CSE 151A HW2

PROBLEM 1:
(14, 3) would be labeled as class 1

Determined by the following code:

```
def l2_dist(x, y):
     return ((x[0]-y[0])**2 + (x[1]-y[1])**2)**0.5

class1 = [(11,11),(13,11),(8,10),(9,9),(7,7),(7,5),(16,3)]
class2 = [(7,11),(15,9),(15,7),(13,5),(14,4),(9,3),(11,3)]
sample = (14, 3)
class_1_score = 0
class_2_score = 0

for pt in class1:
     class_1_score += l2_dist(sample, pt)
for pt in class2:
     class_2_score += l2_dist(sample, pt)

if class_1_score < class_2_score:
     print("class 1")
else:
     print("class 2")
```

PROBLEM 2:

1. $L = \frac{1}{n}\sum_{i=1}^{n}\left(f(x^i) - y^i\right)^2$

$$= \boxed{\frac{1}{n}\sum_{i=1}^{n}\left(\theta_0 + \theta_1 x^i - y^i\right)^2}$$

2. $\frac{dL}{d\theta_0} = \frac{2}{n}\sum_{i=1}^{n}\left(\theta_0 + \theta_1 x^i - y^i\right)$

3. $\frac{dL}{d\theta_1} = \frac{2}{n}\sum_{i=1}^{n}\left(\theta_0 + \theta_1 x^i - y^i\right)\left(x^i\right)$

4. $\theta_0' = \theta_0 - \alpha\left[\frac{2}{n}\sum_{i=1}^{n}\left(\theta_0 + \theta_1 x^i - y^i\right)\right]$

$$\theta_1' = \theta_1 - \alpha\left[\frac{2}{n}\sum_{i=1}^{n}\left(\theta_0 + \theta_1 x^i - y^i\right)\left(x^i\right)\right]$$

PROBLEM 3:

1. $$L = \frac{1}{n}\sum_{i=1}^{n}\left(\theta_0 + \theta_1 x^i - y^i\right)^2 + \lambda|\theta_0| + \lambda|\theta_1|$$

2. $$\frac{dL}{d\theta_0} = \frac{2}{n}\sum_{i=1}^{n}\left(\theta_0 + \theta_1 x^i - y^i\right) + \begin{cases} \lambda & \theta_0 > 0 \\ \text{undefined} & \theta_0 = 0 \\ -\lambda & \theta_0 < 0 \end{cases}$$

3. $$\frac{dL}{d\theta_1} = \frac{2}{n}\sum_{i=1}^{n}\left(\theta_0 + \theta_1 x^i - y^i\right)\left(x^i\right) + \begin{cases} \lambda & \theta_0 > 0 \\ \text{undefined} & \theta_0 = 0 \\ -\lambda & \theta_0 < 0 \end{cases}$$

4. $$\theta_0' = \theta_0 - \alpha\left[\frac{2}{n}\sum_{i=1}^{n}\left(\theta_0 + \theta_1 x^i - y^i\right) + \begin{cases} \lambda & \theta_0 > 0 \\ \text{undefined} & \theta_0 = 0 \\ -\lambda & \theta_0 < 0 \end{cases}\right]$$

$$\theta_1' = \theta_1 - \alpha\left[\frac{2}{n}\sum_{i=1}^{n}\left(\theta_0 + \theta_1 x^i - y^i\right)\left(x^i\right) + \begin{cases} \lambda & \theta_0 > 0 \\ \text{undefined} & \theta_0 = 0 \\ -\lambda & \theta_0 < 0 \end{cases}\right]$$

PROBLEM 4:

1. $L = \frac{1}{n} \sum_{i=1}^{n} (\theta_0 + \theta_1 x^i - y^i)^2 + \lambda \theta_0^2 + \lambda \theta_1^2$

2. $\frac{dL}{d\theta_0} = \frac{2}{n} \sum_{i=1}^{n} (\theta_0 + \theta_1 x^i - y^i) + 2\lambda\theta_0$

3. $\frac{dL}{d\theta_1} = \frac{2}{n} \sum_{i=1}^{n} (\theta_0 + \theta_1 x^i - y^i)(x^i) + 2\lambda\theta_1$

4. $\theta_0' = \theta_0 - \alpha \left[\frac{2}{n} \sum_{i=1}^{n} (\theta_0 + \theta_1 x^i - y^i) + 2\lambda\theta_0 \right]$

   $\theta_1' = \theta_1 - \alpha \left[\frac{2}{n} \sum_{i=1}^{n} (\theta_0 + \theta_1 x^i - y^i)(x^i) + 2\lambda\theta_1 \right]$

```
import numpy as np
from tqdm import tqdm

class Linear_Regression():
    def __init__(self, alpha = 1e-3 , num_iter = 10000, early_stop = 100,
        intercept = True, init_weight = None, penalty = None,
        lam = 1e-3, normalize = False, adaptive=True):
        """
        Linear Regression with gradient descent method.

        Attributes:
        -------------
        alpha: Learning rate.
        num_iter: Number of iterations
        early_stop: Number of steps without improvements
                    that triggers a stop training signal
        intercept: True = with intercept (bias), False otherwise
        init_weight: Optional. The initial weights passed into the model,
                                for debugging
        penalty: {None,l1,l2}. Define regularization type for regression.
                  None: Linear Regression
                  l1: Lasso Regression
                  l2: Ridge Regression
        lam: regularization constant.
        normalize: True = normalize data, False otherwise
        adaptive: True = adaptive learning rate, False = fixed learning rate
        """
        self.alpha = alpha
        self.num_iter = num_iter
        self.early_stop = early_stop
        self.intercept = intercept
        self.init_weight = init_weight
        self.penalty = penalty
        self.lam = lam
        self.normalize = normalize
        self.adaptive = adaptive

    def fit(self, X, y):
        # initialize X, y
        self.X = X
        self.y = np.array([y]).T

        self.max = np.zeros((1, X.shape[1]))
        self.min = np.zeros((1, X.shape[1]))
        for i in range(self.X.shape[1]):
            self.max[:, i] = self.X[:, i].max()
            self.min[:, i] = self.X[:, i].min()

        ############### START TODO 1 ###############
        # Normalize the data using the formula provided in lecture
        if self.normalize:
            self.X = (self.X - self.min) / (self.max - self.min)
        ############### END TODO 1 ###############

        ############### START TODO 2 ###############
        # Add bias (if necessary) by concatanating a constant column into X
        # Hint: go through HW1 Q5 might be helpful
        if self.intercept:
            col = np.ones((len(self.X), 1))
            self.X = np.hstack((self.X, col))
        ############### END TODO 2 ###############

        # initialize coefficient
        self.coef = self.init_weight if self.init_weight is not None\
        else np.array([np.random.uniform(-1,1,self.X.shape[1])]).T

        # start training, self.loss is used to record losses over iterations
        self.loss = []
        self.gradient_descent()

    def gradient(self):
        coef = -2 / len(self.X)

        ############### START TODO 3 ###############
        # Find prediction and gradient
```

```python
        # Hint: Find the model's prediction from the given inputs with the
        #       coefficient, then calculate the gradient
        # If you forgot the formula, find them in lecture 4 and 5
        pred = self.X @ self.coef
        grad = coef * (self.X.T @ (self.y - pred))
        ############### END TODO 3 ###############

        ############### START TODO 4 ###############
        # Implement regularization penalty
        # Hint: Use self.lam
        if self.penalty == 'l2':
            grad += 2 * self.lam * self.coef
        elif self.penalty == 'l1':
            grad += self.lam * np.sign(self.coef)
        else:
            pass
        ############### END TODO 4 ###############

        return grad

    def gradient_descent(self):
        print('Start Training')
        for i in range(self.num_iter):

            ############### START TODO 5 ###############
            # calculate prediction y based on current coefficients (self.coef)
            previous_y_hat = self.X @ self.coef
            grad = self.gradient()
            # calculate the new coefficients after incorporating the gradient
            temp_coef = self.coef - (self.alpha * grad)
            ############### END TODO 5 ###############

            ############### START TODO 6 ###############
            # calculate regularization cost (alias: regularization loss) based on
            # self.coef and temp_coef
            if self.penalty == 'l2':
                previous_reg_cost = self.lam * np.sum(np.square(self.coef))
                current_reg_cost = self.lam * np.sum(np.square(temp_coef))
            elif self.penalty == 'l1':
                previous_reg_cost = self.lam * np.sum(np.abs(self.coef))
                current_reg_cost = self.lam * np.sum(np.abs(temp_coef))
            else:
                previous_reg_cost = 0
                current_reg_cost = 0
            ############### END TODO 6 ###############

            ############### START TODO 7 ###############
            # Calculate error (alias: loss) using sum squared loss
            # and add regularization cost
            pre_error = np.sum(np.square( self.y - previous_y_hat )) + previous_reg_cost
            current_y_hat = self.X @ temp_coef
            current_error = np.sum(np.square( self.y - current_y_hat )) + current_reg_cost
            ############### END TODO 7 ###############

            # Early Stop: early stop is triggered if loss is not decreasing
            # for some number of iterations
            if len(self.loss) > self.early_stop and \
            self.loss[-1] >= max(self.loss[-self.early_stop:]):
                print('-----------------------')
                print(f'End Training (Early Stopped at iteration {i})')
                return self

            ############### START TODO 8 ###############
            # Implement adaptive learning rate

            # Rules: if current error is smaller than previous error,
            # multiply the current learning rate by 1.3 and update coefficients,
            # otherwise by 0.9 and do nothing with coefficients
            if current_error < pre_error:
                self.alpha = 1.3 * self.alpha if self.adaptive else self.alpha
                self.coef = temp_coef
            else:
                self.alpha = 0.9 * self.alpha if self.adaptive else self.alpha
            ############### END TODO 8 ###############
```

```python
            # record stats
            self.loss.append(float(current_error))
            if i % 1000000 == 0:
                print('-----------------------')
                print('Iteration: ' +  str(i))
                print('Coef: '+ str(self.coef))
                print('Loss: ' + str(current_error))
        print('-----------------------')
        print('End Training')
        return self

    def predict(self, X):
        X_norm = np.zeros(X.shape)
        for i in range(X.shape[1]):
            X_norm[:, i] = (X[:, i] - self.min[:, i]) / (self.max[:, i] - self.min[:, i])
        X = X_norm
        ############### START TODO 9 ###############
        # add bias (if necessary, same as TODO 2)
        if self.intercept:
            col = np.ones((len(X), 1))
            X = np.hstack((X, col))

        # Find the model's predictions
        # Hint: Use matrix multiplication ('@' might come in handy here)
        y = X @ self.coef
        return y
        ############### END TODO 9 ###############

    # Congrats! You have reached the end of this model's implementation :)
```

Introduction

Linear regression generally have the form of $Y_i = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + ...$
There are several ways to find the coefficients of the regression:

1. Linear Algebra: $\hat{\theta} = (X^T X)^{-1} X^T Y$ (When X is invertible)
2. Gradient Descent: In this case, we need to write out the loss function and try to minimize the loss.

$$F(x) = \text{Loss Function} = \text{MSE} = \frac{1}{n} \sum_{i=1}^{n} (Y_i - \hat{Y}_i)^2$$

In this part of the assignment, we will be using the second way to implement this linear regression model. More details about the model's implementation can be found in corresponding lectures.

ATTENTION: THERE ARE A TOTAL OF 4 QUESTIONS THAT NEED YOUR ANSWERS

Import necessary packages

You'll be implement your model in `LinearRegression.py` which should be put under the same directory as the location of `Linear_Regression.ipynb`. Since we have enabled `autoreload`, you only need to import these packages once. You don't need to restart the kernel of this notebook nor rerun the next cell even if you change your implementation for `LinearRegression.py` in the meantime.

A suggestion for better productivity if you never used jupyter notebook + python script together: you can split your screen into left and right parts, and have your left part displaying this notebook and have your right part displaying your `LinearRegression.py`

```
# Please do not change this code block
%load_ext autoreload
%autoreload 2

# import numpy, pandas, pyplot for arrays, dataframes, and visualizations
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# import sklearn model to validate our custom model
from sklearn.preprocessing import MinMaxScaler
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split

# Please make sure that your `LinearRegression.py` is under the same folder as this .ipynb notebook
from LinearRegression import Linear_Regression
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

Experiment 1: Perfect Data

In this part, we generate a dataset with a perfect linear relationship to test our model's performance. Here, we use the equation: $y = 5x + 10$ to generate our dataset.

```
X = np.array([np.arange(1, 1000, 5)]).T
y = np.array((5 * X)).flatten() +  10
f'x = {X[:5].flatten()}, y = {y[:5]} for the first 5 values'
```

```
'x = [ 1  6 11 16 21], y = [ 15  40  65  90 115] for the first 5 values'
```

First, let's try to fit our model without any normalization (note: the below cell block could take significant amount of time to complete)

```
%%time
reg = Linear_Regression(num_iter = 10000000)
reg.fit(X,y)
print(f'\nNumber of total iterations: {len(reg.loss)} \nBest Loss: {min(reg.loss)}')
```

```
Start Training
-----------------------
Iteration: 0
Coef: [[-0.4340051 ]
 [-0.92112182]]
Loss: 865767848198043.2
-----------------------
Iteration: 1000000
Coef: [[5.0106459 ]
 [2.92990542]]
```

```
Loss: 2510.644980632616
-----------------------
Iteration: 2000000
Coef: [[5.00689505]
 [5.41968713]]
Loss: 1053.7086796984718
-----------------------
Iteration: 3000000
Coef: [[5.00446736]
 [7.03267452]]
Loss: 442.2435555864984
-----------------------
Iteration: 4000000
Coef: [[5.00289442]
 [8.07763687]]
Loss: 185.61057411811458
-----------------------
Iteration: 5000000
Coef: [[5.00187541]
 [8.75461011]]
Loss: 77.90117324915104
-----------------------
Iteration: 6000000
Coef: [[5.00121461]
 [9.19318211]]
Loss: 32.694892302952795
-----------------------
Iteration: 7000000
Coef: [[5.00078709]
 [9.4773085 ]]
Loss: 13.722166222822484
-----------------------
Iteration: 8000000
Coef: [[5.00051002]
 [9.6613777 ]]
Loss: 5.759237950299417
-----------------------
Iteration: 9000000
Coef: [[5.00033055]
 [9.78062577]]
Loss: 2.417189394568939
-----------------------
End Training

Number of total iterations: 10000000
Best Loss: 1.0144662355661658
CPU times: user 7min 30s, sys: 2.16 ms, total: 7min 30s
```

Then, let's try to fit our model with min-max normalization

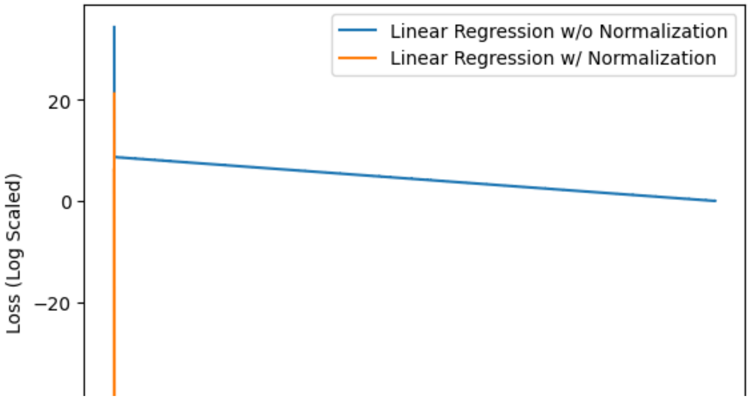
```
%%time
reg_norm = Linear_Regression(num_iter = 10000000, normalize=True)
reg_norm.fit(X,y)
print(f'\nNumber of total iterations: {len(reg_norm.loss)} \nBest Loss: {min(reg_norm.loss)}')
```

```
Start Training
-----------------------
Iteration: 0
Coef: [[2.44828618]
 [5.15287336]]
Loss: 1662372252.7693799
-----------------------
End Training (Early Stopped at iteration 1408)

Number of total iterations: 1408
Best Loss: 2.3697634028328865e-24
CPU times: user 107 ms, sys: 0 ns, total: 107 ms
Wall time: 138 ms
```

Now, let's compare the performance between these two models with/without normalization

```
plt.plot(np.log(reg.loss), label='Linear Regression w/o Normalization')
plt.plot(np.log(reg_norm.loss), label='Linear Regression w/ Normalization')
plt.xlabel("Number of Iterations")
plt.ylabel("Loss (Log Scaled)")
plt.legend()
plt.show()
```



Question 1: What conclusions can you draw from this experiment? Did normalization help? How and why?

- **Answer:** As is immediately evident from the graph, normaization results in drastically faster convergence (1408 vs 10000000 iterations)and far smaller loss (2.370e-24 vs 1.014 best loss). Normalization gives us equal scaling for all features, in this case limiting values to [0, 1). This benefits the efficiency of gradient descent since all weights can be updated at the same scale, all feature weights can have equal influence over the model, and we do not risk overshooting the optimal solution.

Experiment 2: Real-World Data

After you complete the first experiment, let's see how our model performs against real-world data.

The below dataset is taken from the Boston Housing dataset, where there are 13 features and 1 target variable.

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

```
url = 'https://archive.ics.uci.edu/ml/machine-learning-databases/housing/housing.data'
df = pd.read_csv(url, delimiter='\s+', header=None)
df.head()
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	0.00632	18.0	2.31	0	0.538	6.575	65.2	4.0900	1	296.0	15.3	396.90	4.98	24.0
1	0.02731	0.0	7.07	0	0.469	6.421	78.9	4.9671	2	242.0	17.8	396.90	9.14	21.6
2	0.02729	0.0	7.07	0	0.469	7.185	61.1	4.9671	2	242.0	17.8	392.83	4.03	34.7
3	0.03237	0.0	2.18	0	0.458	6.998	45.8	6.0622	3	222.0	18.7	394.63	2.94	33.4
4	0.06905	0.0	2.18	0	0.458	7.147	54.2	6.0622	3	222.0	18.7	396.90	5.33	36.2

```
X, y = np.array(df.drop(13, axis=1)), np.array(df[13])
```

```
X.min(axis = 0)
```

```
array([6.3200e-03, 0.0000e+00, 4.6000e-01, 0.0000e+00, 3.8500e-01,
       3.5610e+00, 2.9000e+00, 1.1296e+00, 1.0000e+00, 1.8700e+02,
       1.2600e+01, 3.2000e-01, 1.7300e+00])
```

Now, let's use the data to fit our model

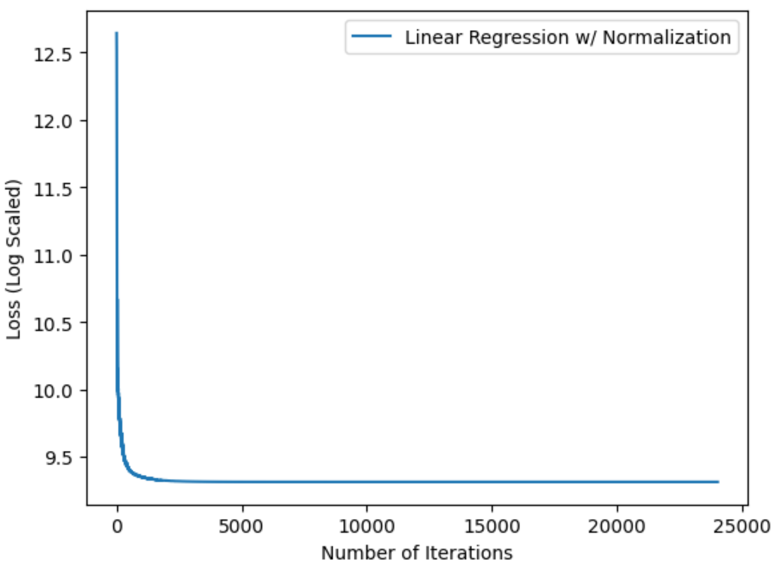
```
%%time
reg = Linear_Regression(num_iter=100000, normalize=True)
reg.fit(X,y)
print(f'\nNumber of total iterations: {len(reg.loss)} \nBest Loss: {min(reg.loss)}')
```

```
Start Training
-----------------------
Iteration: 0
Coef: [[-0.29240151]
 [-0.34226071]
 [ 0.98983001]
 [ 1.00229426]
 [-0.88864513]
 [-0.1306964 ]
 [ 0.40424979]
 [-0.36877215]
 [-0.17323282]
 [-0.5526915 ]
 [ 0.09058152]
 [-0.02994679]
 [-0.88912042]
 [-0.1640253 ]]
Loss: 309608.3171632303
-----------------------
End Training (Early Stopped at iteration 24023)

Number of total iterations: 24023
Best Loss: 11078.784577955423
CPU times: user 1.3 s, sys: 11 ms, total: 1.31 s
Wall time: 1.31 s
```

Let's visualize the loss curve of our model on this dataset

```
plt.plot(np.log(reg.loss), label='Linear Regression w/ Normalization')
plt.xlabel("Number of Iterations")
plt.ylabel("Loss (Log Scaled)")
plt.legend()
plt.show()
```



To verify our model, we can compare our model's performance with respect to the linear regression model implemented in scikit-learn (a.k.a. `sklearn`). Scikit-learn is a popular machine learning library in python that provides many classical machine learning algorithms for many different tasks (regression, classification, clustering, etc). It also contains utility functions for preprocessing, calculating metrics, etc.

If you implemented your model correctly, you should get a very similar output (difference < 1e-3) for RMSE (Root Mean Squared Error) compared to sklearn linear regressor's RMSE.

```
m, n = df.shape
X_norm = X.copy()

# TODO: normalize X using the procedure in your model implementation
norm_max = np.zeros((1, X_norm.shape[1]))
norm_min = np.zeros((1, X_norm.shape[1]))
for i in range(X_norm.shape[1]):
    norm_max[:, i] = X_norm[:, i].max()
    norm_min[:, i] = X_norm[:, i].min()

X_norm = (X_norm - norm_min) / (norm_max - norm_min)
```

```
# Let's build a model with sklearn
lr = LinearRegression()
lr.fit(X_norm,y)

#Compare Root Mean Squared Error.
print(f"Our Model's RMSE: {(sum((reg.predict(X).flatten() - y)**2)/m)**0.5}\
\nSklearn Model's RMSE: {(sum((lr.predict(X_norm) - y)**2)/m)**0.5}")
```

```
Our Model's RMSE: 4.679191295697375
Sklearn Model's RMSE: 4.679191295697285
```

Now, let's have some tweaks with our custom model. First, let's see if an interception (i.e. bias) really helps with our model's performance on the real-world data.

```
%%time
%%capture
reg_bias = Linear_Regression(num_iter=100000, normalize=True, intercept=True)
reg_no_bias = Linear_Regression(num_iter=100000, normalize=True, intercept=False)
reg_bias.fit(X,y)
reg_no_bias.fit(X,y)
```

```
CPU times: user 1.74 s, sys: 6.94 ms, total: 1.75 s
Wall time: 1.76 s
```

```
print(f"Our Model's RMSE with Interception: {(sum((reg_bias.predict(X).flatten() - y)**2)/m)**0.5}\
\nOur Model's RMSE without Interception: {(sum((reg_no_bias.predict(X).flatten() - y)**2)/m)**0.5}")
```

```
Our Model's RMSE with Interception: 4.679191295697386
Our Model's RMSE without Interception: 5.241354231005249
```

Question 2: What conclusions can you make here? Does the addition of an intercept make our model perform better?

- **Answer:** Yes, it did improve the performance of our model. By adding an intercept to our model, we gain more degrees of freedom to make our model more expressive. Adding an intercept allows us to perform translation away from the origin, whereas otherwise we would have to pass through the origin. Thus, by adding the intercept, we improve our ability to fit data that has bias like we see in this case.

Second, let's see if regularization can further help with decreasing our model's loss. Since regularization deals with the problem of overfitting, we need to check our model's performance on the "unseen" data. Here, we will split our data into two parts: `training set` and `test set`, where our model will be fit with the training set, and the performance will be evaluated based on the test set.

```
X_train, X_test, y_train, y_test = train_test_split(X_norm, y, test_size=0.33, random_state=42)
m, n = X_test.shape
```

```
%%time
%%capture
reg = Linear_Regression(num_iter=100000, normalize=True)
reg.fit(X_train, y_train)
# Feel free to tune the lambda hyperparameter for better performance when penalty (regularization) is applied
reg_l1 = Linear_Regression(num_iter=100000, normalize=True, penalty='l1')
reg_l1.fit(X_train, y_train)
reg_l2 = Linear_Regression(num_iter=100000, normalize=True, penalty='l2')
reg_l2.fit(X_train, y_train)
```

```
CPU times: user 1.75 s, sys: 11.4 ms, total: 1.76 s
Wall time: 1.77 s
```

```
print(f"Our Model's RMSE: {(sum((reg.predict(X_test).flatten() - y_test)**2)/m)**0.5}")
print(f"Our L1 Regularized Model's RMSE: {(sum((reg_l1.predict(X_test).flatten() - y_test)**2)/m)**0.5}")
print(f"Our L2 Regularized Model's RMSE: {(sum((reg_l2.predict(X_test).flatten() - y_test)**2)/m)**0.5}")
```

```
Our Model's RMSE: 4.552364553886074
Our L1 Regularized Model's RMSE: 4.551834530477728
Our L2 Regularized Model's RMSE: 4.6906293175959455
```

Question 3: What conclusions can you make here? Does the addition of a regularization make our model perform better on the test set? Why does the addition of it make our model perform better/worse?

- **Answer:** With L2 regularization, less important weights have their magnitude reduced, but not set to zero like in L1 regularization. Because these weights may not be necessary to make our prediction, inclusion of these weights may hinder performance. This could explain the small increase in RMSE for L2 compared to L1 and our non-regularized model performance.

- L1 regularization seems to have made no significant difference compared to our default model. This could be because there aren't too many features to result in an overfitting issue. If the default model is already adequetly generalizable, L1 would not make a difference.

Finally, let's see the role of an adaptive learning rate. Let's see our model's performance when adaptive learning rate is disabled.

```
m, n = X.shape
```

```
%%time
%%capture
reg = Linear_Regression(num_iter=100000, normalize=True)
reg.fit(X, y)
reg_alt = Linear_Regression(num_iter=100000, normalize=True, adaptive=False)
reg_alt.fit(X, y)
```

```
CPU times: user 7.1 s, sys: 7.39 ms, total: 7.11 s
Wall time: 10.1 s
```

```
print(f"Our Model's RMSE with Adaptive LR: {(sum((reg.predict(X).flatten() - y)**2)/m)**0.5}\
\nOur Model's RMSE without Adaptive LR: {(sum((reg_alt.predict(X).flatten() - y)**2)/m)**0.5}")
```

```
Our Model's RMSE with Adaptive LR: 4.679191295697379
Our Model's RMSE without Adaptive LR: 4.733064937414148
```

Question 4: What conclusions can you make here? Does the addition of an adaptive learning rate make our model perform better? What are your reasonings here?

- **Answer:** It's possible we may have already found a global minima for this dataset using our model without adaptive LR which would explain why the difference between the two models is quite small. However, adaptive LR does give us more granularity in our gradient update which could explain the marginally better performance.