Homework Assignment 2

CSE 251A: ML – Learning algorithms

Due: April 29th, 2023, 9:30am (Pacific Time)

Instructions: Please answer the questions below, attach your code in the document, and insert figures to create **a single PDF file**. You may search information online but you will need to write code/find solutions to answer the questions yourself.

Grade: ____ out of 100 points

1 (10 points) Nearest Neighbor

1.1 Nearest neighbor classification

You are given the points belonging to class-1 and class-2 as follows: Class 1 points: $(11, 11), (13, 11), (8, 10)$, Class 2 points: $(7, 11), (15, 9), (15, 7), (13, 5), (14, 4), (9, 3), (11, 3)$ What is the label of the sample $(14, 3)$ using the nearest neighbor classifier using L2 distance?

Distance from (14, 3) to class-1 points:

(11,11) = $\sqrt{(14-11)^2 + (3-11)^2} = 8.544$
(13,11) = $\sqrt{(14-13)^2 + (3-11)^2} = 8.062$
(8,10) = $\sqrt{(14-8)^2 + (3-10)^2} = 8.485$
(9,9) = $\sqrt{(14-9)^2 + (3-9)^2} = 6.708$
(7,7) = $\sqrt{(14-7)^2 + (3-7)^2} = 7.616$
(7,5) = $\sqrt{(14-7)^2 + (3-5)^2} = 8.246$
(16,3) = $\sqrt{(14-16)^2 + (3-3)^2} = 2.000$

Distance from (14,3) to class-2 points:

(7,11) = $\sqrt{(14-7)^2 + (3-11)^2} = 10.050$
(15,9) = $\sqrt{(14-15)^2 + (3-9)^2} = 6.083$
(15,7) = $\sqrt{(14-15)^2 + (3-7)^2} = 6.708$
(13,5) = $\sqrt{(14-13)^2 + (3-5)^2} = 2.236$
(14,4) = $\sqrt{(14-14)^2 + (3-4)^2} = 1.000$
(9,3) = $\sqrt{(14-9)^2 + (3-3)^2} = 5.000$
(11,3) = $\sqrt{((14-11)^2 + (3-3)^2)} = 3.000$

The closest point to (14, 3) is (14, 4) from class-2 with a distance of 1.000. (14, 3) would be classified as class-2.

1.2 Error rate of nearest neighbor

Give an example of a binary classification dataset with 3 points (x, y) for which the 1-NN classifier does not have zero training error (that is, it makes mistakes on the training set). You should plot the three points and show where the error is.

Class 1 points: (0,0), (2,2) Class 2 points: (1,0)

The 1-NN binary classifier would classify the (1,0) point as being under class 1 since it is closest to (0,0). The actual label for (1,0) is actually under class 2 which means that the 1-NN classifier does not have zero training error.

2 (10 points) Gradient Descent - Linear Regression

Consider house rent prediction problem where you are supposed to predict price of a house based on just its area. Suppose you have n samples with their respective areas, $x^{(1)}, x^{(2)}, \ldots, x^{(n)}$, their true house rents $y^{(1)}, y^{(2)}, \ldots, y^{(n)}$. Let's say, you train a linear regressor that predicts $f\left(x^{(i)}\right) = \theta_0 + \theta_1 x^{(i)}$. The parameters θ_0 and θ_1 are scalars and are learned by minimizing mean-squared-error loss through gradient descent with a learning rate α. Answer the following questions.

1. Express the loss function(L) in terms of $x^{(i)}, y^{(i)}, n, \theta_0, \theta_1$.
2. Compute $\frac{\partial L}{\partial \theta_0}$
3. Compute $\frac{\partial L}{\partial \theta_1}$
4. Write update rules for θ_0 and θ_1

3 (10 points) Gradient Descent - Linear Regression with L1 Regularization

Consider the same house rent prediction problem where you are supposed to predict price of a house based on just its area. Suppose you have n samples with their respective areas, $x^{(1)}, x^{(2)}, \ldots, x^{(n)}$, their true house rents $y^{(1)}, y^{(2)}, \ldots, y^{(n)}$. Let's say, you train a linear regressor that predicts $f(x^{(i)}) = \theta_0 + \theta_1 x^{(i)}$. The parameters θ_0 and θ_1 are scalars and are learned by minimizing mean-squared-error loss with L1-regularization through gradient descent with a learning rate α and the regularization strength constant λ. Answer the following questions.

1. Express the loss function(L) in terms of $x^{(i)}, y^{(i)}, n, \theta_0, \theta_1, \lambda$.
2. Compute $\frac{\partial L}{\partial \theta_0}$
3. Compute $\frac{\partial L}{\partial \theta_1}$
4. Write update rules for θ_0 and θ_1

2. $L = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$

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

2. $f = h^2 \qquad h = y_i - (\theta_0 + \theta_1 x_i)$

$f' = 2h \qquad \frac{h'}{\partial \theta_0} = -1$

$2(y_i - \theta_0 + \theta_1 x_i) \cdot (-1)$

$\frac{\partial L}{\partial \theta_0} = -\frac{2}{n}\sum_{i=1}^{n}(y_i - (\theta_0 + \theta_1 x_i))$

3. $f = h^2 \qquad h = y_i - (\theta_0 + \theta_1 x_i)$

$\frac{f'}{\partial L} = 2h \qquad \frac{h'}{\partial \theta_1} = -x_i$

$2(y_i - \theta_0 + \theta_1 x_i) \cdot -x_i$

$\frac{\partial L}{\partial \theta_1} = -\frac{2}{n}\sum_{i=1}^{n} x_i (y_i - (\theta_0 + \theta_1 x_i))$

4. $\theta_0^{T+1} = \theta_0^{T} - \alpha\left(-\frac{2}{n}\sum_{i=1}^{n}(y_i - (\theta_0 + \theta_1 x_i))\right)$

$\theta_1^{T+1} = \theta_1^{T} - \alpha\left(-\frac{2}{n}\sum_{i=1}^{n} x_i (y_i - (\theta_0 + \theta_1 x_i))\right)$

3. $$L = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

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

$$L = \frac{1}{n}\sum_{i=1}^{n}(y_i - (\theta_0 + \theta_1 x_i))^2 + \lambda|\theta_0| + \lambda|\theta_1|$$

2. $$\frac{\partial L}{\partial \theta_0} n = \sum_{i=1}^{n} -2(y_i - \theta_0 + \theta_1 x_i) + \lambda|\theta_0|$$

$$\frac{\partial L}{\partial \theta_0} = \begin{cases} \frac{-2}{n}\sum_{i=1}^{n}(y_i - (\theta_0 + \theta_1 x_i)) + \lambda & \theta_0 > 0 \\ -\frac{2}{n}\sum_{i=1}^{n}(y_i - (\theta_0 + \theta_1 x_i)) & \theta_0 = 0 \\ -\frac{2}{n}\sum_{i=1}^{n}(y_i - (\theta_0 + \theta_1 x_i)) - \lambda & \theta_0 < 0 \end{cases}$$

3. $$\frac{\partial L}{\partial \theta_1} n = \sum_{i=1}^{n} 2x_i(y_i - \theta_0 + \theta_1 x_i) + \lambda|\theta_1|$$

$$\frac{\partial L}{\partial \theta_1} = \begin{cases} -\frac{2}{n}\sum_{i=1}^{n} x_i(y_i - (\theta_0 + \theta_1 x_i)) + \lambda & \theta_1 > 0 \\ -\frac{2}{n}\sum_{i=1}^{n} x_i(y_i - (\theta_0 + \theta_1 x_i)) & \theta_1 = 0 \\ -\frac{2}{n}\sum_{i=1}^{n} x_i(y_i - (\theta_0 + \theta_1 x_i)) - \lambda & \theta_1 < 0 \end{cases}$$

4.
$$\theta_0^{t+1} = \begin{cases} \theta_0^t - \alpha\left(\frac{-2}{n}\sum_{i=1}^{n}(y_i - (\theta_0 + \theta_1 x_i)) + \lambda\right) & \theta_0 > 0 \\ \theta_0^t - \alpha\left(\frac{-2}{n}\sum_{i=1}^{n}(y_i - (\theta_0 + \theta_1 x_i))\right) & \theta_0 = 0 \\ \theta_0^t - \alpha\left(\frac{-2}{n}\sum_{i=1}^{n}(y_i - (\theta_0 + \theta_1 x_i)) - \lambda\right) & \theta_0 < 0 \end{cases}$$

$$\theta_1^{t+1} = \begin{cases} \theta_1^t - \alpha\left(\frac{-2}{n}\sum_{i=1}^{n} x_i(y_i - (\theta_0 + \theta_1 x_i)) + \lambda\right) & \theta_1 > 0 \\ \theta_1^t - \alpha\left(\frac{-2}{n}\sum_{i=1}^{n} x_i(y_i - (\theta_0 + \theta_1 x_i))\right) & \theta_1 = 0 \\ \theta_1^t - \alpha\left(\frac{-2}{n}\sum_{i=1}^{n} x_i(y_i - (\theta_0 + \theta_1 x_i)) - \lambda\right) & \theta_1 < 0 \end{cases}$$

Hint:

$$\frac{d|w|}{dw} = \begin{cases} 1 & w > 0 \\ \text{undefined} & w = 0 \\ -1 & w < 0 \end{cases}$$

4 (10 points) Gradient Descent - Linear Regression with L2 Regularization

Consider the same house rent prediction problem where you are supposed to predict price of a house based on just its area. Suppose you have n samples with their respective areas, $x^{(1)}, x^{(2)}, \ldots, x^{(n)}$, their true house rents $y^{(1)}, y^{(2)}, \ldots, y^{(n)}$. Let's say, you train a linear regressor that predicts $f(x^{(i)}) = \theta_0 + \theta_1 x^{(i)}$. The parameters θ_0 and θ_1 are scalars and are learned by minimizing mean-squared-error loss with L2-regularization through gradient descent with a learning rate α and the regularization strength constant λ. Answer the following questions.

1. Express the loss function(L) in terms of $x^{(i)}, y^{(i)}, n, \theta_0, \theta_1, \lambda$.
2. Compute $\frac{\partial L}{\partial \theta_0}$
3. Compute $\frac{\partial L}{\partial \theta_1}$
4. Write update rules for θ_0 and θ_1

5 (50 points) Implementing a Linear Regression Model from Scratch

Now, you will implement a linear regression model from scratch. We have provided a skeleton code file (i.e. LinearRegression.py) for you to implement the algorithm as well as a notebook file (i.e. Linear_Regression.ipynb) for you to conduct experiment and answer relevant questions. Libraries such as numpy and pandas may be used for auxiliary tasks (such as matrix multiplication, matrix inversion, and so on), but not for the algorithms. That is, you can use numpy to implement your model, but cannot directly call libraries such as scikit-learn to get a linear regression model for your skeleton code. We will grade this question based on the three following criteria:

1. Your implementation in code. Please do not change the structure of our skeleton code.
2. Your model's performance (we check if your model behaves correctly based on the results from multiple experiments in the notebook file).
3. Your written answers for questions in the notebook file.

6 (10 points) Ridge regression

Consider the loss function for ridge regression (ignoring the intercept term):

$$L(\mathbf{w}) = \sum_{i=1}^{n} (y^{(i)} - w \times \mathbf{x^{(i)}})^2 + \lambda ||\mathbf{w}||^2$$

4. $L = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$

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

2. $f = h^2$ $\quad h = y_i - \theta_0 - \theta_1 x_i$

$f' = 2h$ $\quad h' = -1$

$$\frac{\partial L}{\partial \theta_0} = -\frac{2}{n}\sum_{i=1}^{n}(y_i - (\theta_0 + \theta_1 x_i)) + 2\lambda\theta_0$$

3. $f = h^2$ $\quad h = y_i - \theta_0 - \theta_1 x_i$

$f' = 2h$ $\quad h' = -x_i$

$$\frac{\partial L}{\partial \theta_1} = -\frac{2}{n}\sum_{i=1}^{n} x_i(y_i - (\theta_0 + \theta_1 x_i)) + 2\lambda\theta_1$$

4. $\theta_0^{t+1} = \theta_0^t - \alpha\left(-\frac{2}{n}\sum_{i=1}^{n}(y_i - (\theta_0 + \theta_1 x_i)) + 2\lambda\theta_0\right.$

$\theta_1^{t+1} = \theta_1^t - \alpha\left(-\frac{2}{n}\sum_{i=1}^{n} x_i(y_i - (\theta_0 + \theta_1 x_i)) + 2\lambda\theta_1\right.$

6. $L(w) = \sum_{i=1}^{n} (y_i - wx_i)^2 + \lambda \|w\|^2$

$f = h^2$ $\quad h = y_i - wx_i$

$f' = 2h$ $\quad h' = x_i$

$$\nabla L(w) = \sum_{i=1}^{n} 2x_i (y_i - wx_i) + 2\lambda w$$

$$w^{t+1} = w^t - \alpha \nabla L(w)$$

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.

$\hspace{30mm}$ $F(x)$ = Loss Function = MSE = $ \frac{1}{n}\sum^{n}_{i=1} (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

In [ ]:
```
import numpy as np
from tqdm import tqdm

class Linear_Regression():
    def __init__(self, alpha = 1e-3 , num_iter = 10000, early_stop = 1000,
        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:
            ones = np.ones((self.X.shape[0], 1))
            self.X = np.hstack((self.X, ones))
        ################ 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
        # 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

        predict = np.dot(self.X, self.coef) + self.intercept

        grad = coef * np.dot(self.X.T, self.y - predict)

        ################ 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 = np.dot(self.X, self.coef) + self.intercept
            grad = self.gradient()
            # calculate the new coefficients after incorporating the gradient
            temp_coef = self.coef - (self.alpha * grad)
            current_y_hat = np.dot(self.X, temp_coef) + self.intercept
            ############### 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_
            current_error = np.sum(np.square(self.y - current_y_hat)) + current_reg

            ############### 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 = self.alpha * 1.3 if self.adaptive else self.alpha
                self.coef = temp_coef
            else:
                self.alpha = self.alpha * 0.9 if self.adaptive else self.alpha
            ############### END TODO 8 ###############

            # 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[
        X = X_norm
        ############### START TODO 9 ###############
        # add bias (if necessary, same as TODO 2)
        if self.intercept:
            ones = np.ones((X.shape[0], 1))
            X = np.hstack((X, ones))

        # Find the model's predictions
        # Hint: Use matrix multiplication ('@' might come in handy here)

        y = np.dot(X, self.coef) + self.intercept
        return y
        ############### END TODO 9 ###############

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

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`

In [1]:
```
# 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
from LinearRegression import Linear_Regression
```

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.

In [2]:
```
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'
```

Out[2]:
```
'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)

In [3]:
```
%%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: [[-0.63423516]
 [-0.82443543]]
Loss: 930004617649606.5
----------------------
Iteration: 1000000
Coef: [[5.00957561]
 [2.64060077]]
Loss: 2031.2659326943508
----------------------
Iteration: 2000000
Coef: [[5.00620185]
 [4.880105  ]]
Loss: 852.5156210384578
----------------------
Iteration: 3000000
Coef: [[5.0040183 ]
 [6.33095318]]
Loss: 357.8026247004858
----------------------
Iteration: 4000000
Coef: [[5.00260382]
 [7.27087584]]
Loss: 150.17103655005417
----------------------
Iteration: 5000000
Coef: [[5.00168689]
 [7.87979836]]
Loss: 63.0268720112429
----------------------
Iteration: 6000000
Coef: [[5.00109252]
 [8.27428452]]
Loss: 26.45218885706526
----------------------
Iteration: 7000000
Coef: [[5.000708  ]
 [8.52984999]]
Loss: 11.102099354293244
----------------------
Iteration: 8000000
Coef: [[5.00045874]
 [8.69541637]]
Loss: 4.6595778664398315
----------------------
Iteration: 9000000
Coef: [[5.00029727]
 [8.8026775 ]]
Loss: 1.955644333529587
----------------------
End Training

Number of total iterations: 10000000
Best Loss: 0.8207658911565509
```

```
CPU times: user 3min 53s, sys: 971 ms, total: 3min 54s
Wall time: 3min 54s
```

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

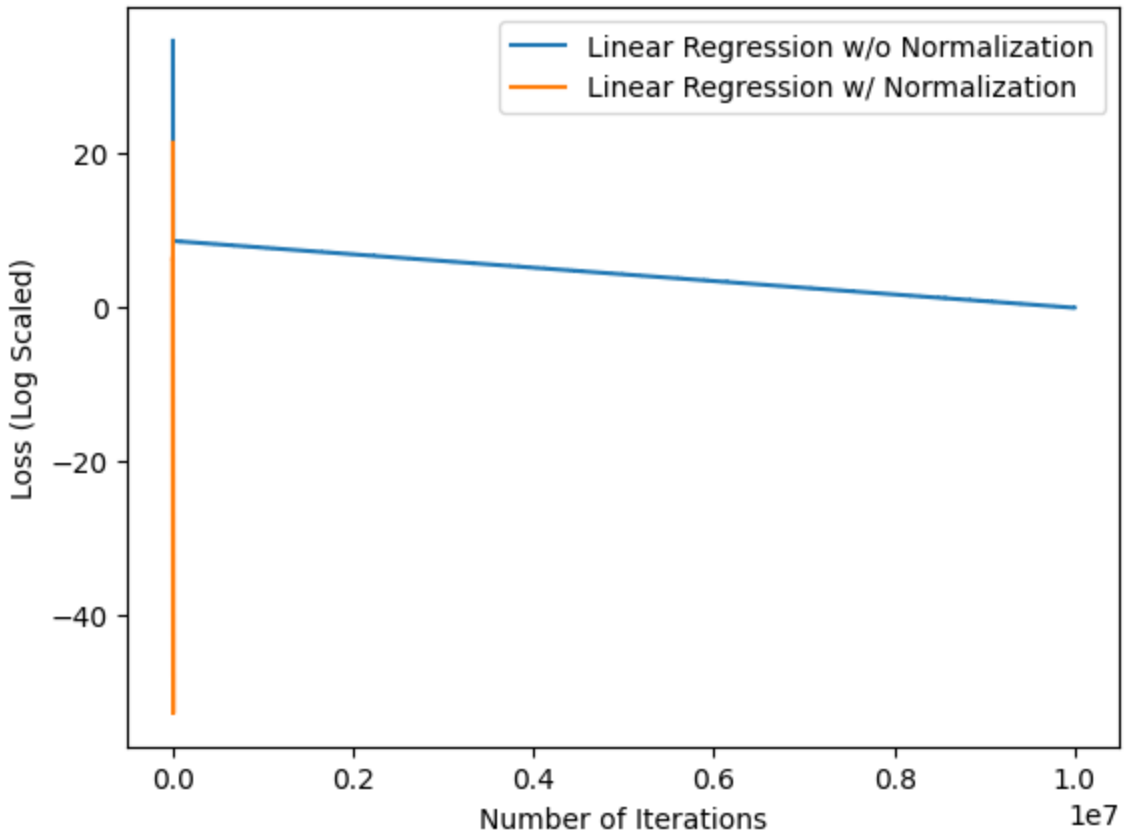
In [6]:
```
%%time
reg_norm = Linear_Regression(num_iter = 10000000, normalize=True)
reg_norm.fit(X,y)
print(f'\nNumber of total iterations: {len(reg_norm.loss)} \nBest Loss: {min(reg_no
```

```
Start Training
----------------------
Iteration: 0
Coef: [[2.39714861]
 [4.33865804]]
Loss: 1662220851.8496847
----------------------
End Training (Early Stopped at iteration 2276)

Number of total iterations: 2276
Best Loss: 1.182003936834197e-23
CPU times: user 64.3 ms, sys: 1.22 ms, total: 65.6 ms
Wall time: 64.7 ms
```

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

In [7]:
```
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:**

From this graph we can see that the linear regression with normalization performs much better than the linear regression without. As we can see from the graph above, the normalized method reduces the amount of iterations needed to train the model due to scaling the dataset so that the coefficients can be calculated more efficiently. This is because normalization standardizes the scale of the features to a more condensed scale. Datasets can be normalized by comparing the mean and standard deviation to each value in the dataset to produce a new value that is influenced by all 3 variables, the un-normalized variable, the mean, and the standard deviation. With this new normalized dataset, we help to smooth out the data and ensure that all values share the same scale. This can help remove convergence issues. In this case however, we are using min-max normalization to ensure that all values land between 0 and 1. This is a good normalization method if there are few weaker outliers present in the dataset but struggles when there are prevalent outliers in which case z-score normalization should be used.

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

In [8]:
```
url = 'https://archive.ics.uci.edu/ml/machine-learning-databases/housing/housing.da
df = pd.read_csv(url, delimiter='\s+', header=None)
df.head()
```

Out[8]:

	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

In [9]:
```
X, y = np.array(df.drop(13, axis=1)), np.array(df[13])
```

In [10]:
```
X.min(axis = 0)
```

Out[10]:
```
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

In [11]:
```
%%time
reg = Linear_Regression(num_iter=100000, normalize=True)
```

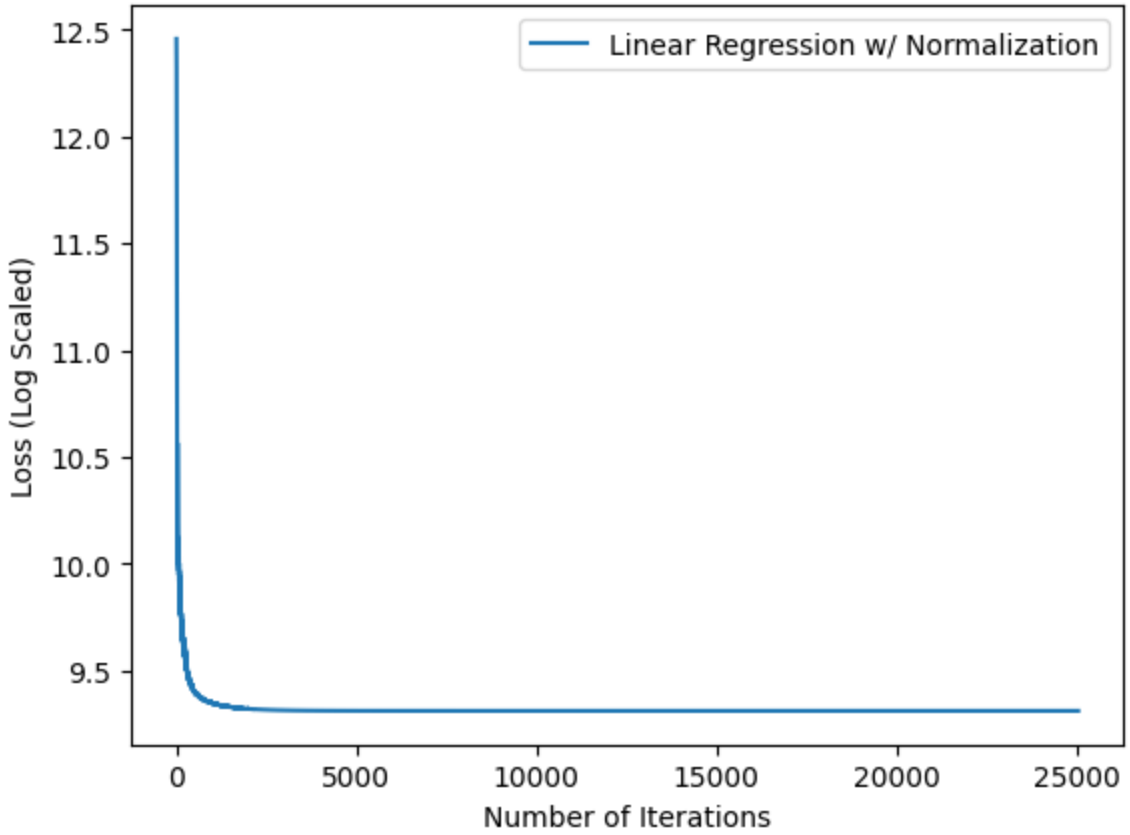
```
reg.fit(X,y)
print(f'\nNumber of total iterations: {len(reg.loss)} \nBest Loss: {min(reg.loss)}'
```

```
Start Training
-----------------------
Iteration: 0
Coef: [[-0.46442807]
 [ 0.96971022]
 [-0.53393649]
 [-0.53376951]
 [-0.63835771]
 [ 0.92693494]
 [-0.5186515 ]
 [ 0.13603426]
 [-0.29085871]
 [ 0.38600024]
 [ 0.76539771]
 [ 0.27021677]
 [-0.10520312]
 [ 0.32332255]]
Loss: 256640.5229761705
-----------------------
End Training (Early Stopped at iteration 25058)

Number of total iterations: 25058
Best Loss: 11078.784577955432
CPU times: user 686 ms, sys: 9.69 ms, total: 696 ms
Wall time: 696 ms
```

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

In [12]:
```
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.

In [13]:
```
m, n = df.shape
X_norm = X.copy()

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

X_norm = (X_norm - X_norm_min)/ (X_norm_max - X_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: 4.679191295697377
Sklearn Model's RMSE: 4.679191295697283
```

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.

In [14]:
```
%%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 1.03 s, sys: 8.5 ms, total: 1.04 s
Wall time: 1.04 s
```

In [15]:
```
print(f"Our Model's RMSE with Interception: {(sum((reg_bias.predict(X).flatten() -
\nOur Model's RMSE without Interception: {(sum((reg_no_bias.predict(X).flatten() -
```

```
Our Model's RMSE with Interception: 4.679191295697369
Our Model's RMSE without Interception: 5.241354231005265
```

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

- **Answer:**

We can conclude that the inclusion of an intercept does help the model perform more efficiently. The inclusion of the intercept also allows for the coefficients to associate with a one-unit increase in the corresponding independent variable. Having the intercept allows our model to capture all linear patterns while a model with no intercept can only capture the patterns that pass through the origin. The intercept is useful when some values may be 0 and is dependent on the model. In this case, since crime can hypothetically equal zero, we should include the intercept to account for that.

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.

In [16]:
```
X_train, X_test, y_train, y_test = train_test_split(X_norm, y, test_size=0.33, rand
m, n = X_test.shape
```

In [17]:
```
%%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 (
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 1.25 s, sys: 10.2 ms, total: 1.26 s
Wall time: 1.26 s
```

In [18]:
```
print(f"Our Model's RMSE: {(sum((reg.predict(X_test).flatten() - y_test)**2)/m)**0.
print(f"Our L1 Regularized Model's RMSE: {(sum((reg_l1.predict(X_test).flatten() -
print(f"Our L2 Regularized Model's RMSE: {(sum((reg_l2.predict(X_test).flatten() -
```

```
Our Model's RMSE: 4.552364547809143
Our L1 Regularized Model's RMSE: 4.552225681287769
Our L2 Regularized Model's RMSE: 4.571225487283601
```

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:**

We can conclude that L1 regularization is the best method to ensure that the model is not overfitted. L1 is used when there are many features that are not expected to contribute much to the outcome while l2 is used when many features are expected to contribute to the outcome. In this case we see L1 perform better than L2 by .02 and the unregularized RMSE by .0001 which indicates that L1 is probably the best choice to combat overfitting. We can also conclude that there are features that do not contribute to the outcome of the predicted values meaning L2 regularization may not be the best choice for combating overfitting. L1 has the chance to set features to zero, removing them from the model, while L2 tends to converge all features towards to but not exactly zero.

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

In [19]:
```
m, n = X.shape
```

In [20]:
```
%%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 3.24 s, sys: 19.6 ms, total: 3.26 s
Wall time: 3.26 s
```

In [21]:
```
print(f"Our Model's RMSE with Adaptive LR: {(sum((reg.predict(X).flatten() - y)**2)
\nOur Model's RMSE without Adaptive LR: {(sum((reg_alt.predict(X).flatten() - y)**2
```

```
Our Model's RMSE with Adaptive LR: 4.6791912956973825
Our Model's RMSE without Adaptive LR: 4.734184583969607
```

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:**

We can conclude that the model performs better with an adaptive LR. We use the adaptive LR so that the model can more efficiently converge and adapt to changes in the data or parameters. This learning rate can learn from the previous iteration and update to better converge the data when the gradient is large or small. Larger gradients with an adaptive LR converge faster while smaller gradients converge slower. Cases where an adaptive LR is preferred is when the dataset cointains outliers, when the data is noisy and unfiltered, or when the model is complex with a large amount of parameters. In this case, while 13 may not be a lot compared to deep neural networks, it is still a fairly large dataset to run linear regression on in which case we should use an adaptive LR to handle drastic changes in the gradient.

where $(\mathbf{x^{(1)}}, y(1)), \ldots, (\mathbf{x^{(n)}}, y^{(n)}) \in R^d \times R$ are n data points and labels; and $\mathbf{w} \in R^d$. There is a closed-form equation for the optimal $\mathbf{w}$, but suppose that we decide instead to minimize the function using local search.

1. What is $\nabla L(\mathbf{w})$?
2. Write down the update step for gradient descent.

7 What to submit?

A single **PDF** file that includes:

1. Answers for Q1-Q4 and Q6
2. LinearRegression.py file for your linear regression model implementation. If you are unable to directly convert the .py file into a PDF, you can copy and paste all the code from that .py file into the Linear_Regression.ipynb within a single cell.
3. The Linear_Regression.ipynb file for your experiments and relevant answers. **Please do not clear the outputs of each code cell for submission as we need to check your model's outputs as well.**