Problem 1

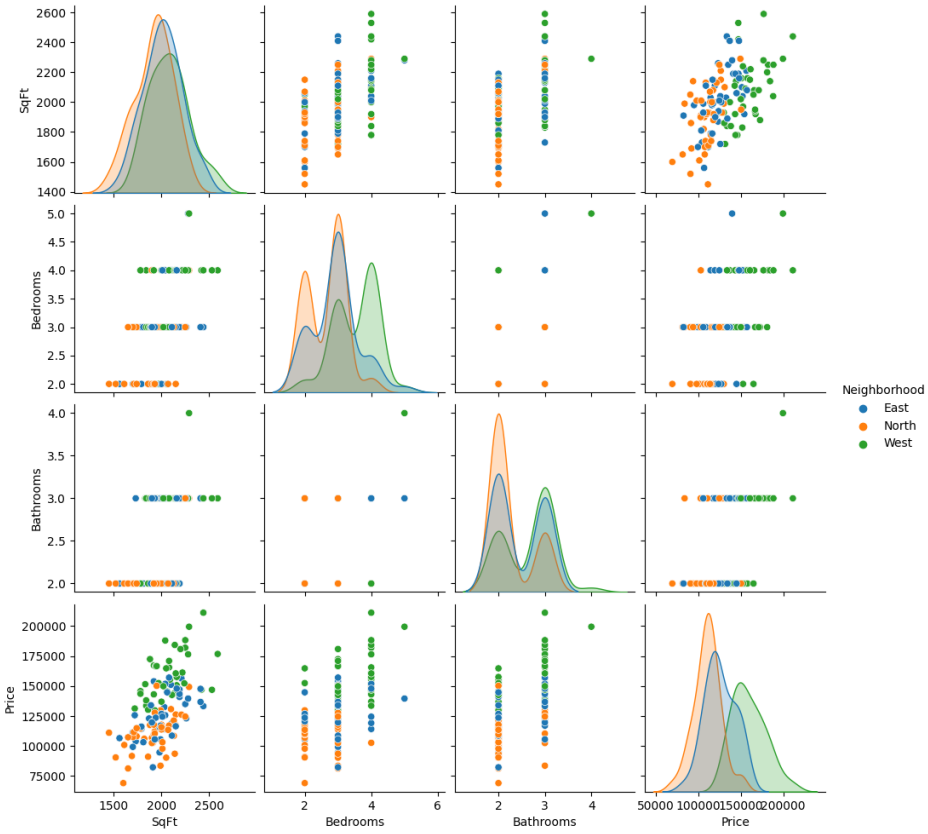
Step 1: After having a look with `df.info()` and `df.describe()`, the output shows as follows:

```
              SqFt    Bedrooms   Bathrooms          Price
count   128.000000  128.000000  128.000000     128.000000
mean   2000.937500    3.023438    2.445312  130427.343750
std     211.572431    0.725951    0.514492   26868.770371
min    1450.000000    2.000000    2.000000   69100.000000
25%    1880.000000    3.000000    2.000000  111325.000000
50%    2000.000000    3.000000    2.000000  125950.000000
75%    2140.000000    3.000000    3.000000  148250.000000
max    2590.000000    5.000000    4.000000  211200.000000
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 128 entries, 0 to 127
Data columns (total 5 columns):
 #   Column        Non-Null Count  Dtype
---  ------        --------------  -----
 0   SqFt          128 non-null    int64
 1   Bedrooms      128 non-null    int64
 2   Bathrooms     128 non-null    int64
 3   Neighborhood  128 non-null    category
 4   Price         128 non-null    int64
dtypes: category(1), int64(4)
memory usage: 4.4 KB
None
```

The dataset contains 128 entries and 5 attributes: `SqFt`, `Bedrooms`, `Bathrooms`, `Neighborhood`, and `Price`. The `Neighborhood` feature is **categorical**, and the rest are numerical. Houses range from 1450 to 2590 square feet, with 2 to 5 bedrooms and 2 to 4 bathrooms. Prices vary significantly, from 69,100 to 211,200. There are no missing values like `NaN` or `null`, and the dataset occupies 4.4 KB of memory.

Step 2: After some visualization, the output shows as:



The pairplot indicates that **larger square footage, more bedrooms, and more bathrooms** are all moderately associated with higher house prices. The distribution of points suggests that **the neighborhood** also influences house prices, with some neighborhoods appearing to have consistently higher or lower prices.

The heatmap confirms **moderate positive correlations** between house size, number of bedrooms, number of bathrooms, and price， with the correlation coefficients all above 0.5 but not so high.

Step 3: After performing the transformer using `ColumnTransformer`, we transfer the previous data set into the following `df_encoded`, with the first three column as one-hot encoding representing *Neighbourhood*, followed by the rest as *SqFt*, *Bedrooms* and *Bathrooms*. Here is part of the array:

```
array([[1.000e+00, 0.000e+00, 0.000e+00, 1.790e+03, 2.000e+00, 2.000e+00,
        1.143e+05],
       [1.000e+00, 0.000e+00, 0.000e+00, 2.030e+03, 4.000e+00, 2.000e+00,
        1.142e+05],
       [1.000e+00, 0.000e+00, 0.000e+00, 1.740e+03, 3.000e+00, 2.000e+00,
        1.148e+05],
```

After that, we randomly split the data with `train test split` into two parts for later analysis.

Step 4: By executing the linear regression using `model.fit` and `model.predict`, we report the **training RMSE** to be 1.4408543108972288e-12 and the **testing RMSE** to be 0, which seems to be a valid and accurate match.

Problem 2

Step 1: As instructed, we load the diabetes dataset. To better coordinate with the matrix operations, I add to `X` an additional first column of all ones, functioning as the intercept. Also, `y` is converted into an 2D array.

With reference, `learning_rate = 0.1` and `iterations = 1000` are chosen for the problem. `m` indicates the rows of the design matrix with intercept `X_b`, and `theta` is initialized to be $\theta_i \sim N(0, 1)$.

For the step of gradient descent, the gradient $\frac{2}{m} X_b^T(\theta^T X_b - y)$ is implemented as

```
gradients = 2/m * X_b.T.dot(X_b.dot(theta) - y)
```

and update as $\theta := \theta - \alpha \nabla f(\mathbf{x})$.

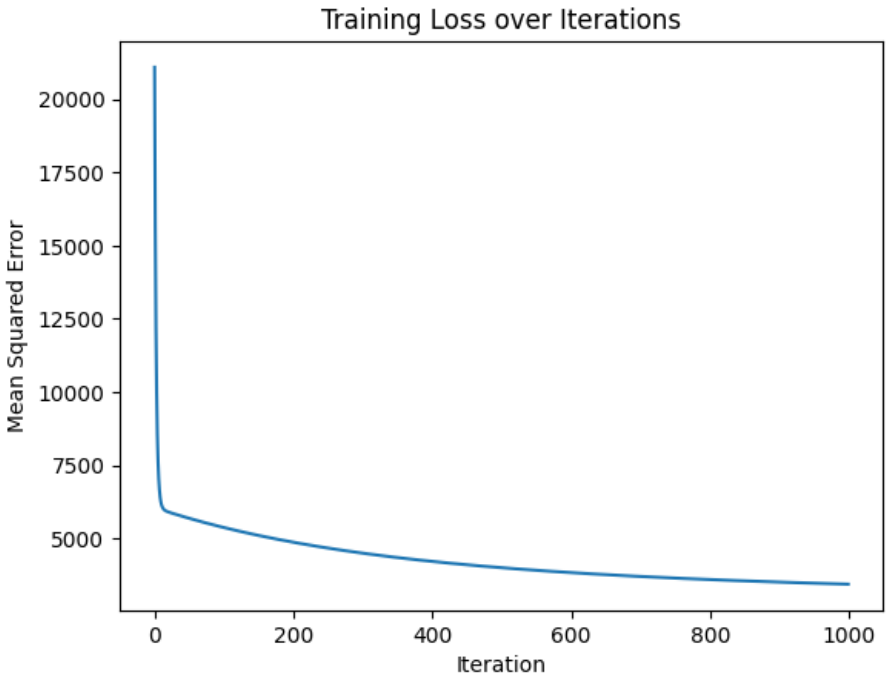
The result shows as:

```
array([[ 152.13348416],
       [  48.50843326],
       [ -32.44713402],
       [ 257.87911654],
       [ 181.25152656],
       [  35.9793122 ],
       [  10.45359258],
       [-148.13336034],
       [ 135.2707613 ],
       [ 229.07983485],
       [ 128.46635381]])
```

Step 2: After splitting the whole dataset into two, I run the linear regression again on the test set. This time a list `train_loss` is initialized, and for each iteration the loss is appended:

```
loss = mean_squared_error(y_train, X_train.dot(theta))
train_loss.append(loss)
```

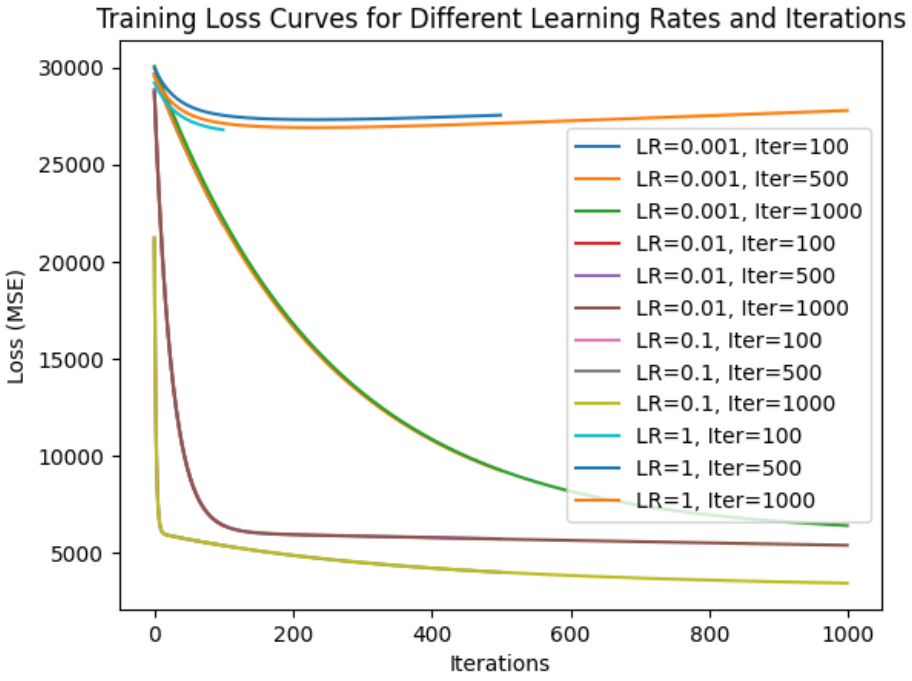
After the training, we generate both the prediction of training set and that on the test set, `y_train_pred` and `y_test_pred`, after which the RMSE is calculated respectively: The **training RMSE** is roughly 58.672, and the **testing RMSE** is 55.583. The plot looks like:



Step 3: In this part, I tuned different hyper-parameter of gradient descents, which are **learning rate** `LR` and iterations `Iter`. The only difference lies in encapsulating the steps into the `gradient_descent` function, and add two `for` iterations for every experiment. The result tells that:

```
LR: 0.001, Iter: 100, Training RMSE: 148.08575669916843, Testing RMSE: 138.59615078130028
LR: 0.001, Iter: 500, Training RMSE: 96.1373205583258, Testing RMSE: 87.37643581536264
LR: 0.001, Iter: 1000, Training RMSE: 80.07649571882452, Testing RMSE: 73.31543992980345
LR: 0.01, Iter: 100, Training RMSE: 80.09388107011452, Testing RMSE: 73.38531514979655
LR: 0.01, Iter: 500, Training RMSE: 75.69320096292356, Testing RMSE: 70.96342208236184
LR: 0.01, Iter: 1000, Training RMSE: 73.46176222414701, Testing RMSE: 68.75493975871984
LR: 0.1, Iter: 100, Training RMSE: 73.39366161382566, Testing RMSE: 68.68437119492309
LR: 0.1, Iter: 500, Training RMSE: 63.28627286684141, Testing RMSE: 59.17672788209183
LR: 0.1, Iter: 1000, Training RMSE: 58.68920034516669, Testing RMSE: 55.604002757840455
LR: 1, Iter: 100, Training RMSE: 163.6610006570304, Testing RMSE: 161.99054529715568
LR: 1, Iter: 500, Training RMSE: 165.95113689077763, Testing RMSE: 169.33777373700747
LR: 1, Iter: 1000, Training RMSE: 166.67603473647068, Testing RMSE: 170.46821567504807
```

with the corresponding curves as:



From the plot the influnces of these hyper-parameters could be analysed:

- **Iterations:** A sufficent number of iterations often yields a complete gradient descent, but it will be a waste of resources if this number is too large. An iteration of 600 could have been enough.
- **Learning rate:** If the learning rate is too large, the step might be so large that it steps over the optima all the time. If the learning rate is too small, it might take such a huge number of iterations to descend to the optima. A learning rate of 0.1 seems good.