Bangladesh University of Engineering and Technology

EEE 6002 Mid-Term Assignment

Selected Topics in Electrical and Electronic Engineering – Privacy Preserving Machine Learning

Name: Mohammad Zunaed
ID: 0419062239
Email: 0419062239@eee.buet.ac.bd
Semester: April, 2020
Submission Date: 27 February, 2021

1 Assignment Question

Midterm assignment instructions,

- Choose any dataset from the UCI Machine Learning Repository.
- Make sure the dataset satisfies the following criteria:
  - It is a binary classification dataset.
  - The feature dimension is at least 500.
- Split the dataset into train, validation and test sets (e.g., using 70%, 10% and 20% splits). You can use sklearn library for splitting the data into these partitions.
- Write the Logistic Regression code by yourself (do not use any built-in classification function).
- Include the following plots:
  - Training and validation loss vs iterations
  - Norm of the gradient vs iterations
  - Training and validation loss vs number of training samples
  - Training and validation loss vs regularization coefficient λ
  - Required number of iterations vs step size α
  - Precision and Recall vs threshold
  - True Positive Rate vs False Positive Rate (ROC curve)
- The report must contain a brief description of the dataset and the complete code.

2 Dataset Description

For this assignment, we have selected the following dataset: Parkinson's Disease Classification Data Set. The data used in this study were gathered from 188 patients with PD (107 men and 81 women) with ages ranging from 33 to 87 at the Department of Neurology in CerrahpaÅŸa Faculty of Medicine, Istanbul University. The control group consists of 64 healthy individuals (23 men and 41 women) with ages varying between 41 and 82. During the data collection process, the microphone is set to 44.1 KHz and following the physicians examination, the sustained phonation of the vowel /a/ was collected from each subject with three repetitions.

The total number of data instances in this dataset is 756. Each set of data has a feature dimension of 753. 564 instances has been diagnosed with parkinson disease resulting in class imbalance. The dataset fulfills the criteria set by the assignment. It has at least 500 features per patient and it is a binary classification dataset. Various speech signal processing algorithms including Time Frequency Features, Mel Frequency Cepstral Coefficients (MFCCs), Wavelet Transform based Features, Vocal Fold Features and TWQT features have been applied to the speech recordings of patients to extract these 753 clinically useful information for PD assessment.

First, the dataset is split into 80% and 20% respectively for train and test dataset. This test dataset is used for only evaluation of our trained model. The 80% dataset is further split into 90% and 10% respectively for train and validation dataset. The model is trained on the train dataset and the best coefficients for the model is selected on the result based on validation dataset. Before spliting the dataset, the whole dataset has been normalized into $\{0, 1\}$ range. The data distribution is shown in the table 1.

Table 1: Data Distribution.

Type	Train	Validation	Test
Sample Number	543	61	152
Percentage	72	8	20

The function that I use to process the dataset and required libraries for my implementation are given next:

```
import os
import time
import torch
import random
import numpy as np
from tqdm import tqdm
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.metrics import precision_score, recall_score, roc_curve,
    roc_auc_score
import pandas as pd
from sklearn.preprocessing import MinMaxScaler

def get_data():
    # read the dataset
    df = pd.read_csv('./pd_speech_features.csv')
    X_features = df.values[1:,1:-1].astype(np.float)
    y_labels = df.values[1:,-1].astype(np.int)

```

```
    # normalize the dataset
    scaler = MinMaxScaler()
    scaler = scaler.fit(X_features)
    X_features = scaler.transform(X_features)

    # train, validation, test split
    X, X_test, y, y_test = train_test_split(X_features, y_labels, test_size
    =0.20, random_state=42)
    X_train, X_validation, y_train, y_validation = train_test_split(X, y,
    test_size=0.10, random_state=42)

    return X_train, y_train, X_validation, y_validation, X_test, y_test
```

3 Logistic Regression with Differential Privacy mechanism

The code for Logistic Regression has been implemented from scratch. The code files are uploaded in the following github repository https://github.com/rafizunaed/BUET-EEE_6002-PPML-Assignments.
Suppose the coefficients are ω, input features are X, ground truth is y, regularization coefficient is λ, m is the total number of data, n is the number of coefficients, r is the feature dimension, and α is the step size. The algorithm for logistic regression is:

$$logits, f(x) = \omega X = \omega_1 x_0 + \omega_2 x_1 + \omega_3 x_2 + \omega_4 x_3 + ... + \omega_n x_r \tag{1}$$

$$sigmoid\ function,\ \sigma(z) = \frac{1}{1 + \exp(-z)} \tag{2}$$

$$hypothesis,\ \hat{y}(\omega, X) = \sigma(\omega X) \tag{3}$$

$$loss,\ L(y, \hat{y}, \omega) = -\ \Big[\ \frac{1}{m}\sum_{i=1}^{m}(y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)) - \frac{\lambda}{2m}\sum_{i=1}^{n}||\omega_i||_2^2\ \Big] \tag{4}$$

$$gradient,\ \nabla_\omega L = \frac{1}{m}(X^T(\hat{y} - y) + \lambda\omega) \tag{5}$$

$$weight\ update,\ \omega = \omega - \alpha\nabla_\omega L \tag{6}$$

Here, ω_1 is the bias term and we will assume x_0 is 1. The functions implementing the above algorithm are given below:

```
def set_random_state(seed_value):
    random.seed(seed_value)
    torch.manual_seed(seed_value)
    torch.cuda.manual_seed(seed_value)
    torch.cuda.manual_seed_all(seed_value)
    os.environ['PYTHONHASHSEED'] = str(seed_value)
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False
    np.random.seed(seed_value)

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def calculate_loss(features, target, coefficients,
   regularization_coefficient):
    logits = np.dot(features, coefficients)
```

```
    logits_sigmoid = sigmoid(logits)
    loss = (- np.sum(target*np.log(logits_sigmoid)+ (1-target)*np.log(1-logits_sigmoid)) \
            + 0.5*regularization_coefficient*np.dot(coefficients, coefficients))/len(target)
    return loss

def train_one_iteration(features, target, coefficients, learning_rate, regularization_coefficient):
    f_x = np.dot(features, coefficients)
    p_x = sigmoid(f_x)
    gradient = (np.dot(features.T, (p_x - target)) + regularization_coefficient * coefficients)/features.shape[0]
    coefficients -= learning_rate * gradient
    return coefficients, gradient

def logistic_regression(features_train, target_train, features_validation, target_validation, add_x0,
                        iterations, learning_rate, regularization_coefficient):
    set_random_state(42)

    if add_x0:
        x0 = np.ones((features_train.shape[0], 1))
        features_train = np.hstack((x0, features_train))
        x0 = np.ones((features_validation.shape[0], 1))
        features_validation = np.hstack((x0, features_validation))
    coefficients = np.zeros(features_train.shape[1])

    best_coefficients = np.zeros(features_train.shape[1])
    best_loss = 9999
    count = 0
    patience = 100
    best_iteration = 0

    all_train_loss=[]
    all_validation_loss=[]
    all_gradients=[]
    progess_bar = tqdm(total=iterations, unit='iterations')
    for step in range(iterations):
        coefficients, gradient = train_one_iteration(features_train, target_train, coefficients, learning_rate, regularization_coefficient)

        train_loss = calculate_loss(features_train, target_train, coefficients, regularization_coefficient)
        validation_loss = calculate_loss(features_validation, target_validation, coefficients, regularization_coefficient)

        if validation_loss < best_loss:
            best_loss = validation_loss
            best_coefficients = coefficients.copy()
            count = 0
            best_iteration = step
        else:
            count += 1

        progess_bar.set_postfix({'Train loss':train_loss, ' Validation_loss':validation_loss})
        progess_bar.update(1)

```

```
        all_train_loss.append(train_loss)
        all_validation_loss.append(validation_loss)
        all_gradients.append(gradient)

        if count == patience:
            learning_rate = learning_rate * 0.5
            count = 0

    progess_bar.close()

    all_gradients = np.stack(all_gradients)
    all_gradients_norm = np.linalg.norm(all_gradients, axis=-1)

    return best_coefficients, all_train_loss, all_validation_loss,
   all_gradients_norm, best_iteration
```

4 Plots

First, we will plot loss vs iterations, the norm of the gradients, precision and recall scores vs threshold, and the ROC curves. For this, we selected step size=0.04, regularization coefficient=0.001 and ran the training for 4000 iterations. We calculate and plot these figures by running the following function:

```
def train(add_x0=True, iterations=5000, learning_rate=0.5,
   regularization_coefficient=0.001):

    X_train, y_train, X_validation, y_validation, X_test, y_test = get_data
   ()
    best_coefficients, all_train_loss, all_validation_loss,
   all_gradients_norm, _ = logistic_regression(
                                                X_train, y_train,
                                                X_validation, y_validation,
                                                add_x0,
                                                iterations,
                                                learning_rate,
                                                regularization_coefficient,
                                                )

    os.makedirs('./logs/', exist_ok=True)
    # plot of 5a. train and validation loss vs iterations
    iterations = np.arange(0, iterations)
    plt.plot(iterations, all_train_loss, label='train_loss')
    plt.plot(iterations, all_validation_loss, label='validation_loss')
    plt.legend()
    plt.xlabel('iterations')
    plt.ylabel('loss')
    plt.title('loss vs iterations [5a]')
    plt.savefig("./logs/5a_loss_vs_iterations.png", bbox_inches='tight',
   pad_inches=0)
    plt.show()

    # plot of 5b. Norm of the gradient vs iterations
    plt.plot(iterations, all_gradients_norm)
    plt.xlabel('iterations')
    plt.ylabel('Norm of the gradient')
    plt.title('Norm of the gradient vs iterations [5b]')
    plt.savefig("./logs/5b_Norm_of_the_gradient_vs_iterations.png",
   bbox_inches='tight', pad_inches=0)
    plt.show()
```

```
    if add_x0:
        X_test = np.hstack((np.ones((X_test.shape[0], 1)),X_test))
    logits = np.dot(X_test, best_coefficients)
    prediction_probabilites = sigmoid(logits)

    all_precisions=[]
    all_recalls=[]
    all_thresholds=np.arange(0, 1, 0.01)
    for threshold in all_thresholds:
        y_true = y_test.copy()
        y_pred = prediction_probabilites >= threshold
        all_precisions.append(precision_score(y_true, y_pred, average='macro'))
        all_recalls.append(recall_score(y_true, y_pred, average='macro'))

    # plot of 5f. Precision and Recall vs threshold
    plt.plot(all_thresholds, all_precisions, label='Precision')
    plt.plot(all_thresholds, all_recalls, label='Recall')
    plt.legend()
    plt.xlabel('Thresholds')
    plt.ylabel('Precision and Recall Score')
    plt.title('Precision and Recall Score vs Thresholds [5f]')
    plt.savefig("./logs/5f_Precision_and_Recall_Score_vs_Thresholds.png", bbox_inches='tight', pad_inches=0)
    plt.show()

    # plot of 5g. True Positive Rate vs False Positive Rate (ROC curve)
    fpr, tpr, _ = roc_curve(y_true, prediction_probabilites)
    roc_score = roc_auc_score(y_true, prediction_probabilites, 'macro')
    plt.plot(fpr, tpr, color='darkorange', lw=2, label='ROC curve (area = {:.3f})'.format(roc_score))
    plt.legend(loc="lower right")
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('True Positive Rate vs False Positive Rate (ROC curve) [5g]')
    plt.savefig("./logs/5g_True_Positive_Rate_vs_False_Positive_Rate.png", bbox_inches='tight', pad_inches=0)
    plt.show()

    log_dict = {
        'all_train_loss': all_train_loss,
        'all_validation_loss': all_validation_loss,
        'all_gradients_norm': all_gradients_norm,
        'iterations': iterations,
        'all_thresholds': all_thresholds,
        'all_precisions': all_precisions,
        'all_recalls': all_recalls,
        'fpr': fpr,
        'tpr': tpr,
        'roc_score': roc_score,
        }

    np.save('./logs/log_dict_1.npy', log_dict)
if __name__ == '__main__':
    # fig a,b,f,g
    train(iterations=4000, learning_rate=0.04, regularization_coefficient=0.001)
```

4.1 Training and Validation loss vs iterations

From the figure 1, we can see that validation loss converges between 3500 and 4000 iterations. The difference between training and validation loss indicates the overfitting of the model.

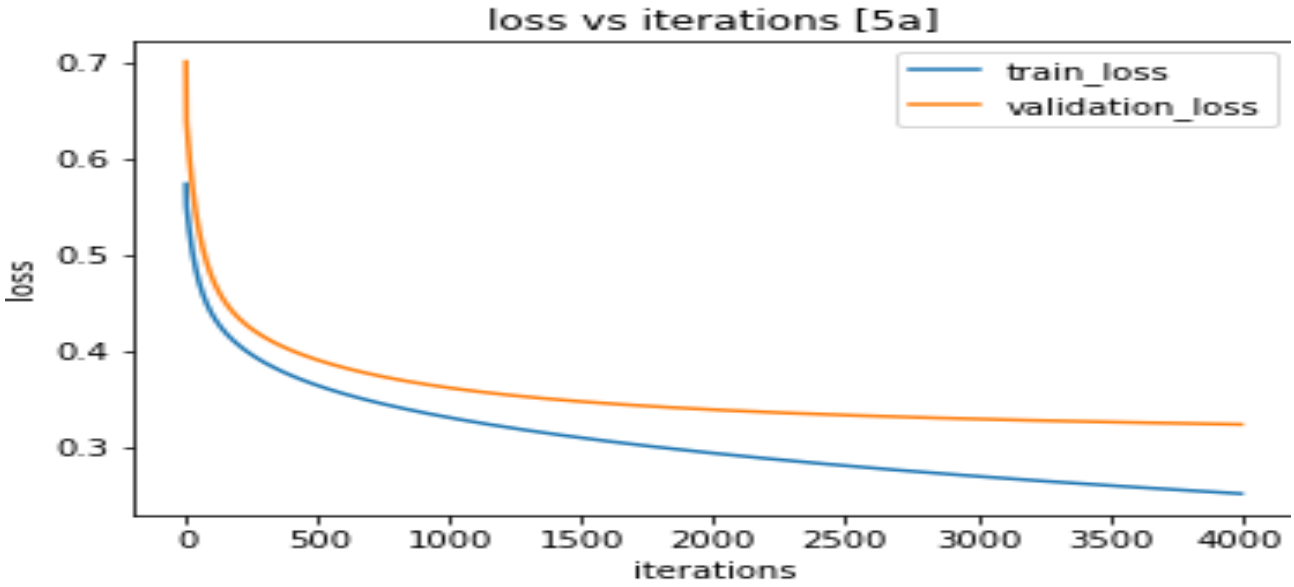

Figure 1: Training and Validation loss vs iterations.

4.2 Norm of the gradient vs iterations

Now, we plot the Norm of the gradient against iterations in the figure 2. We see that, with the increasing number of iterations, the norm of the gradient is decreasing. This means that our model is converging to minimize the loss.

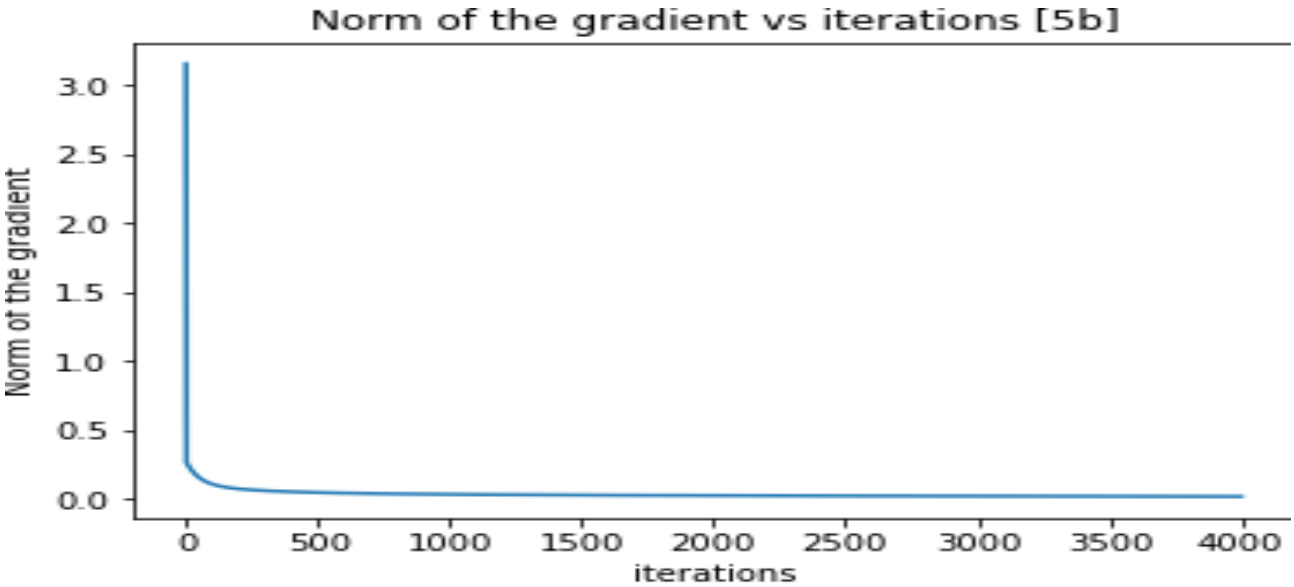

Figure 2: Norm of the gradient vs iterations.

4.3 Precision and recall vs iterations

After that, we evaluate our model on the test dataset by loading the best coefficients. We vary the threshold from 0 to 1 and plot precision and recall versus thresholds. The plots are given in the figure 3. We can see from the figure that high precision is achieved at lower threshold while high recall is achieved at higher threshold.

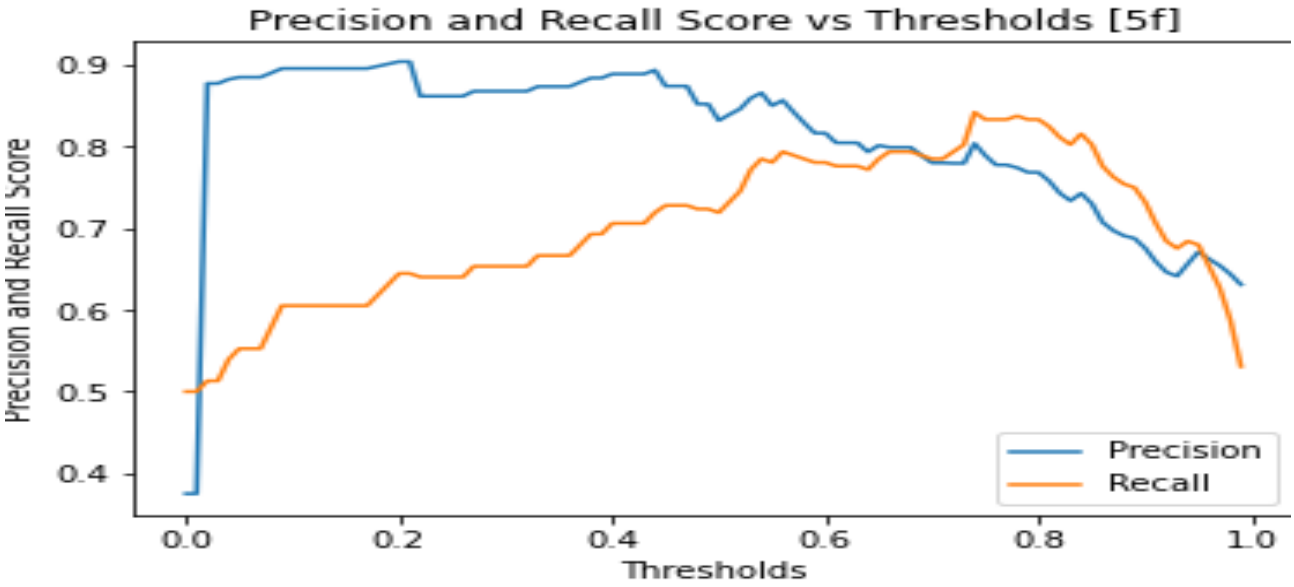

Figure 3: Precision and recall vs iterations.

4.4 True positive rate vs False positive rate

Now, we plot the ROC curve of our model prediction by plotting the true positive rate versus the false positive rate in figure 4. Our model has achieved ROC area of 0.891 which is very close to ideal area of 1.

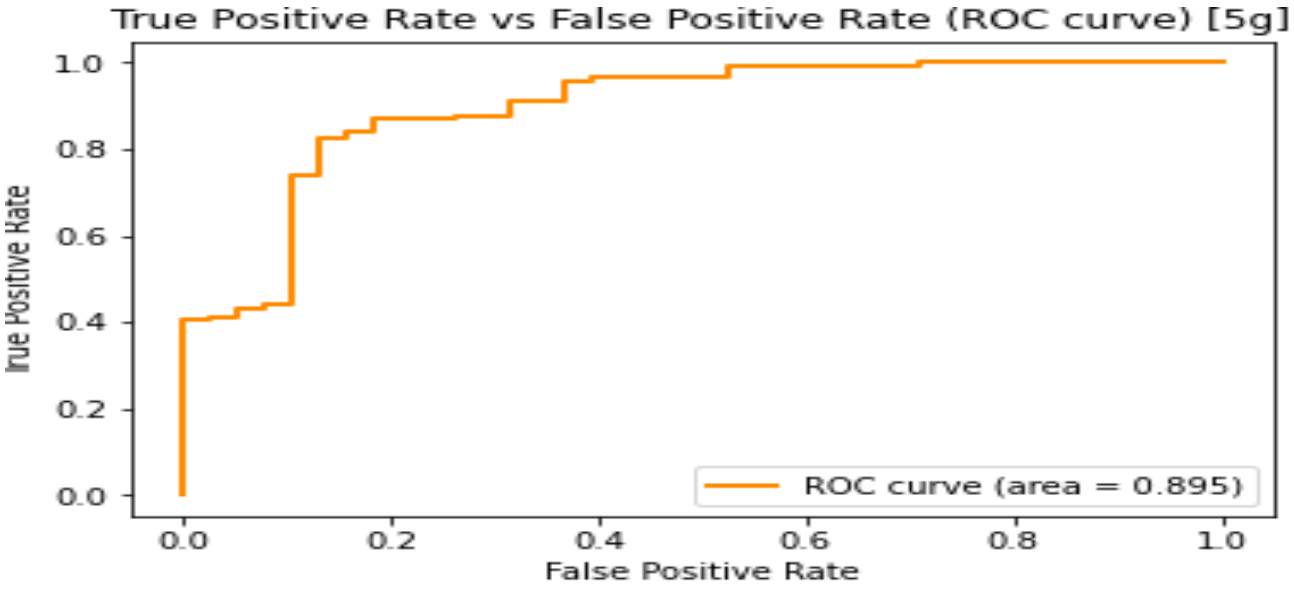

Figure 4: True positive rate vs False positive rate.

4.5 Training and validation loss vs Number of training samples

For demonstrating the effect of the number of training samples on training and validation loss, we gradually decrease the number of training samples keeping the validation dataset intact, and plot training and validation loss vs the number of training samples in figure 1. We can see from the figure that, for the non-differential privacy method as the number of training samples decreases, the validation loss is increasing as well. It is expected as less number of training samples means fewer data available for the model to learn. Inversely, training loss is decreasing at the lower number of training samples because of overfitting.

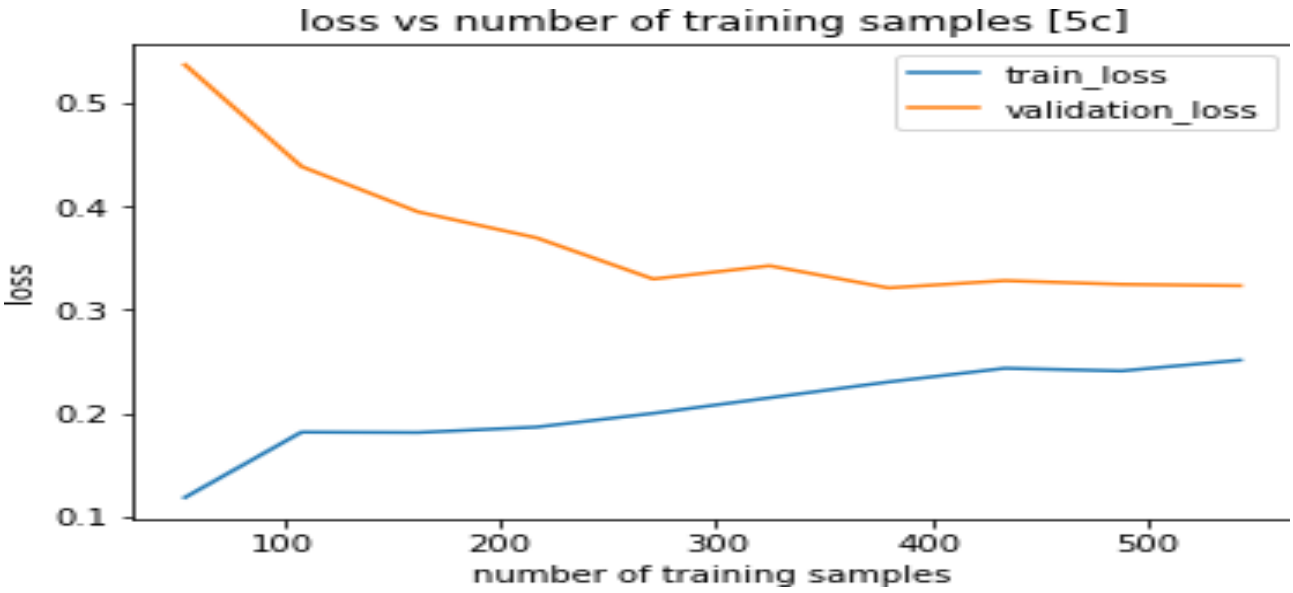

Figure 5: Training and validation loss vs Number of training samples.

The code to calculate these plots are given below:

```
def check_num_samples_effect(add_x0=True, iterations=5000, learning_rate
    =0.5, regularization_coefficient=0.001):

     keep_train_percentage = [1, 0.9, 0.8, 0.7, 0.6, 0.5, 0.4, 0.3, 0.2, 0.1]
     train_losses=[]; val_losses=[]; all_number_training_samples=[];
     for percentage in keep_train_percentage:
          print('Current percentage: {}'.format(percentage))
          X_train, y_train, X_validation, y_validation, X_test, y_test =
    get_data()
          new_len = int(X_train.shape[0]*percentage)
          X_train = X_train[0:new_len,...]
          y_train = y_train[0:new_len]
          best_coefficients, all_train_loss, all_validation_loss,
    all_gradients_norm, _ = logistic_regression(
                                       X_train, y_train,
                                       X_validation, y_validation,
                                       add_x0,
                                       iterations,
                                       learning_rate,
                                       regularization_coefficient,
                                       )

          train_losses.append(np.min(all_train_loss))
          val_losses.append(np.min(all_validation_loss))
          all_number_training_samples.append(new_len)
          time.sleep(1)
     all_train_losses = np.array(train_losses)
```

```
    all_val_losses = np.array(val_losses)
    all_number_training_samples = np.array(all_number_training_samples)
    idx = np.argmin(all_val_losses)
    print('Best val_loss achieved is: {} at num_samples: {}'.format(
    all_val_losses[idx], all_number_training_samples[idx]))
    os.makedirs('./logs/', exist_ok=True)
    # plot of 5d. train and validation loss vs number of training samples
    plt.plot(all_number_training_samples, all_train_losses, label='
    train_loss')
    plt.plot(all_number_training_samples, all_val_losses, label='
    validation_loss')
    plt.legend()
    plt.xlabel('number of training samples')
    plt.ylabel('loss')
    plt.title('loss vs number of training samples [5c]')
    plt.savefig("./logs/5c_loss_vs_number_of_training_samples.png",
    bbox_inches='tight', pad_inches=0)
    plt.show()
    log_dict = {
        'all_train_loss': all_train_losses,
        'all_validation_loss': all_val_losses,
        'all_number_training_samples': all_number_training_samples,
        }
    np.save('./logs/log_dict_3.npy', log_dict)
if __name__ == '__main__':
    # fig c
    check_num_samples_effect(iterations=4000, learning_rate=0.04,
    regularization_coefficient=0.001)
```

4.6 Training and validation loss vs Regularization coefficient

We vary the regularization coefficient and calculate the best training and validation loss for each regularization coefficient. We plot these losses vs regularization coefficients in the figure 6. From the figure, we can see that both training and validation loss increases with the regularization coefficient. It is expected as the regularization coefficient add a penalty to the loss function. For very low values of regularization coefficients, the effect on training and validation loss is minimal, almost constant.

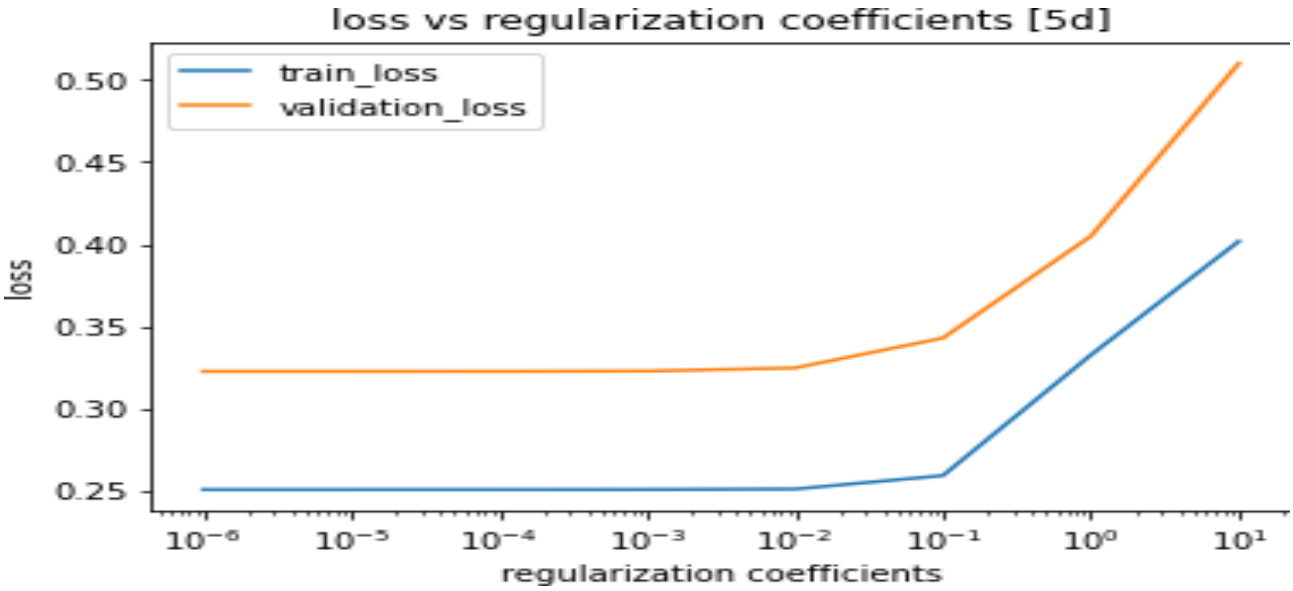

Figure 6: Training and validation loss vs Regularization coefficient.

The code to calculate these plots are given below:

```
def check_regularization_effect(add_x0=True, iterations=5000, learning_rate=0.5, regularization_coefficient=0.001):

    regularization_coefficients=[10, 1, 0.1, 0.01, 0.001, 0.0001, 0.00001, 0.000001]
    train_losses=[]; val_losses=[];
    X_train, y_train, X_validation, y_validation, X_test, y_test = get_data()
    for regularization_coefficient in regularization_coefficients:
        print('Current regularization_coefficient: {}'.format(regularization_coefficient))
        best_coefficients, all_train_loss, all_validation_loss, all_gradients_norm, _ = logistic_regression(
                                                X_train, y_train,
                                                X_validation, y_validation,
                                                add_x0,
                                                iterations,
                                                learning_rate,
                                                regularization_coefficient,
                                                )

        train_losses.append(np.min(all_train_loss))
        val_losses.append(np.min(all_validation_loss))
        time.sleep(1)

    all_train_losses = np.array(train_losses)
    all_val_losses = np.array(val_losses)
    regularization_coefficients = np.array(regularization_coefficients)

    idx = np.argmin(all_val_losses)
    print('Best val_loss achieved is: {} at regulariztion coefficient: {}'.format(all_val_losses[idx], regularization_coefficients[idx]))

    os.makedirs('./logs/', exist_ok=True)
    # plot of 5d. train and validation loss vs regularization coefficients
    plt.plot(regularization_coefficients, all_train_losses, label='train_loss')
    plt.plot(regularization_coefficients, all_val_losses, label='validation_loss')
    plt.xscale('log')
    plt.legend()
    plt.xlabel('regularization coefficients')
    plt.ylabel('loss')
    plt.title('loss vs regularization coefficients [5d]')
    plt.savefig("./logs/5d_loss_vs_regularization_coefficients.png", bbox_inches='tight', pad_inches=0)
    plt.show()

    log_dict = {
        'all_train_loss': all_train_losses,
        'all_validation_loss': all_val_losses,
        'regularization_coefficients': regularization_coefficients,
        }

    np.save('./logs/log_dict_2.npy', log_dict)
if __name__ == '__main__':
    # fig d
    check_regularization_effect(iterations=4000, learning_rate=0.04, regularization_coefficient=0.001)
```

4.7 Required number of iterations vs Step size

Now, we vary the step size and record the number of iterations required for the convergence of the model. We also recorded the best validation loss for each of the step sizes. We plot the required number of iterations vs step size in the figure 7 with blobs representing the best validation losses. The blobs are plotted in the 'jet' color map. Red color blob means highest loss validation value, while royal blue means lowest validation loss value.

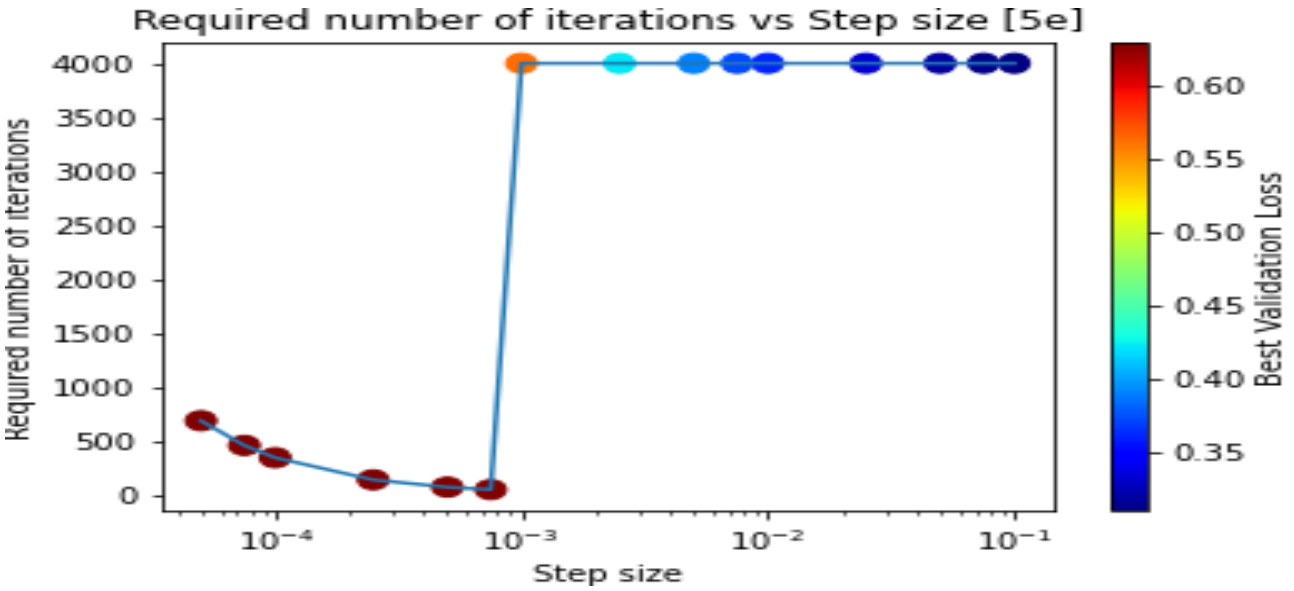

Figure 7: Required number of iterations vs Step size.

```
def check_step_size_effect(add_x0=True, iterations=5000, learning_rate=0.5,
    regularization_coefficient=0.001):

     learning_rates=[0.1, 0.075, 0.05, 0.025, 0.01, 0.0075, 0.005, 0.0025,
    0.001, 0.00075, 0.0005, 0.00025, 0.0001,
                     0.000075, 0.00005]
     val_losses=[]; all_required_number_iterations=[];
     X_train, y_train, X_validation, y_validation, X_test, y_test = get_data
    ()
     for learning_rate in learning_rates:
         print('Current step size: {}'.format(learning_rate))
         _, _, all_validation_loss, _, best_iteration = logistic_regression(
                                                 X_train, y_train,
                                                 X_validation, y_validation,
                                                 add_x0,
                                                 iterations,
                                                 learning_rate,
                                                 regularization_coefficient,
                                                 )

         all_required_number_iterations.append(best_iteration)
         val_losses.append(np.min(all_validation_loss))
         time.sleep(1)

     all_required_number_iterations = np.array(all_required_number_iterations
    )
     all_best_loss = np.array(val_losses)
     step_sizes = np.array(learning_rates)

     idx = np.argmin(all_best_loss)
```

```
    print('Best val_loss achieved is: {} at step size: {}'.format(all_best_loss[idx], step_sizes[idx]))

    os.makedirs('./logs/', exist_ok=True)
    # plot of 5e. Required number of iterations vs Step size
    plt.scatter(step_sizes, all_required_number_iterations, s=80, c=all_best_loss, cmap='jet')
    plt.plot(step_sizes, all_required_number_iterations)
    plt.xscale('log')
    plt.colorbar(label='Best Validation Loss')
    plt.xlabel('Step size')
    plt.ylabel('Required number of iterations')
    plt.title('Required number of iterations vs Step size [5e]')
    plt.savefig("./logs/5e_Required_number_of_iterations_vs_Step_size.png", bbox_inches='tight', pad_inches=0)
    plt.show()

    log_dict = {
        'all_required_number_iterations': all_required_number_iterations,
        'all_best_loss': all_best_loss,
        'step_sizes': step_sizes,
        }

    np.save('./logs/log_dict_4.npy', log_dict)
if __name__ == '__main__':
    # fig e
    check_step_size_effect(iterations=4000, learning_rate=0.04, regularization_coefficient=0.001)
```