
                self.world_dim = dimension

                self.movement_map = {
                    "right" : (1,0),
                    "up" : (0,1),
                    "down" : (0,-1),
                    "left" : (-1,0)
                }

                self.start_state = init_pos
                self.bounded = 1

            def get_next_values(self, state, inp):
                next_state = self.compute_position(state,inp)
                output = next_state

                return next_state,output

            def compute_position(self,state, inp):
                # inp : Position
                # out : Position
                if self.bounded == 0:
                    return state
                if not self.check_dimension:
                    return state

                state.x +=  self.movement_map[inp][0]
                state.y +=  self.movement_map[inp][1]

                return state

            def check_dimension(self,state):
```

```
            # check if the element is out of out of boundary
            if state.x > self.world_dim.width or state.x < 0 or state.y > self.world_dim.he
                self.bounded = 0
                return False
            return True
```

```
In [ ]: robot = RobotSM(Position(0, 0), Dimension(5, 5))
        robot.start()
        robot.transduce(["right", "right", "up", "up", "up", "left", "down"])
        pos = robot.state
        print(pos)
        assert pos.x == 1 and pos.y == 2
```

```
In [ ]: robot = RobotSM(Position(0, 0), Dimension(5, 5))
        robot.start()
        robot.transduce(["right", "right", "up", "up", "up", "left", "down"])
        pos = robot.state
        assert pos.x == 1 and pos.y == 2
```

```
In [ ]: ###
        ### AUTOGRADER TEST - DO NOT REMOVE
        ###
```

HW4. *Search SM:* Write a function `sm_search` that takes in the following arguments: - `sm_to_search`: is the State Machine instance to search. This argument is of the type `MapSM` as defined in CS4. You should use the `get_next_values()` of this State Machine instance to explore the next state in your search. - `initial_state`: is the start state of the search. If it is not provided, it should be assigned to the `start_state` of sm_search. - `goal_test`: is a function that returns `True` if the argument is the end state of the search. If it is not provided, it should be eassigned to the `done` function of the state machine.

This function performs a **breadth-first-search** algorithm to explore the next states.

The output is a `list` of `Step` instances from the `init_state` to the end state which is determined by the `goal_test` function.

This problem requires you to complete the following: - `Queue` class from Problem Set 4 HW2. - `MapSM` class in CS4. - `SearchNode` and `Step` classes in CS5.

```
In [ ]: class Queue:
            def __init__(self):
                self.__items = []

            def enqueue(self, item):
                self.__items.append(item)

            def dequeue(self):
                return self.__items.pop(0) if not self.is_empty else None

            def peek(self):
                return self.__items[0]
```

```
    @property
    def is_empty(self):
        return len(self.__items) == 0

    @property
    def size(self):
        return len(self.__items)
```

```
In [ ]: # Copy over the implementation of StateSpaceSearch from Cohort
        from abc import abstractmethod

        class StateSpaceSearch(StateMachine):
            @property
            @abstractmethod
            def statemap(self):
                pass

            @property
            @abstractmethod
            def legal_inputs(self):
                pass
```

```
In [ ]: class MapSM(StateSpaceSearch):

            def __init__(self, start):
                self.start_state = start

            @property
            def statemap(self):
                # There are 4 actions : 0,1,2,3 -- encoded
                statemap = {"S": ["A", "B"],
                            "A": ["S", "C", "D"],
                            "B": ["S", "D", "E"],
                            "C": ["A", "F"],
                            "D": ["A", "B", "F", "H"],
                            "E": ["B", "H"],
                            "F": ["C", "D", "G"],
                            "H": ["D", "E", "G"],
                            "G": ["F", "H"]}
                return statemap

            @property
            def legal_inputs(self):
                max_neighbour = -1
                for _,neighbours in self.statemap.items():
                    max_neighbour = max(max_neighbour,len(neighbours))
                return set(range(max_neighbour))
```

```
    def get_next_values(self, state, inp):
        neighbours = self.statemap.get(state,None)

        # default values
        next_state = state
        output = state

#          if not neighbours:
#              return next_state,output

        if inp < len(neighbours) and neighbours:
            next_state = neighbours[inp]
            output = next_state

        return next_state, output
```

```
In [ ]: class Step:
            def __init__(self, action, state):
                self.action = action
                self.state = state

            def __eq__(self, other):
                return self.action == other.action and self.state == other.state

            def __str__(self):
                return f"action: {self.action:}, state: {self.state:}"

        class SearchNode:
            def __init__(self, action, state, parent):
                self.state = state
                self.action = action
                self.parent = parent

            # @return : list of Step instances
            # S -> A-> C
            # C.path() --> [Step(None,S),Step(ActionA,A) ,Step(ActionC,C))]

            # Recursively calling from C.path()
            # C.path()
            # A.path()
            # S.path()
            def path(self):
                # using recursion

                # base state
                if self.parent is None:
```

```
                return [Step(self.action, self.state)]
            else:
                return self.parent.path() + [Step(self.action,self.state)]


        # @ args
        # state -> string
        # @return
        # boolean
        def in_path(self, state):
            if self.state == state: # asking if the state is the expected state
                return True
            elif self.parent == None: # not in path anymore
                return False
            else:
                # recursion to check the parent is equals to the state
                return self.parent.in_path(state) # passing the parent object as self.stat

        def __eq__(self, other):
            if self is None and other is None:
                return True
            elif self is None:
                return False
            elif other is None:
                return False
            else:
                return self.state == other.state and self.parent == other.parent and \
                       self.action == other.action
```

```
In [ ]: def sm_search(sm_to_search, initial_state=None, goal_test=None):
            # check if initial_state is provided
            # if it is, use it
            # otherwise, get the start state of sm_to_search
            if initial_state == None:
                # replace None to take the start state of sm_to_search
                init_state = sm_to_search.start_state
            else:
                init_state = initial_state

            # check if goal_test is provided,if it is, use it
            # otherwise, use the done method as the goal function
            if goal_test == None:
                goal_func = sm_to_search.done
            else:
                goal_func = goal_test

            # create a Queue instance to store the node to explore
            bfs_queue = Queue()
```

```
# if the initial state is the goal state,
# then we are done and exit
if goal_func(init_state):
    return [Step(None, init_state)]

# otherwise, add the current node into the agenda
# Start bfs from root
# We create SearchNode Instance with state as init state
bfs_queue.enqueue(SearchNode(None, init_state, None))

# explore as long as the Queue is not empty
while not bfs_queue.is_empty:

    # Take out the parent from the Queue
    current_node = bfs_queue.dequeue()

    # create a list to keep track which child state have been explored
    visited = []

    # get all the legal input values
    actions = sm_to_search.legal_inputs

    #iterate over all legal inputs
    for a in actions: # a will be 0,1,2,3
        # get the next possible state using the current action
        # get next values returns (nextstate,output). next state is a string
        next_state = sm_to_search.get_next_values(current_node.state, a)[0]

        # create a new search node from the new_s
        next_state_search_node = SearchNode(a,next_state,current_node) # Search no

        # if the new state is the goal state, then we exit and return the path
        if goal_func(next_state):
            return next_state_search_node.path()

        # Checking 2 conditions before adding them to the queue. dont want to repe
        # do not explore states that have already been explored or about to be exp
        # if the next_state is already in the list of new child state, ignore it
        elif next_state in visited:
            continue

        # if the next_state is in the path of the current node, ignore it
        # doesnt this mean that its already visited. so wouldnt the first conditio
        elif current_node.in_path(next_state):
            continue
```

```
                # otherwise, add the new state into the list
                # and the new node into the Queue
                else:
                    # step 1. add the new state into the new_child_state
                    visited.append(next_state)

                    # step 2. add the new node into the Queue
                    bfs_queue.enqueue(next_state_search_node)
        return None
```

```
In [ ]: mapSM = MapSM("S")
        ans = sm_search(mapSM , "S" , lambda s: s=="H" )
        steps = [(step.action, step.state) for step in ans]
        assert steps == [(None, "S"), (0, "A"), (2, "D"), (3, "H")]
        for step in ans:
            print(step)
```

```
In [ ]: ###
        ### AUTOGRADER TEST - DO NOT REMOVE
        ###
```

Week12_Cohort

December 10, 2021

1 Week 12 Problem Set

1.1 Cohort Sessions

CS1. Define an Abstract Class for a State Machine, called `StateMachine`. The class has two properties: - `state`: which is the current state of the machine - `start_state`: which is the initial state of the machine

The class should define the following methods: - `start()`: this method set the `state` property using the value in `start_state`. Once `state` has a value, the machine is considered started. - `step(inp)`: this method takes in the current input and returns the current output. This method should move the state machine to the next state based on the current input and its current state. You should call `get_next_values(state, inp)` in your implementation. - `done(state)`: this method always return `False`. A child class can override thid method to give a different condition to end the state machine. - `is_done()`: is to be used internally to check if the state machine should terminate or not. This method simply calls `done(state)` and pass on the current `state`. The method `transduce(inp_list)` calls this method to check if it should terminates or not. - `transduce(inp_list)`: this method calls `start()` to initialize the `state` with the `start_state` and run the state machine by calling `step(inp)` for every item in the `inp_list`. The method runs the state machine and produces the output list according to the number of input in the `inp_list` or when the state machine terminates according to the output of `is_done()` method. This method should call `is_done()` to see if it should terminate at a particular state.

This class should be an Abstract Class. Implement the following way: - `SM` class inherits from `abc.ABC`, which is Python's Abstract Base Class (ABC). - Any implementation of State Machine's instances must declare the following property: `start_state`. - Any implementation of State Machine's instances must implement the following abstract method: `get_next_values()` that takes in the current `state` and the the current `input` and output a tuple of the `next_state` and the current `output` .

```
In [3]: from abc import ABC, abstractmethod

        class StateMachine(ABC):
            def __init__(self):
                self.state = None

            def start(self):
                self.state = self.start_state

            def step(self, inp):
```

```
            self.state,out = self.get_next_values(self.state,inp)
            return out

        def transduce(self, inp_list):
            output_list = []
            self.start()
            for inp in inp_list:
                if not self.is_done():
                    out = self.step(inp)
                    output_list.append(out)
            return output_list

        @abstractmethod
        def get_next_values(self, state, inp):
            pass

        def done(self, state):
            return False

        def is_done(self):
            return self.done(self.state)
```

```
In [4]: class Test(StateMachine):
            start_state = 0

            def get_next_values(self, state, inp):
                next_state = state + inp
                output = next_state
                return next_state, output

            def done(self, state):
                if state == -1:
                    return True
                else:
                    return False

        class NoImplement(StateMachine):
            def __init__(self):
                start_state = 0

        t1 = Test()
        t1.start()
        assert t1.state == 0
        out = t1.step(2)
        print(out)
        print(t1.state)
        assert t1.state ==2 and out == 2

        t2 = Test()
        out = t2.transduce([1,2,3,4])
        print(out)
        assert out == [1, 3, 6, 10]

        t3 = Test()
        out = t3.transduce([1, -2, 3])
        assert out == [1, -1]

        try:
            t4 = NoImplement()
            raise AssertionError
        except TypeError:
            pass
```

```
2
2
[1, 3, 6, 10]
```

```
In [5]: class Test(StateMachine):
            start_state = 0

            def get_next_values(self, state, inp):
                next_state = state + inp
                output = next_state
                return next_state, output

            def done(self, state):
                if state == -1:
                    return True
                else:
                    return False

        class NoImplement(StateMachine):
            def __init__(self):
                start_state = 0

        t1 = Test()
        t1.start()
        assert t1.state == 0
        out = t1.step(2)
        assert t1.state ==2 and out == 2

        t2 = Test()
        out = t2.transduce([1,2,3,4])
```