ML Basics

Inclass Project 1 - MA4144

This project contains 12 tasks/questions to be completed.

After finishing project run the entire notebook once and **save the notebook as a pdf** (File menu -> Save and Export Notebook As -> PDF). You are **required to upload this PDF on moodle**.

Use this cell to use any include any imports

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
```

Section 1

In this section we will analyse a dataset to look into its statistical propeties. For this we will use the DiabetesTrain.csv dataset. Each row corresponds to a single patient. The first 8 columns correspond to the features of the patients that may help predict risk of diabetes. The outcome column is a binary column represting the risk of diabetes, outcome 1 : high risk of diabetes and outcome 0 little to no risk of diabetes.

Q1. Read the dataset into a pandas dataframe called diabetesData.

```
#TODO

diabetesData = pd.read_csv("DiabetesTrain.csv", usecols=range(9))
#diabetesData.sample(5)
diabetesData = diabetesData.dropna()
```

Q2. Let us define the following events.

A = Patient has BMI less than 25

B = Patient has Glucose level greater than 100

C = Patient has had more than 2 pregnancies

D = Patient has high risk of diabetes

Based on the above definitions determine the following probabilities. Pandas dataframe inbuilt functions such as count, group by, would be useful for this task.

```
#TODO

total_patients = len(diabetesData)
```

```
#P(A)
P_A = (diabetesData['BMI'] < 25).sum() / total_patients
print("P_A: ", P_A)

#P(B)
P_B = (diabetesData['Glucose'] >100).sum() / total_patients
print("P_B: ", P_B)

#P(C)
P_C = (diabetesData['Pregnancies'] >2).sum() / total_patients
print("P_C: ", P_C)

#P(D)
P_D = (diabetesData['Outcome'] == 1).sum() / total_patients
print("P_D: ", P_D)

#P(A, D)
P_A_D =((diabetesData['BMI'] < 25) & (diabetesData['Outcome'] ==
1)).sum()/total_patients
print("P_A_D: ", P_A_D)

#P(B, D)
P_B_D =((diabetesData['Glucose'] >100) & (diabetesData['Outcome'] ==
1)).sum()/total_patients
print("P_B_D: ", P_B_D)

#P(C, D)
P_C_D =((diabetesData['Pregnancies'] >2) & (diabetesData['Outcome'] ==
1)).sum()/total_patients
print("P_C_D: ", P_C_D)

#Indicate which one out of A, B, C contributes the most towards high
risk of diabetes.
#Assign one of 'A', 'B', 'C' to the following variable Q2, indicating
your answer.
#Hint: Compute the necessary conditional probabilities and then
compare.

#Conditional  probabilities (ex:- Probabilty of D Given A )
P_D_G_A = P_A_D / P_A if P_A > 0 else 0

P_D_G_B = P_B_D / P_B if P_B > 0 else 0

P_D_G_C = P_C_D / P_C if P_C > 0 else 0

Q2 = max(('A', P_D_G_A), ('B', P_D_G_B), ('C', P_D_G_C), key=lambda x:
x[1])[0]
print("Q2: ",Q2)
```

```
P_A:  0.14786967418546365
P_B:  0.7368421052631579
P_C:  0.5388471177944862
P_D:  0.37844611528822053
P_A_D:  0.007518796992481203
P_B_D:  0.3558897243107769
P_C_D:  0.2531328320802005
Q2:  B
```

Q3. Now we will compute the covariance and correlation matrices from scratch. For this do not use any inbuilt functions. Follow the steps outlined below. Each step is graded.

Step1: Convert the diabetesData dataframe into a 2-dimensional numpy array with the same number of rows and columns as in the dataframe. Name it diabetesX.

```
#TODO

diabetesX = diabetesData.to_numpy()
```

Step2: In diabetesX; center every column, by subtracting each column by the column mean and reassign it to diabetesX.

```
#TODO

#After centering
diabetesX = diabetesX-np.mean(diabetesX,axis=0)
```

Step3: Compute the covariance matrix. Use, matrix operations in numpy such as matrix multiplication, matrix transpose and don't forget to average. Assign it to the variable cov.

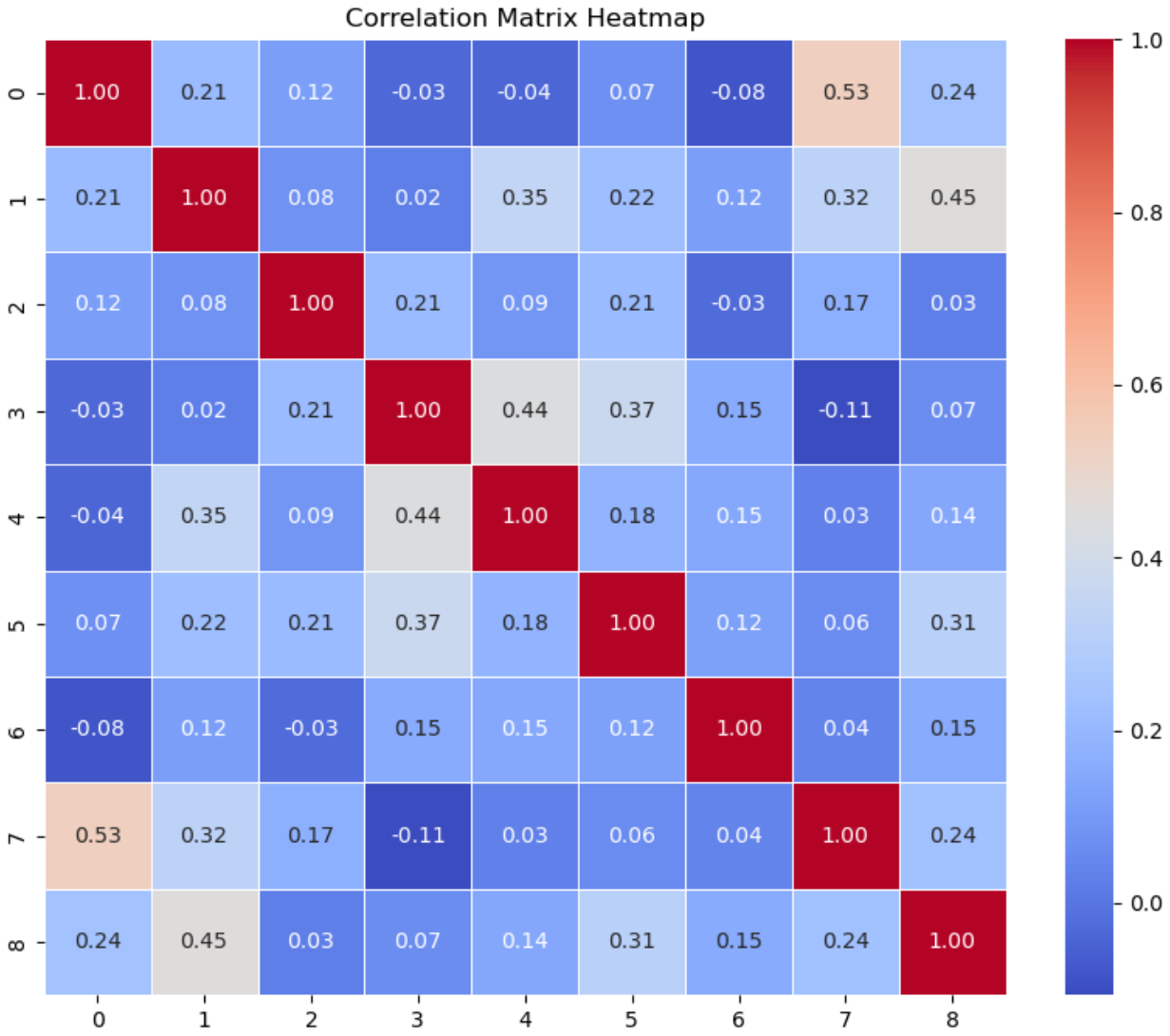
```
#TODO

cov = (diabetesX.T @ diabetesX)/(diabetesX.shape[0])
```

Step4: Compute the matrix varmat, whose (i, j) entry is $\sqrt{var(i)var(j)}$, where $var(k)$ is the variance of the kth column of the diabetesX matrix. The varinces can be extracted from the covariance matrix itself appropriately and varmat can be computed by appropriate matrix multiplication of a column matrix and a row matrix.

```
#TODO
# Extract variances
var = np.diag(cov)

# Compute square root of variances
sqrtVar = np.sqrt(var)
# Compute varmat using outer product
varmat = np.outer(sqrtVar, sqrtVar)
```

Step5: Use the cov matrix and varmat matrix appropriately to compute the correlational matrix corr. And then use seaborn (sns imported above) to plot an annotated heatmap of the correlation matrix.

```
#TODO
varmat[varmat == 0] = 1 #prevent division by zero
corr = cov / varmat

#Plot the heatmap.
plt.figure(figsize=(10, 8))
sns.heatmap(corr, annot=True, fmt=".2f", cmap="coolwarm",
linewidths=0.5)
plt.title("Correlation Matrix Heatmap")
plt.show()
```



From the heatmap read the following correlations. Also, answer the question below.

```
#Corr(BMI, Outcome)
Corr1 = round(corr[5,8],2)
print(Corr1)

#Corr(Glucose, Outcome)
Corr2 = round(corr[1,8],2)
print(Corr2)

#Corr(Pregnancies, Outcome)
Corr3 = round(corr[0,8],2)
print(Corr3)

#print(Corr1, Corr2, Corr3)

#Out of the 8 features, which two features are the most correlated.
Fill in the list variable bestcorr

# Get upper triangle of correlation matrix, ignoring diagonal
upper_triangle = np.triu(corr, k=1)

# Find the indices of the maximum absolute correlation value
max_corr_index = np.unravel_index(np.argmax(np.abs(upper_triangle)),
corr.shape)
bestcorr_index = list(max_corr_index)
bestcorr = [diabetesData.columns[bestcorr_index[0]],
diabetesData.columns[bestcorr_index[1]]]
print(bestcorr)
```

```
0.31
0.45
0.24
['Pregnancies', 'Age']
```

Section 2

In this section we will train a logistic regression model on the diabetes dataset to predict the risk of diabetes given patient data. We will then use it to perform predictions on previously unseen (test) data.

Q4. Preprocess Data.

Implement a function normalizeData that normalizes each column of a data array. Recall that a column x is normalized by $z = \frac{x - mean(x)}{std(x)}$. The function should return the normalized data matrix and the mean and standard deviations of each column (we'll need these to normalize the test data later).

```
def normalizeData(X, mean = np.array([]), std = np.array([])):
```

```
    #TODO
    X=np.array(X)

    # Implement such that if the mean and std are empty arrays then
mean and std is calculated here, else use the mean and std passed in
as parameters
    if mean.size == 0 and std.size == 0:
        mean = np.mean(X, axis=0)  # Column-wise mean
        std = np.std(X, axis=0, ddof=1)  # Column-wise sample std
(ddof=1)
    std[std == 0] = 1

    normalized_X = (X-mean)/std

    return normalized_X, mean, std
```

Q5. Sigmoid function and it's derivative.

1. Implement the sigmoid function. Given a numpy array, compute the sigmoid values of the array. It should return an array of sigmoid values. If you see any overflow errors/warnings popping up, that is because the numbers coming out of the exponential function e^{-t} in the sigmoid function are too large to handle. In that case, please clip the input between some large enough negative and positive value (look into numpy.clip).
2. Implement the derivative of the sigmoid function. Recall that if, $p=\sigma(t)$, then $p'=p(1-p)$. The function should take in an array of p values and then return the array of p'.

```
#Sigmoid function
def sigmoid(t):

    #TODO
    t = np.clip(t, -500, 500)
    sig = 1/(1+np.exp(-t))

    return sig

#Derivative of sigmoid function
def derivSigmoid(p):

    #TODO

    deriv_p = p*(1-p)

    return deriv_p
```

Q6. Compute sigmoid probabilities.

Recall that $p = P(y = 1 \vee x) = \sigma(x^T \omega + b)$, where $\omega = \begin{bmatrix} \omega_1 \\ \omega_2 \\ \vdots \\ \omega_d \end{bmatrix}$ are the weights and b is the bias term.

Implement a function sigProg to calculate those probabilities, given a matrix X of data, weight vector ω and bias b. It is possible to compute the array of probabilities for the entire dataset at once without a single for-loop, by just using matrix mutltiplications, which is the rightway to achieve maximum efficiency. You are supposed to use the above implemented sigmoid function. This should return an array of probabilities.

```
def sigProg(X, w, b):

    #TODO
    p = sigmoid(np.dot(X,w)+b)

    return p
```

Q7. Compute loss gradient.

Recall that the loss function for logistic regression is given by,

$L(\omega, b) = \frac{1}{N} \sum_{i=1}^{N} (p_i - y_i)^2 + \lambda reg(\omega)$ where is the sigmoid probability $p_i = P(y_i = 1 \vee x_i)$ for the data point (x_i, y_i) and $reg(\omega)$ is the regularization term - it could be either ridge $reg(\omega) = ||\omega||_2^2$ or lasso $reg(\omega) = ||\omega||_1$ or no regularization at all. $\lambda \geq 0$ is the regularization constant. N is the number of datapoints.

Then, the gradient of the loss function with respect to ω and the derivative with respect to b is given by,

$$\nabla_\omega L = \frac{1}{N} \sum_{i=1}^{N} (p_i - y_i) p'_i x_i + \lambda reg'(\omega)$$

$$\frac{\partial L}{\partial b} = \frac{1}{N} \sum_{i=1}^{N} (p_i - y_i) p'_i$$

Here,

$$reg'(\omega) = \begin{cases} \omega & \text{if ridge} \\ sign(\omega) & \text{if lasso} \end{cases}$$

Implement a function named gradient to compute the gradient and derivative of the loss function, given data matrix X, output vector y, weight vector ω and bias b. The function should also be able to take in other parameters such as reg (= "none", "ridge" or "lasso") and lambda. It should return the gradient and derivative. All these computations can be done without a single for loop, only by using numpy builtins for matrix computations, which is the most desired way to achieve efficiency

```
def gradient(X, y, w, b, reg = "none", Lambda = 0.1):

    #TODO
    # Number of data points
    N = X.shape[0]
    p = sigProg(X, w, b)
    p_prime = derivSigmoid(p)
    error = (p - y) * p_prime

    grad_w = (1 / N) * np.dot(X.T, error)
    deriv_b = (1 / N) * np.sum(error)

    if reg == "ridge":
        grad_w += Lambda * w
    elif reg == "lasso":
        grad_w += Lambda * np.sign(w)

    return grad_w, deriv_b
```

Q8. Gradient descent algorithm.

Recall the following update rule for gradient descent,

$\omega \leftarrow \omega - \eta \nabla_{\omega} L$

$b \leftarrow b - \eta \frac{\partial L}{\partial b}$

where $\eta > 0$ is the learning rate.

Implement the gradient descent algorithm in a function grad_descent given the gradient vector $\nabla_{\omega} L$ (grad_w), derivative $\frac{\partial L}{\partial b}$ (deriv_b), weights ω and bias b. Also should be able to accept the learning rate η (eta). The function should return the updated ω and b.

```
def grad_descent(grad_w, deriv_b, w, b, eta = 0.01):

    #TODO
    w = w - eta*grad_w

    b = b - eta*deriv_b

    return w, b
```

Q9. Train model

Implement the function train. It should initialize ω and b to some random values or zeros. Then interatively update the ω and b using gradient descent, untill the number of interations reach the maximum number of interations specified as max_iter. The function will take in training data

(X, y), regularization details (type of reg, and λ), and learning rate η (eta). Finally, it should return the trained ω and b.

```
# Compute the MSE loss
def compute_loss(X, y, w, b, reg="none", Lambda=0.1):
    p = sigmoid(np.dot(X, w) + b)
    loss = np.mean((p - y) ** 2)
    if reg == "ridge":
        loss += Lambda * np.sum(w**2)
    elif reg == "lasso":
        loss += Lambda * np.sum(np.abs(w))
    return loss

def train(X, y, reg = 'none', Lambda = 0.1, eta = 0.01, max_iter =
1000):

    #TODO

    # Number of features
    d = X.shape[1]

    w = np.random.randn(d)  # Small random values for weights
    b = np.random.randn()  # Initialize bias to small random value

    losses = []
    w_norms = []

    for _ in range(max_iter):
        grad_w, deriv_b = gradient(X, y, w, b, reg, Lambda)
        w, b = grad_descent(grad_w, deriv_b, w, b, eta)
        losses.append(compute_loss(X, y, w, b, reg, Lambda))
        w_norms.append(np.linalg.norm(w))

    return w, b, losses, w_norms
```

Q10. Predict using the model

Implement a function predict to output predictions for given input data X, trained weights ω and b. Make sure that the output should only consist of 1's and 0's. The function should return the predictions $\hat{y}$ (yhat).

```
def predict(X, w, b):

    #TODO
    p = sigProg(X, w, b)
    yhat = (p >= 0.5).astype(int)

    return yhat
```

Q11. Use the above logistic regression algorithm on the diabetes dataset. Train a model (ω, b). Then evaluate it. Use classification error to evaluate it.

Start by separating the diabetesData into input (features) and output (Outcome), and store them in numpy matrices X_train and y_train respectively, and then normalizing the input features.

Some tips to improve accuracy.

1. Use cross validation to find the best hyperparameters η, λ, and regularization type.
2. Plot the loss function versus iterations while training (you can do this by additionally collecting the loss values within the train function), having a smooth decreasing graph will indicate proper training.
3. As above plot the ℓ_2-norm of the ω vector versus iterations, we expect smooth decreasing curve for this as well.

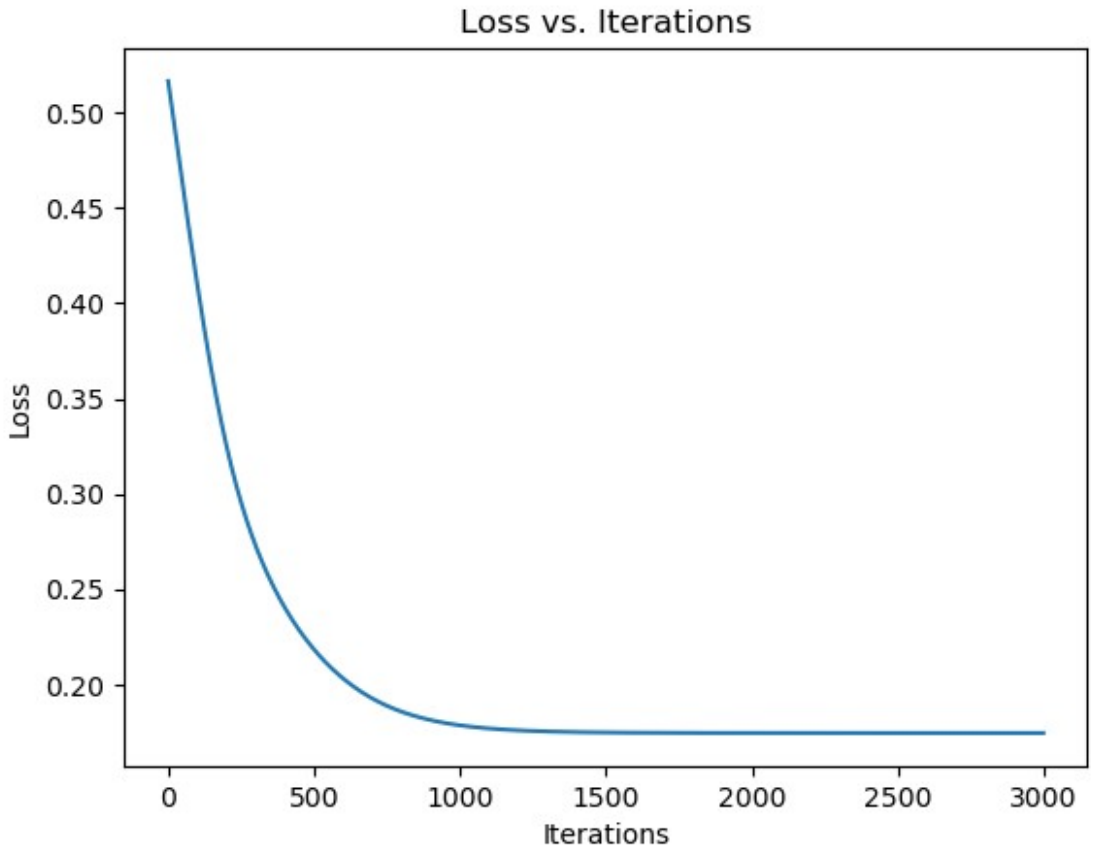
More tips and additional clarifications in class.

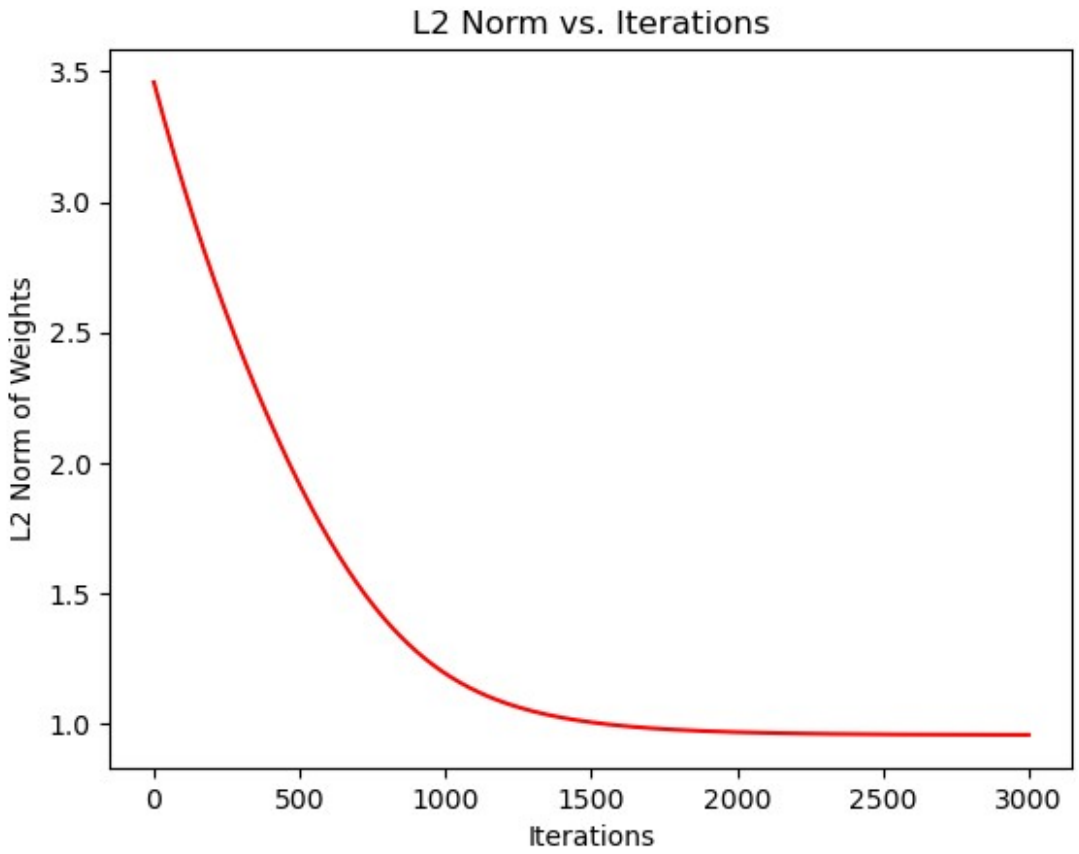
```
#K-fold cross validation
def k_fold_cross_validation(X, y, eta_values=[0.001, 0.01, 0.1],
lambda_values=[0.01, 0.1, 1.0], regularization_types=["none", "ridge",
"lasso"], K=5):
    # Shuffle indices
    np.random.seed(42)
    indices = np.random.permutation(len(X))

    # Split indices into K folds
    fold_size = len(X) // K
    folds = [indices[i * fold_size: (i + 1) * fold_size] for i in
range(K)]

    best_hyperparams = None
    best_error = float("inf")  # Track lowest classification error

    # Iterate over all hyperparameter combinations
    for eta in eta_values:
        for Lambda in lambda_values:
            for reg in regularization_types:
                errors = []  # Store classification errors for each
fold

                for k in range(K):
                    # Get validation fold
                    val_idx = folds[k]
                    X_val, y_val = X[val_idx], y[val_idx]

                    # Get training data (all other folds)
                    train_idx = np.hstack([folds[i] for i in range(K)
if i != k])
                    X_train_cv, y_train_cv = X[train_idx],
y[train_idx]
```

```
                    # Train model
                    w, b, _, _= train(X_train_cv, y_train_cv, reg=reg,
Lambda=Lambda, eta=eta, max_iter=1000)

                    # Predict on validation set
                    y_val_pred = predict(X_val, w, b)

                    # Compute classification error
                    error = np.mean(y_val_pred != y_val)
                    errors.append(error)

                # Compute average error across all folds
                avg_error = np.mean(errors)

                # Track best hyperparameters
                if avg_error < best_error:
                    best_error = avg_error
                    best_hyperparams = (eta, Lambda, reg)
    return best_hyperparams

#TODO
#Train and evaluate your model.
X_train = diabetesData.iloc[:, :-1].values
y_train = diabetesData.iloc[:, -1].values

X_train, mean, std = normalizeData(X_train)


best_hype_params = k_fold_cross_validation(X_train,y_train)
print("(eta, lambda, reg): ",best_hype_params)
# Train the logistic regression model
w, b,losses, w_norms = train(X_train, y_train, best_hype_params[2],
best_hype_params[1], best_hype_params[0], max_iter=3000)

#w, b,losses, w_norms = train(X_train, y_train, "ridge" ,0.1 , 0.1,
max_iter=1000)

# Predict on training data
y_pred = predict(X_train, w, b)
classification_error = np.mean(y_pred != y_train)
print(classification_error)

accuracy = np.mean(y_pred == y_train)
print(f"Accuracy: {accuracy * 100:.2f}%")

plt.plot(losses)
plt.xlabel("Iterations")
plt.ylabel("Loss")
plt.title("Loss vs. Iterations")
plt.show()
```

```
plt.plot(w_norms, color='r')
plt.xlabel("Iterations")
plt.ylabel("L2 Norm of Weights")
plt.title("L2 Norm vs. Iterations")
plt.show()
```

```
(eta, lambda, reg):  (0.1, 0.01, 'ridge')
0.23809523809523808
Accuracy: 76.19%
```





Q12. Testing on previously unseen test data - final part.

1. Load the DiabetesTest.csv file.
2. Use your best trained model to predict the 'Outcome' for this test data, assign your predictions to y_test_pred, this should be an array of 0's and 1's.
3. This will be graded on the level of accuracy of your predictions.
4. Remember to properly normalize your data before using them in the model. For normalization use the mean and standard deviation of the training data not the test data.

```
#TODO

diabetesTest = pd.read_csv("DiabetesTest.csv")

X_test_norm, _, _ = normalizeData(diabetesTest, mean, std)

y_test_pred = predict(X_test_norm, w, b)

print(y_test_pred)
```

```
[0 0 1 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0
1 0
 0 0 0 0 0 0 1 1 0 1 1 1 0 0 0 1 0 1 0 1 0 0 0 0 0 0 0 0 0 1 1 0 0 0 1
0 0
 0 0 0 1 1 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 1]
```