Module 1: Linear Regression

To write legible answers you will need to be familiar with both Markdown and Latex

Before you turn this problem in, make sure everything runs as expected. First, restart the kernel (in the menubar, select Kernel→→Restart) and then run all cells (in the menubar, select Cell→→Run All).

Make sure you fill in any place that says "YOUR CODE HERE" or "YOUR ANSWER HERE", as well as your name below:

In [ ]:
```
NAME = ""
STUDENT_ID = ""
```

Question 1 - Linear Regression

In this question, you will be implementing the linear regression algorithm from scratch in Python. Linear regression aims to map feature vectors to a continuous value in the range $[-\infty, +\infty]$ by linearly combining the feature values.

Model Representation

Data is represented as a dataframe or a feature matrix. Let our feature matrix be X whose dimensions are $n \times m$, θ be a weight matrix of dimensions $m \times 1$, the bias vector b a column vector of dimension $m \times 1$. Using these we can predict $\hat{Y}$ by the following relationship:

$$\hat{Y} = X\theta + b$$

Data: Facebook posts metrics

This data contains features describing posts from a cosmetic brand's Facebook page. The authors use the following features:

- Category,
- Page total likes: Number of people who have liked the company's page),
- Type: Type of content (Link, Photo, Status, Video),
- Post month: Month the post was published (January, February, March, ..., December),
- Post hour: Hour the post was published (0, 1, 2, 3, 4, ..., 23) ,
- Post weekday: Weekday the post was published (Sunday, Monday, ...,

Saturday) ,

- Paid: If the company paid to Facebook for advertising (yes, no)

to model:

'Lifetime Post Total Reach', 'Lifetime Post Total Impressions', 'Lifetime Engaged Users', 'Lifetime Post Consumers', 'Lifetime Post Consumptions', 'Lifetime Post Impressions by people who have liked your Page', 'Lifetime Post reach by people who like your Page', 'Lifetime People who have liked your Page and engaged with your post', 'comment', 'like', 'share', 'Total Interactions'.

There are many possible features we could try to model, but we will focus on 'Total Interactions'. Our feature space will include: Category, Page total likes, Post month, Post hour, Post weekday, and Paid. We drop "Type" simply to avoid preprocessing.

You can read more about the dataset here.

Downloading the data

In [ ]:
```
!wget http://archive.ics.uci.edu/ml/machine-learning-databases/00368/Facebook_metri
import zipfile
with zipfile.ZipFile('./Facebook_metrics.zip', 'r') as zip_ref:
    zip_ref.extractall('./')
```

Reading in data

In [ ]:
```
import pandas as pd
import numpy as np
np.random.seed(144)
'''
Shuffles the data in place
'''
def shuffle_data(data):
    np.random.shuffle(data)

# Read in the data
lr_dataframe = pd.read_csv('dataset_Facebook.csv',sep=';')
lr_dataframe.dropna(inplace=True)
columns_to_drop = ['Type','Lifetime Post Total Reach', 'Lifetime Post Total Impress
       'Lifetime Engaged Users', 'Lifetime Post Consumers',
       'Lifetime Post Consumptions',
       'Lifetime Post Impressions by people who have liked your Page',
       'Lifetime Post reach by people who like your Page',
       'Lifetime People who have liked your Page and engaged with your post',
       'comment', 'like', 'share']
lr_dataframe.drop(columns=columns_to_drop,inplace=True)

# Normalizing all remaining columns
def normalize_col(col):
    return (col - col.min())/(col.max() - col.min())

lr_dataframe = lr_dataframe.apply(normalize_col)

# Get entries as a numpy array
lr_data = lr_dataframe.values[:, :]

# Shuffle once for reproducibility
shuffle_data(lr_data)

lr_dataframe.head()
```

a) Splitting data in X and Y

In this part we will write functions to split our data into a feature matrix X (augmented by a column of 1's to account for the bias term) and a column vector we wish to predict Y and further split X and Y into X_train, X_test, y_train and y_test for training and testing.

i) Split the dataset into X and Y. In order to vectorize our calculations, we utilize the bias trick. This simply means we need to append ones to our X matrix.

ii) Split X and Y into the training and test sets using the provided percentage split (default is 80% training and 20% test).

In [ ]:
```
'''
Combines one column of all ones and the matrix X to account for the bias term
(setting x_0 = 1) - [Hint: you may want to use np.hstack()]
Takes input matrix X
Returns the augmented input
'''
def bias_trick(X):
    # YOUR CODE HERE

'''
Separates feature vectors and targets
Takes raw data
Returns X as the matrix of feature vectors and Y as the vector of targets
'''
def separate_data(data):
    # Split into X (remember to use bias trick) and Y
    # YOUR CODE HERE

'''
Takes raw data in and splits the data into
X_train, y_train, X_test, y_test
Returns X_train, y_train, X_test, y_test
'''
def train_test_split(data, train_size=.80):
    # YOUR CODE HERE
    return
```

b) Training and testing our model

Refer to the following derivation of the gradient when implementing linear regression and gradient descent below.

For this question, we'll use and implement the Mean Squared Error (MSE), and gradient descent algorithm. Suppose our dataset consists of n records, each with d features:

$$X = \begin{bmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,d} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,d} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n,1} & x_{n,2} & \cdots & x_{n,d} \end{bmatrix}$$

One way to include a bias is to augment X with a column of ones:

$$X = \begin{bmatrix} 1 & x_{1,1} & x_{1,2} & \cdots & x_{1,d} \\ 1 & x_{2,1} & x_{2,2} & \cdots & x_{2,d} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n,1} & x_{n,2} & \cdots & x_{n,d} \end{bmatrix}$$

We also have n labels corresponding to the correct classification of each of the above records, $y = [y_1, y_2, \cdots, y_n]^T$, i.e.:

$$y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$$

We will try to find the optimal parameter values $\theta = [\theta_0, \theta_1, \cdots, \theta_d]^T$ of our linear regression model, where θ_0 is the bias weight. To simplify our notation, let

$$\hat{y} = X\theta = \begin{bmatrix} X_{1,0}\theta_0 + X_{1,1}\theta_1 + \cdots + X_{1,d}\theta_d \\ X_{2,0}\theta_0 + X_{2,1}\theta_1 + \cdots + X_{2,d}\theta_d \\ \vdots \\ X_{n,0}\theta_0 + X_{n,1}\theta_1 + \cdots + X_{n,d}\theta_d \end{bmatrix} = \begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \\ \vdots \\ \hat{y}_n \end{bmatrix}$$

We seek θ such that the MSE is minimized (the 1/2 factor makes the derivation easier). Let the MSE be a function of θ, $J(\theta)$:

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

Since the above is a convex function, it has a unique minimum value. Taking the derivative with respect to θ_i, we get:

$$\frac{\partial}{\partial \theta_j} J(\theta) = \frac{1}{2n} \sum_{i=1}^{n} \frac{\partial}{\partial \theta_j} (\hat{y}_i - y_i)^2$$

$$= \frac{1}{n} \sum_{i=1}^{n} (\hat{y}_i - y_i) \frac{\partial}{\partial \theta_j} (\hat{y}_i)$$

Recall the chain rule from calculus, and that each $\hat{y}_i$ is a function of the θ_i, so the above becomes:

$$\frac{\partial}{\partial \theta_j} J(\theta) = \frac{1}{n} \sum_{i=1}^{n} (\hat{y}_i - y_i) x_{i,j}$$

YOU SHOULD:

i) Get training and testing set by calling train_test_split()

ii) Define a weight (θ) vector

ii) Implement Gradient Descent using the information above

iii) Record the Sum Squared Error for training and test data

iv) Return the weight matrix, train errors and test errors

v) Plot the training and test errors and comment on the plot.

In [ ]:
```
'''
Takes the target values and predicted values and calculates the squared error
between them
'''
def mse(y_pred, y_true):
    # YOUR CODE HERE
    return

'''
Implementation of the derivative of MSE.
Returns a vector of derivations of loss with respect to each of the dimensions
[\partial loss / \partial \theta_j]
'''
def mse_derivative(X,y,theta):
    # YOUR CODE HERE
    return

'''
Gradient descent step.
Takes X, y, theta vector, and alpha.
Returns an updated theta vector.
'''
def gradient_descent_step(X,y, theta, alpha):
    # YOUR CODE HERE
    return theta
```

In [ ]:
```
# Carry out training task
# YOUR CODE HERE

# Plot the training error and test error for different epochs (iterations of the
# algorithm). Your plot be MSE error vs epochs.
# YOUR CODE HERE
    test_errors = []

    # Define theta
    theta = np.zeros((X_train.shape[1]))

    # Carry out training loop
    for i in range(num_epochs):
        train_error = # YOUR CODE HERE
        train_errors.append(train_error)

        test_error = # YOUR CODE HERE
        test_errors.append(test_error)

        # Do gradient descent on the training set
        theta = # YOUR CODE HERE
    return theta, train_errors, test_errors
```

Data may not follow a linear relationship from the independent variable X to the dependent variable y. Fitting a linear model to this would be inaccurate and yield a high loss.

If we want to model an order d polynomial relationship between X and y we can augment our initial linear model where instead of having:

$$y_i = \theta_0 + \theta_1 x_i$$

We have:

$$y_i = \theta_0 + \theta_1 x_i + \theta_2 x_i^2 + \cdots + \theta_d x_i^d$$

We can use the same linear regression algorithm we if we first augment X and add extra columns (or dimensions).

$$\mathbf{X} = \begin{bmatrix} x_1 & x_1^2 & \cdots & x_1^d \\ x_2 & x_2^2 & \cdots & x_2^d \\ \vdots & \vdots & \ddots & \vdots \\ x_n & x_n^2 & \cdots & x_n^d \end{bmatrix}$$

Then our new higher order $\hat{y}$ is computed same as before.

$$\hat{y} = X\theta = \begin{bmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^d \\ 1 & x_2 & x_2^2 & \cdots & x_2^d \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^d \end{bmatrix} \begin{bmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_d \end{bmatrix} = \begin{bmatrix} \theta_0 + \theta_1 x_1 + \theta_2 x_1^2 + \cdots + \theta_d x_1^d \\ \theta_0 + \theta_1 x_2 + \theta_2 x_2^2 + \cdots + \theta_d x_2^d \\ \vdots \\ \theta_0 + \theta_1 x_n + \theta_2 x_n^2 + \cdots + \theta_d x_n^d \end{bmatrix} = \begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \\ \vdots \\ \hat{y}_n \end{bmatrix}$$

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

def normalize_data(data):
    return (data - np.min(data))/(np.max(data) - np.min(data))

np.random.seed(33)
x = np.random.uniform(-10, 10, 1000)
poly_coeffs = np.random.uniform(-1,1, size=(4,1))
y = poly_coeffs[0] + poly_coeffs[1]*x + poly_coeffs[2]*(x ** 2) + poly_coeffs[3]*(x

x2 = np.random.uniform(-10, 10, 1000)
poly_coeffs = np.random.uniform(-1,1, size=(3,1))
y2 = poly_coeffs[0] - 2000 + poly_coeffs[1]*x2 + 50*poly_coeffs[2]*(x2 ** 2)  + np.

x = np.concatenate([x,x2])
y = np.concatenate([y,y2])
x = normalize_data(x)
y = normalize_data(y)

plt.scatter(x,y, s=10)
plt.show()

poly_data = np.hstack((x.reshape(-1,1),y.reshape(-1,1)))
np.random.shuffle(poly_data)
x = poly_data[:,0]
y = poly_data[:,1]
```

In [ ]:
```
import numpy as np
from sklearn.linear_model import LinearRegression
```

In [ ]:
```
reg = LinearRegression().fit(x.reshape(-1,1), y)
```

In [ ]:
```
def compute_line_from_regr(X_data, y_data, regr):
    l_bound = np.min(X_data)
    r_bound = np.max(X_data)
    return [l_bound, r_bound], [l_bound * regr.coef_ + regr.intercept_, r_bound * r

plt.scatter(x,y, s=10)
line_x, line_y = compute_line_from_regr(x.reshape(-1,1),y,reg)
plt.plot(line_x, line_y, color='r')
plt.show()
```

As we see above, this data doesn't follow a linear relationship, it follows some complex polynomial. In the next section you'll try to fit a higher degree polynomial to it.

Weight regularization

When we try to fit a d-order polynomial to our data, we could end up overfitting. This happens when you try to fit a higher dimensional curve than what the distribution of our data actually exhibits. We can mitigate this by choosing an order d that matches your data closely, but often times this is not directly apparant in noisy data. Another method to avoid overfitting is **regularizing**, where you modify your loss to keep weights small which flattens our polynomial. This helps us avoid learning polynomials that are too complex for our data.

To add regularization we modify our original loss function J to include our regularizing term and a new hyperparameter that we tune λ. This controls the amount of regularizing we impose on the weights. We use the loss computed from the validation set to tweak this parameter.

$$J(\theta) = \frac{1}{2n}\sum_{i=1}^{n}(h^{(i)} - y^{(i)})^2 + \lambda\sum_{j=1}^{d}\theta_j^2$$

Our gradient computation also changes:

$$\frac{\partial}{\partial\theta_j}J(\theta) = \frac{1}{n}\sum_{i=1}^{n}(h^{(i)} - y^{(i)})x_{i,j} + 2\lambda\theta_j$$

We apply this gradient the same way as before in our gradient descent algorithm:

$$\theta_j = \theta_j - \alpha\frac{\partial}{\partial\theta_j}J(\theta)$$

In [ ]:
```
'''
Takes raw data in and splits the data into
X_train, y_train, X_test, y_test, X_val, y_val
Returns X_train, y_train, X_test, y_test, X_val, y_val
'''
def train_test_validation_split(data, test_size=.20, validation_size=.20):
    # YOUR CODE HERE
    return
```

In [ ]:

```
'''
Adds columns to your data up to the specified degree.
Ex: If degree=3, (x) -> (x, x^2, x^3)
'''
def add_polycols(X,degree):
    x_col = X[:,-1]

    for i in range(2, degree+1):
        X = np.hstack((X,(x_col**i).reshape(-1,1)))
    return X

'''
Takes the target values and predicted values and calculates the absolute error
between them
'''
def mse(y_pred, y_true):
    # YOUR CODE HERE
    # Feel free to use your implementation from Q1
    return

'''
Implementation of the derivative of MSE.
Returns a vector of derivations of loss with respect to each of the dimensions
[\partial loss / \partial \theta_i]
'''
def mse_derivative(X,y,theta):
    # YOUR CODE HERE
    # Feel free to use your implementation from Q1
    return

'''
Computes L2 norm from theta scaled by lambda.
Returns a scalar L2 norm.
'''
def l2norm(theta, lamb):
    # YOUR CODE HERE
    return

'''
Computes derivative of L2 norm scaled by lambda.
Returns a vector of derivative of L2 norms.
'''
def l2norm_derivative(theta, lamb):
    # YOUR CODE HERE
    # Note there is no regularization on the bias term.
    return

'''
Computes total cost (cost function + regularization term)
'''
def compute_cost(X, y, theta, lamb):
    # YOUR CODE HERE
    return

'''
Gradient descent step.
```

```
Takes X, y, theta vector, and alpha.
Returns an updated theta vector.
'''
def gradient_descent_step(X, y, theta, alpha, lamb):
    # YOUR CODE HERE
    # This differs from your Q1 implementation
    return


def polynomial_regression(data, degree, num_epochs=100000, alpha=1e-4, lamb=0):
    # Get training, testing, and validation sets by calling train_test_validation_s
    # YOUR CODE HERE

    # Record training and validation errors in lists
    train_errors = []
    val_errors = []

    # Add the appropriate amount of columns to each of your sets of data.
    X_train = add_polycols(X_train, degree)
    X_val = add_polycols(X_val, degree)
    X_test = add_polycols(X_test, degree)

    # Define theta
    theta = np.zeros((X_train.shape[1]))
    # Carry out training loop

    for i in range(num_epochs):
        train_error = # YOUR CODE HERE
        train_errors.append(train_error)

        val_error = # YOUR CODE HERE
        val_errors.append(val_error)

        # Do gradient descent on the training set
        theta = # YOUR CODE HERE

        # This prints the validation loss
        if i % (num_epochs//10) == 0:
            print(f'({i} epochs) Training loss: {train_error}, Validation loss: {va
    print(f'({i} epochs) Final training loss: {train_error}, Final validation loss:

    # Compute the testing loss
    test_error = # YOUR CODE HERE
    print(f'Final testing loss: {test_error}')
    return theta, train_errors, val_errors
```

As we mentioned above, we use the validation set's loss to tweak our hyperparameters. Please carry out the training task while monitoring the validation loss and varying the polynomial order d and regularization constant λ. Your answer should get close to minimizing the validation and testing losses.

In [ ]:
```
# degree d
polynomial_order =

# regularization constant lambda
regularization_param =

theta, train_errors, val_errors = polynomial_regression(poly_data, polynomial_order
```

In [ ]:
```
# Call plot_results() to see how your polynomial fits.
def plot_results(theta, X, Y):
    y_hat = sum([t*X**i for i,t in enumerate(theta)])
    plt.scatter(X, y_hat, s=10, color='r')
    plt.scatter(X, Y, s=10)
    plt.show()
```