Bangladesh University of Engineering and Technology

EEE 6002 Final 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

1 Assignment Question

Final assignment instructions,

- Choose the MNIST dataset and consider digits 5 and 8 to be the two classes.
- 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).
- Modify the Logistic Regression code such that it satisfies differential privacy (you can use either Laplace mechanism or Gaussian mechanism).
- Include the following plots for both non-private and private classifiers for $\epsilon = 0.01$ and $\delta = 1e - 5$:
  - 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)
- Plot the classification accuracy against various ϵ values for fixed δ and sample size.
- Plot the classification accuracy against varying sample size for a fixed ϵ and δ.
- The report must contain a brief description of the dataset and the complete code.

2 Dataset Description

The MNIST dataset is a large dataset of handwritten digits widely used in the field of machine learning. The images in the dataset were normalized to fit into a 28x28 pixel bounding box and anti-aliased. Their color map is grayscale. The MNIST database contains 60,000 training images and 10,000 testing images. We downloaded the dataset from the PyTorch repository using their built-in function and merge the files. The MNIST dataset contains ten digits. However, for our problem, we need only the digits 5 and 8. For the gaussian mechanism, we normalize the dataset by 2-norm, and for the laplacian mechanism, we normalize it by 1-norm. Afterward, we split the dataset into train, validation, and test dataset according to the portion given in the assignment. The data distribution is shown in the table 1. Each of the data has 785 (including bias term) features. A few examples are shown in the figure 1.

Table 1: Data Distribution.

Type	Train	Validation	Test
Sample Number	9459	1051	2628
Percentage	72	8	20

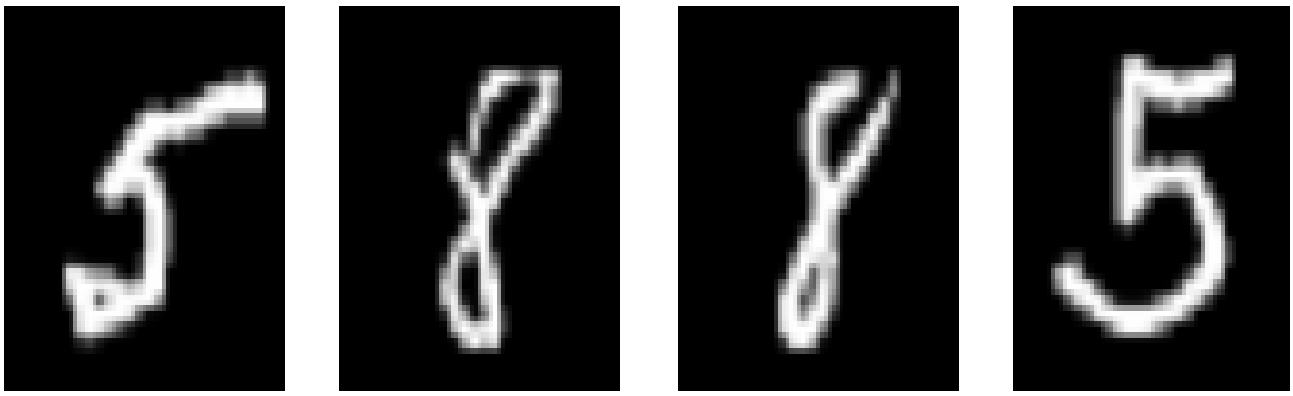


Figure 1: MNIST DATASET EXAMPLES.

Required libraries for my implementation are:

```
import os
import time
import torch
import random
import numpy as np
from tqdm import tqdm
import matplotlib.pyplot as plt
from torchvision import datasets, transforms
from sklearn.model_selection import train_test_split
from sklearn.metrics import precision_score, recall_score, roc_curve,
    roc_auc_score, accuracy_score
```

The function that I use to process the dataset is given next:

```
def get_data(do_norm=False, norm_order=2):
    transform = transforms.Compose([transforms.ToTensor()])
    train_data = datasets.MNIST(root='data', train=True, download=True, transform=transform)
    test_data = datasets.MNIST(root='data', train=False, download=True, transform=transform)

    train_loader = torch.utils.data.DataLoader(train_data, batch_size=512)
    test_loader = torch.utils.data.DataLoader(test_data, batch_size=512)

    X_all = []; y_all = [];

    for _, (X,y) in enumerate(train_loader):
        X_all.append(X.squeeze(1).view(X.size(0), -1).data.numpy())
        y_all.append(y.data.numpy())

    for _, (X,y) in enumerate(test_loader):
        X_all.append(X.squeeze(1).view(X.size(0), -1).data.numpy())
        y_all.append(y.data.numpy())

    X_all = np.concatenate(X_all)
    y_all = np.concatenate(y_all)

    X_all = X_all[np.logical_or(y_all == 5, y_all == 8)]
    y_all = y_all[np.logical_or(y_all == 5, y_all == 8)]
    y_all[y_all == 5] = 0
    y_all[y_all == 8] = 1

    if do_norm:
        X_all_norm = np.linalg.norm(X_all, norm_order, axis=-1)
        X_all_norm_max = np.max(X_all_norm)
        X_all = X_all / X_all_norm_max

    X, X_test, y, y_test = train_test_split(X_all, y_all, 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:

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

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

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

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

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

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

Here, ω_1 is the bias term and we will assume x_0 is 1. In the case of the Gaussian or Laplacian differential privacy mechanism, the only change will occur in the gradient calculation phase. We will add a noise term with the gradient before updating the weight values. For the Gaussian mechanism, the data will be norm-2 normalized. The updated steps for the Gaussian mechanism are:

$$sensitivity,\ \Delta_f = \frac{2}{m} \tag{7}$$

$$gaussian\ noise, n_g = N(0, \frac{2\Delta_f^2}{\epsilon^2}\log(1.25/\delta)) \tag{8}$$

$$new\ gradient,\ (\nabla_\omega L)_{new} = \nabla_\omega L + n_g \tag{9}$$

$$weight\ update,\ \omega = \omega - \alpha(\nabla_\omega L)_{new} \tag{10}$$

Here, N is normal distribution with 0 mean value. For the Laplacian mechanism, the data will be norm-1 normalized. The updated steps for the Laplacian mechanism are:

$$sensitivity,\ \Delta_f = \frac{2}{m} \tag{11}$$

$$beta, \beta = \frac{\Delta_f}{\epsilon} \tag{12}$$

$$laplacian\ noise, n_l = L_p(0, beta) \tag{13}$$

$$new\ gradient,\ (\nabla_\omega L)_{new} = \nabla_\omega L + n_l \tag{14}$$

$$weight\ update,\ \omega = \omega - \alpha(\nabla_\omega L)_{new} \tag{15}$$

Here, L_p is laplacian distribution with 0 mean value.

The functions implementing the above algorithms are given below:

```
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_dp_laplace(features, target, coefficients, learning_rate,
                                   regularization_coefficient, epsilon=0.01):
    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]
    sensitivity = 2 / features.shape[0]
    beta = sensitivity / epsilon
    laplace_noise = np.random.laplace(0, beta, gradient.shape[0])
    gradient_new = gradient + laplace_noise
    coefficients -= learning_rate * gradient_new
    return coefficients, gradient

def train_one_iteration_dp_gaussian(features, target, coefficients, learning_rate,
                                   regularization_coefficient, epsilon=0.01, delta=1e-5):
    sigma = (2/features.shape[0]) * np.sqrt(2*np.log(1.25/delta)) / epsilon
    gaussian_noise = np.random.normal(loc=0.0, scale=sigma*sigma, size=coefficients.shape[0])
    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]
    gradient_new = gradient + gaussian_noise
    coefficients -= learning_rate * gradient_new
    return coefficients, gradient

def train_one_iteration_non_dp(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, method='non_dp', epsilon=0.01, delta=1e-5):
    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):
        if method == 'non_dp':
            coefficients, gradient = train_one_iteration_non_dp(features_train, target_train, coefficients, learning_rate, regularization_coefficient)
        elif method == 'gaussian':
            coefficients, gradient = train_one_iteration_dp_gaussian(features_train, target_train, coefficients, learning_rate, regularization_coefficient, epsilon, delta)
        elif method == 'laplace':
            coefficients, gradient = train_one_iteration_dp_laplace(features_train, target_train, coefficients, learning_rate, regularization_coefficient, epsilon)

        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

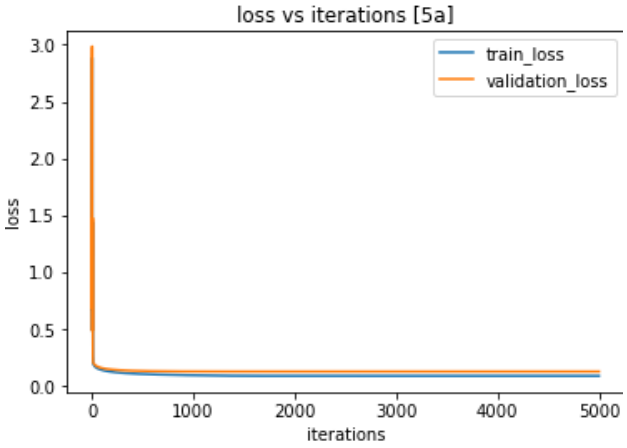
First, we will plot loss vs iterations, the norm of the gradients, precision and recall scores vs threshold, and the ROC curve for both non-differential and differential privacy methods. We calculate and plot these figures by running the following function:

```
def train(add_x0=True, iterations=5000, learning_rate=0.5,
    regularization_coefficient=0.001, method='non_dp',
           epsilon=0.01, delta=1e-5, data_norm=False, norm_order=2):

    X_train, y_train, X_validation, y_validation, X_test, y_test = get_data(
    data_norm, norm_order)
    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,
                                        method,
                                        epsilon,
                                        delta,
                                        )

    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".format(method),
    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".format
    (method), bbox_inches='tight', pad_inches=0)
    plt.show()

    if add_x0:
        X_test = np.hstack((np.ones((X_test.shape[0], 1)),X_test))
```

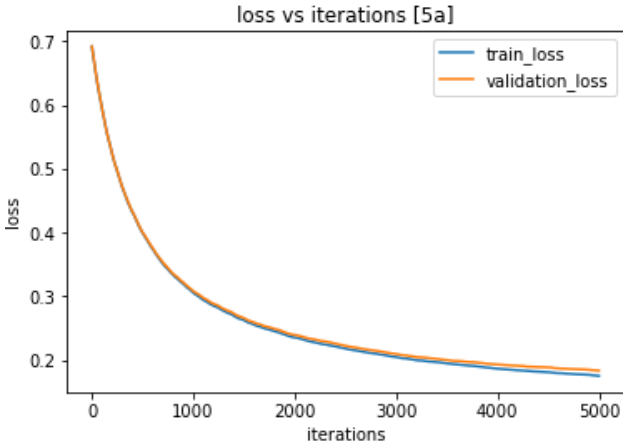
```
    logits = np.dot(X_test, best_coefficients)
    prediction_probabilites = sigmoid(logits)

    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".format(method), 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".format(method), 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'.format(method), log_dict)
if __name__ == '__main__':
    # fig a,b,f,g for non_dp, gaussian, laplace
    train(iterations=5000, method='non_dp', data_norm=False)
    train(iterations=5000, method='gaussian', data_norm=True, norm_order=2)
    train(iterations=5000, learning_rate=5, regularization_coefficient=0.001, method='laplace', data_norm=True, norm_order=1)
```

4.1 Training and Validation loss vs iterations

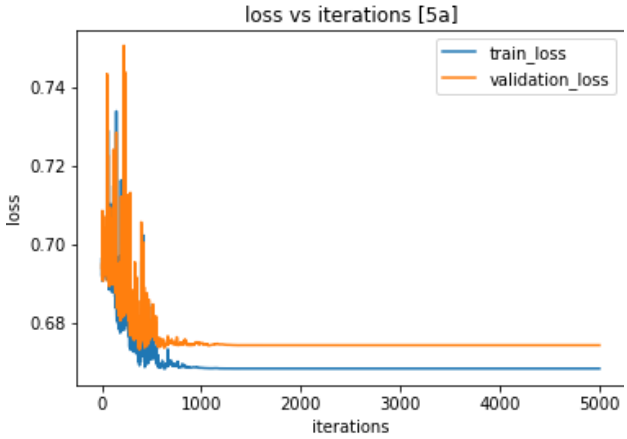
From the figure 2, we can see that the non-differential privacy method quickly converges but the gaussian method takes more iterations to converge. Also, the non-differential privacy method has a lower loss value than the gaussian method. However, the laplacian method has a very high loss value and also does not converge smoothly. In all the cases, validation loss is slightly higher than the training losses which indicates slight overfitting.



(a) Without Differential Privacy Method



(b) Gaussian Differential Privacy Method

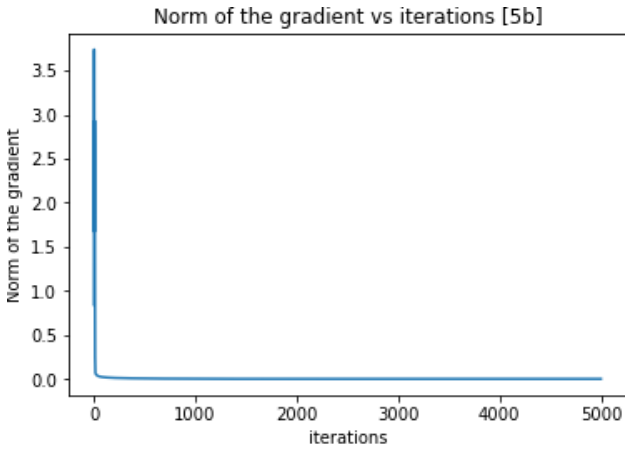


(c) Laplacian Differential Privacy Method

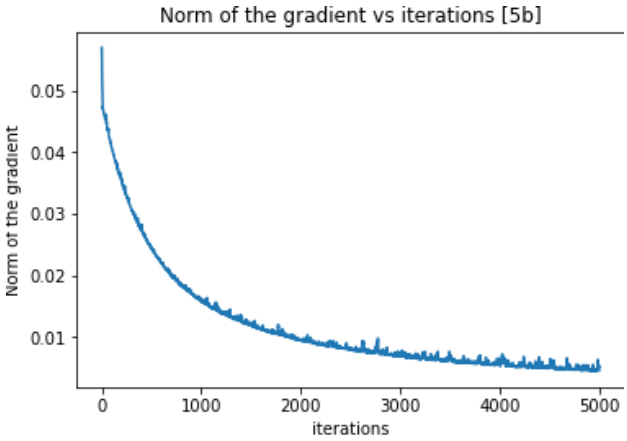
Figure 2: Training and Validation loss vs iterations.

4.2 Norm of the gradient vs iterations

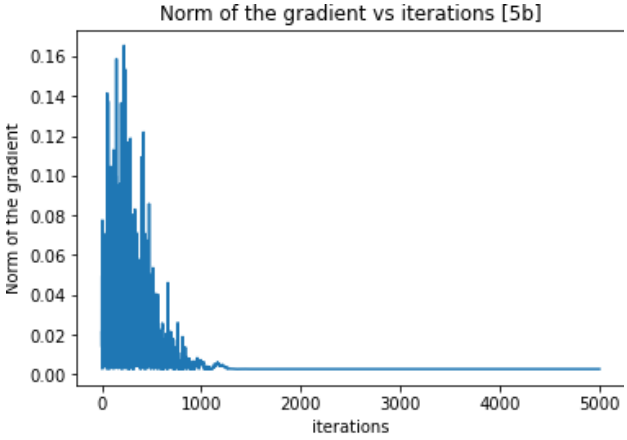
Now, we plot the Norm of the gradient against iterations in the figure 3. We see that, with the increasing number of iterations, the norm of the gradient is decreasing. This means that our model is converging to minimize the loss. For the non-differential privacy method, the curve is smooth. But for the differential methods, curves are not smooth due to added noise. The Gaussian mechanism is much more stable and close to the non-differential method than the Laplacian mechanism.



(a) Without Differential Privacy Method



(b) Gaussian Differential Privacy Method

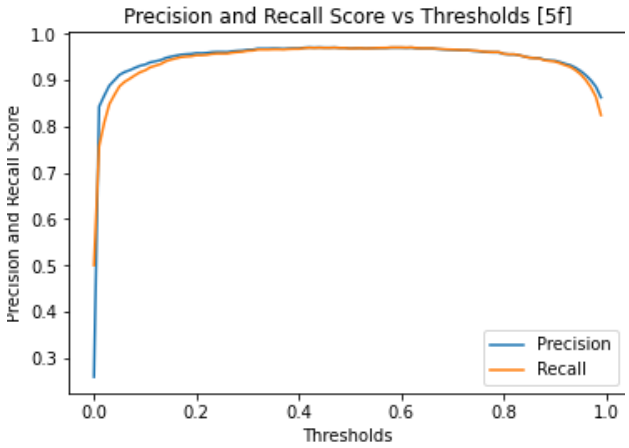


(c) Laplacian Differential Privacy Method

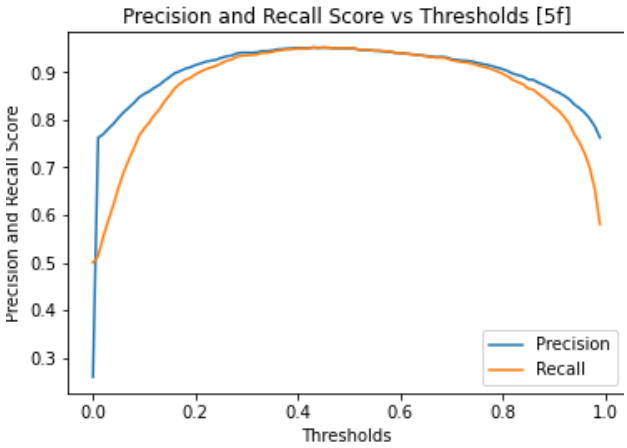
Figure 3: Norm of the gradient vs iterations.

4.3 Precision and recall vs iterations

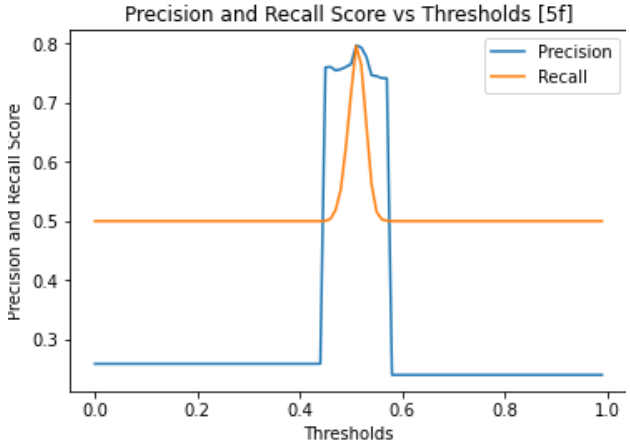
After that, we evaluate our model on the test dataset by loading the best coefficients. We vary the threshold from 0 to 1 and plot precision and recall versus thresholds. The plots are given in the figure 4. Abrupt behavior is noticed during the evaluation of the model when the laplacian mechanism is applied. The non-differential privacy method has very high precision and recall values in almost all the thresholds. The gaussian method achieves slightly less precision and recall values but they are very comparable to the non-differential method.



(a) Without Differential Privacy Method



(b) Gaussian Differential Privacy Method

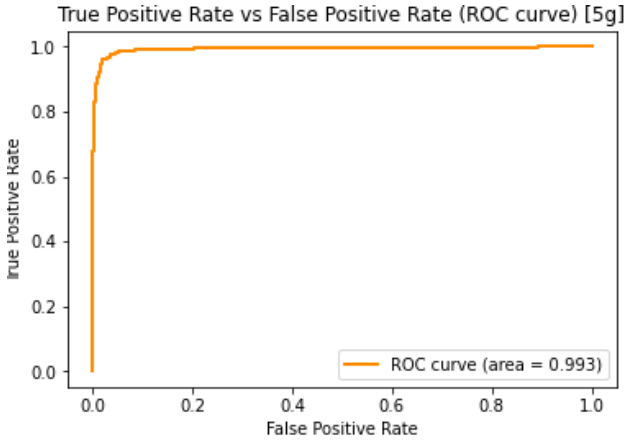


(c) Laplacian Differential Privacy Method

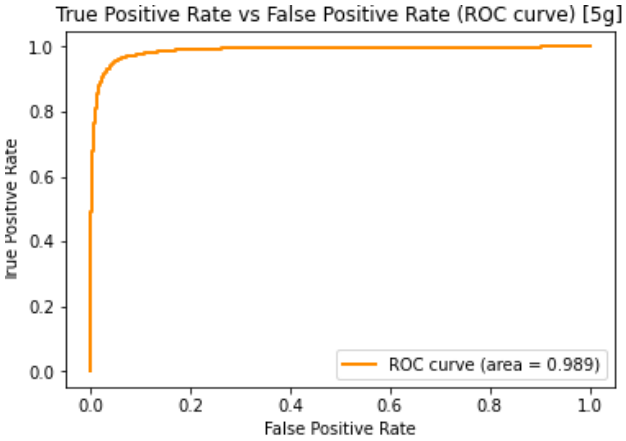
Figure 4: Precision and recall vs iterations.

4.4 True positive rate vs False positive rate

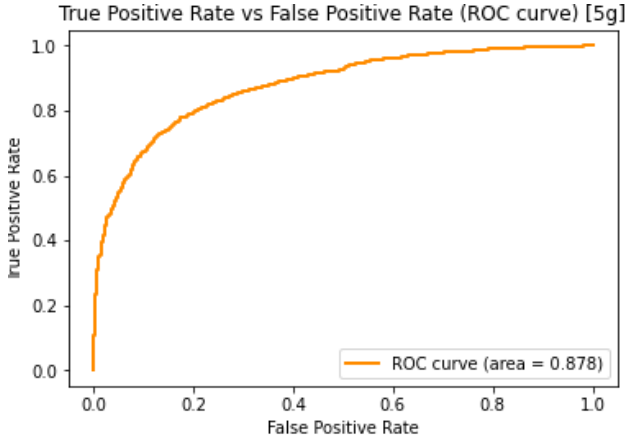
Now, we plot the ROC curve of our model prediction by plotting the true positive rate versus the false positive rate in figure 5. The non-differential method achieved a ROC area of 0.993 while the Gaussian achieved 0.989 and the Laplacian achieved 0.878. It is expected as the Gaussian method usually achieves a better result than the Laplacian method. Also, the non-differential method achieves the highest ROC value due to no privacy mechanism. The privacy mechanisms compromise small performance by giving more security to the data.



(a) Without Differential Privacy Method



(b) Gaussian Differential Privacy Method

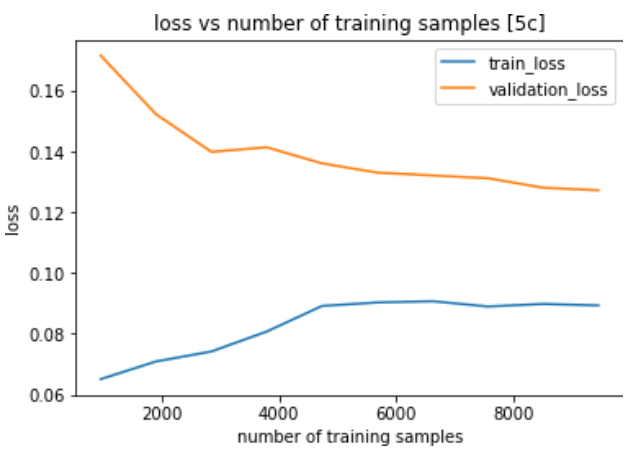


(c) Laplacian Differential Privacy Method

Figure 5: True positive rate vs False positive rate.

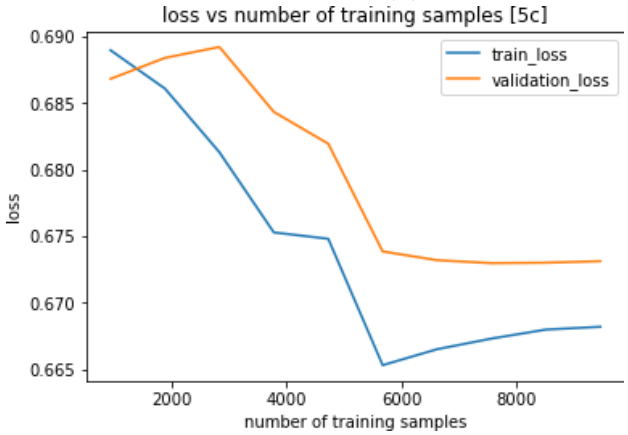
4.5 Training and validation loss vs Number of training samples

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 6. 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. However, for differential privacy methods this relation does not hold. With the increasing number of training samples, both training and validation loss are decreasing. Because, noise is added in each iteration and as a result, the model cannot overfit. As a result, the difference between training and validation loss is very low here.



(a) Without Differential Privacy Method

(b) Gaussian Differential Privacy Method



(c) Laplacian Differential Privacy Method

Figure 6: Training and validation loss vs Number of training samples.

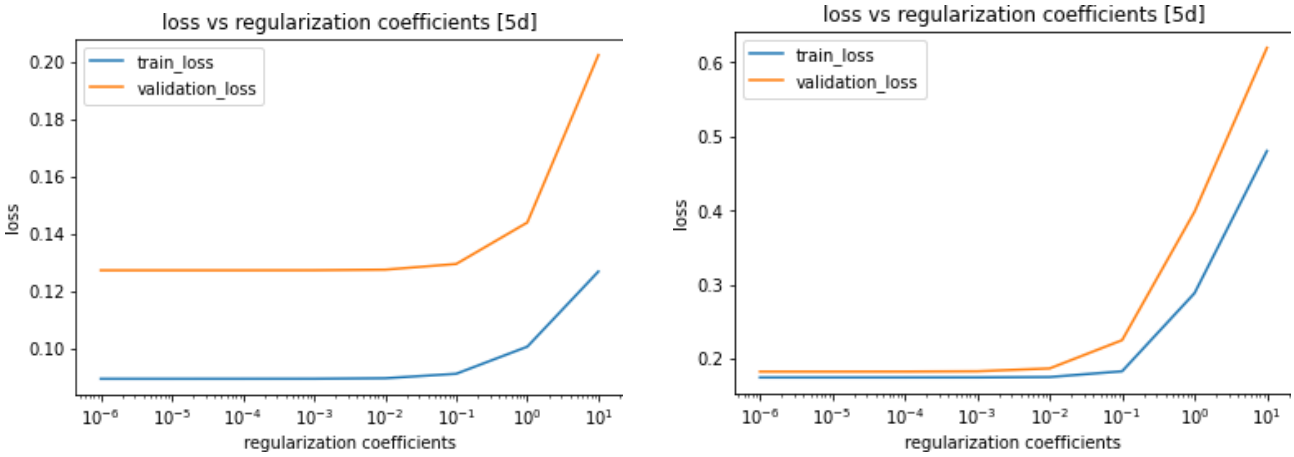
The code to calculate these plots are given below:

```
def check_num_samples_effect(add_x0=True, iterations=5000, learning_rate
    =0.5, regularization_coefficient=0.001, method='non_dp',
           epsilon=0.01, delta=1e-5, data_norm=False, norm_order=2):

    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(data_norm, norm_order)
```

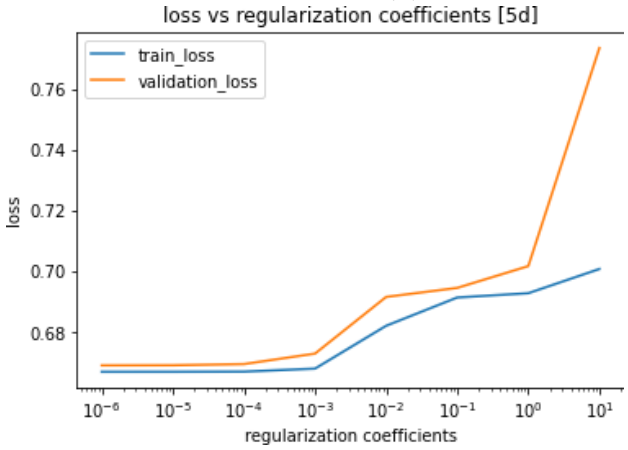
```
        new_len = int(X_train.shape[0]*percentage)
        X_train = X_train[0:new_len,...]
        y_train = y_train[0:new_len]
        best_coefficients, all_train_loss, all_validation_loss,
    all_gradients_norm, _ = logistic_regression(
                                        X_train, y_train,
                                        X_validation, y_validation,
                                        add_x0,
                                        iterations,
                                        learning_rate,
                                        regularization_coefficient,
                                        method,
                                        epsilon,
                                        delta,
                                        )

        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".format
    (method), 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'.format(method), log_dict)
if __name__ == '__main__':
    # fig c for non_dp, gaussian, laplace
    check_num_samples_effect(iterations=5000, method='non_dp', data_norm=
    False)
    check_num_samples_effect(iterations=5000, method='gaussian', data_norm=
    True, norm_order=2)
    check_num_samples_effect(iterations=5000, learning_rate=5,
    regularization_coefficient=0.001, method='laplace', data_norm=True,
    norm_order=1)
```

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 7. 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.



(a) Without Differential Privacy Method (b) Gaussian Differential Privacy Method



(c) Laplacian Differential Privacy Method

Figure 7: Training and validation loss vs Regularization coefficient.

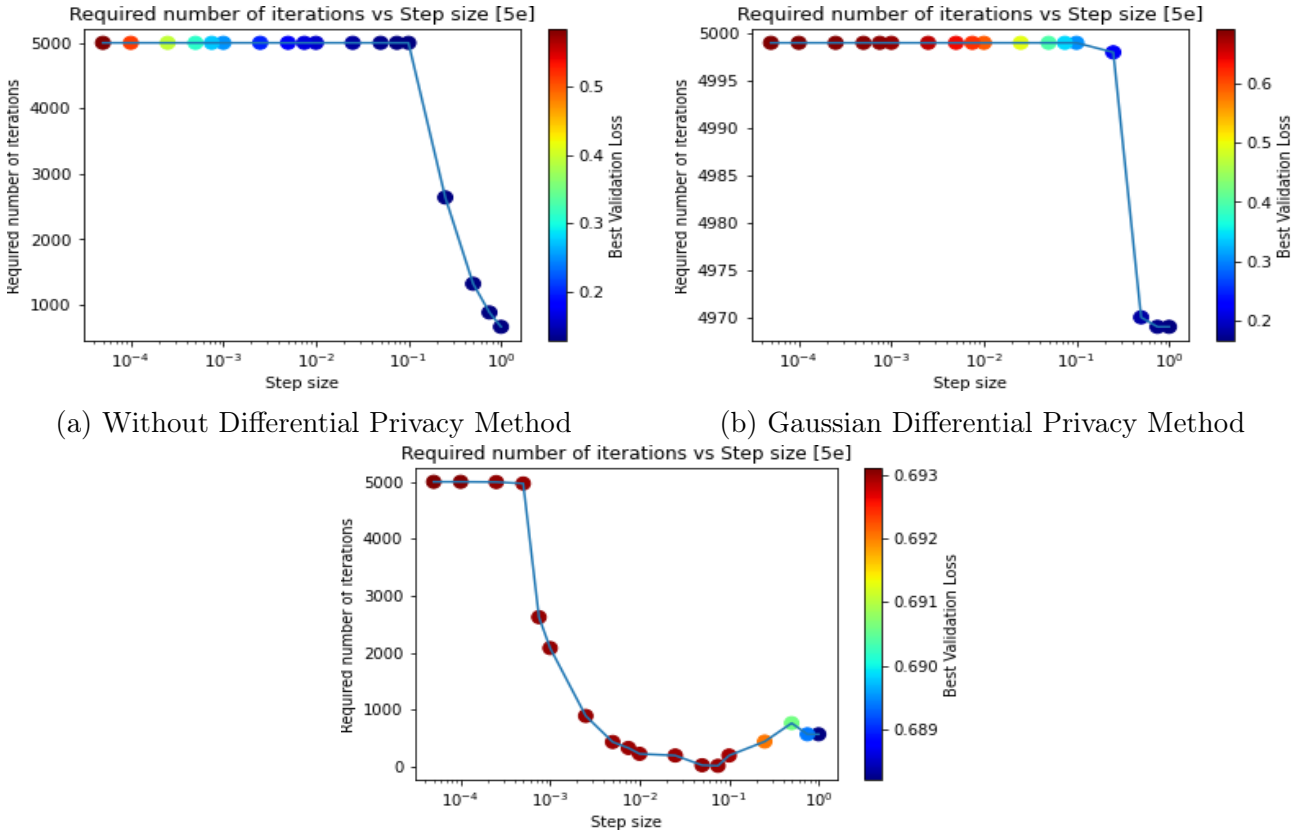
The code to calculate these plots are given below:

```
def check_regularization_effect(add_x0=True, iterations=5000, learning_rate=0.5, regularization_coefficient=0.001, method='non_dp',
          epsilon=0.01, delta=1e-5, data_norm=False, norm_order=2):

    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(data_norm, norm_order)
    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,
                                        method,
                                        epsilon,
                                        delta,
                                        )

        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".format(method), 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'.format(method), log_dict)
if __name__ == '__main__':
    # fig d for non_dp, gaussian, laplace
    check_regularization_effect(iterations=5000, method='non_dp', data_norm=False)
```

```
50     check_regularization_effect(iterations=5000, method='gaussian',
       data_norm=True, norm_order=2)
51     check_regularization_effect(iterations=5000, learning_rate=5,
       regularization_coefficient=0.001, method='laplace', data_norm=True,
       norm_order=1)
```

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 8 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.



(a) Without Differential Privacy Method (b) Gaussian Differential Privacy Method

(c) Laplacian Differential Privacy Method

Figure 8: Required number of iterations vs Step size.

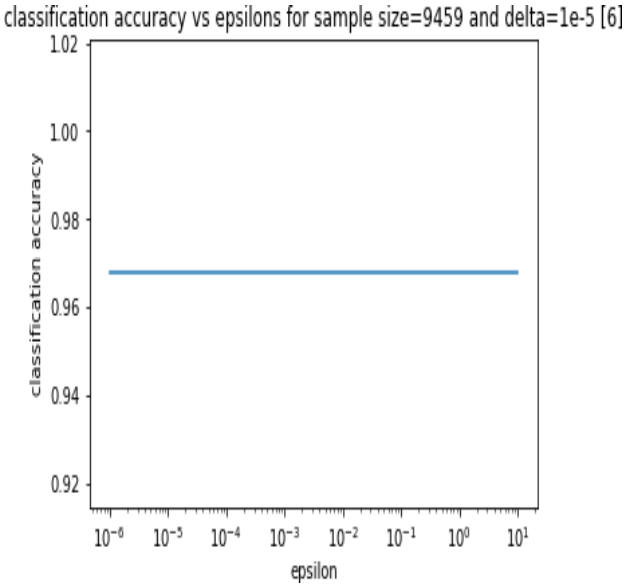
From the figure, we can see that the required number of iterations is decreasing with the increase of step size. The best validation loss is decreasing with the increase in the step size. For non-differential privacy and the Gaussian method, the best validation loss is achieved around step size=0.5. The required number of iterations is also less at this step size compared to lower step size values. For the Laplacian method, the best validation loss is achieved at step size=1. It required a bit more iterations than the lowest required number of iterations which is achieved at a lower step size value. The code to calculate these plots are given below:

```
1  def check_step_size_effect(add_x0=True, iterations=5000, learning_rate=0.5,
       regularization_coefficient=0.001, method='non_dp', epsilon=0.01, delta=1e
       -5, data_norm=False, norm_order=2):
2      learning_rates=[1, 0.75, 0.5, 0.25, 0.1, 0.075, 0.05, 0.025, 0.01,
       0.0075, 0.005, 0.0025, 0.001,
```

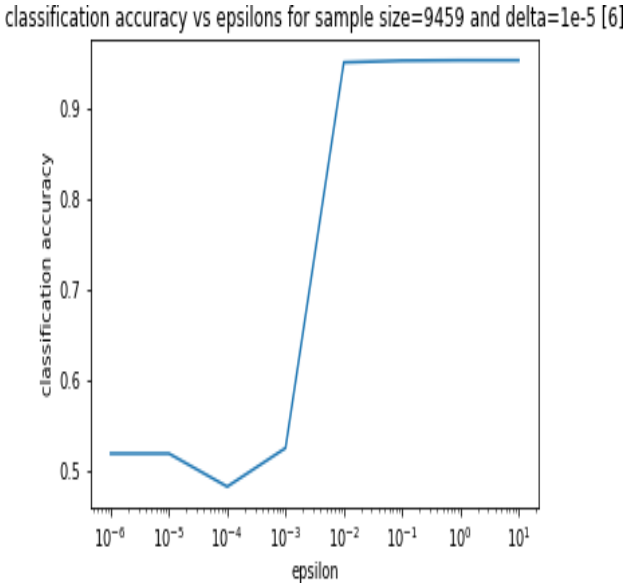
```
                    0.00075, 0.0005, 0.00025, 0.0001, 0.00005]
    val_losses=[]; all_required_number_iterations=[];
    X_train, y_train, X_validation, y_validation, X_test, y_test = get_data(data_norm, norm_order)
    for learning_rate in learning_rates:
        print('Current step size: {}'.format(learning_rate))
        _, _, all_validation_loss, _, best_iteration = logistic_regression(
                                            X_train, y_train,
                                            X_validation, y_validation,
                                            add_x0,
                                            iterations,
                                            learning_rate,
                                            regularization_coefficient,
                                            method,
                                            epsilon,
                                            delta,
                                            )
        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".format(method), 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'.format(method), log_dict)
if __name__ == '__main__':
    # fig e for non_dp, gaussian, laplace
    check_step_size_effect(iterations=5000, method='non_dp', data_norm=False)
    check_step_size_effect(iterations=5000, method='gaussian', data_norm=True, norm_order=2)
    check_step_size_effect(iterations=5000, learning_rate=5, regularization_coefficient=0.001, method='laplace', data_norm=True, norm_order=1)
```

4.8 Classification accuracy vs epsilon values for fixed delta and sample size.

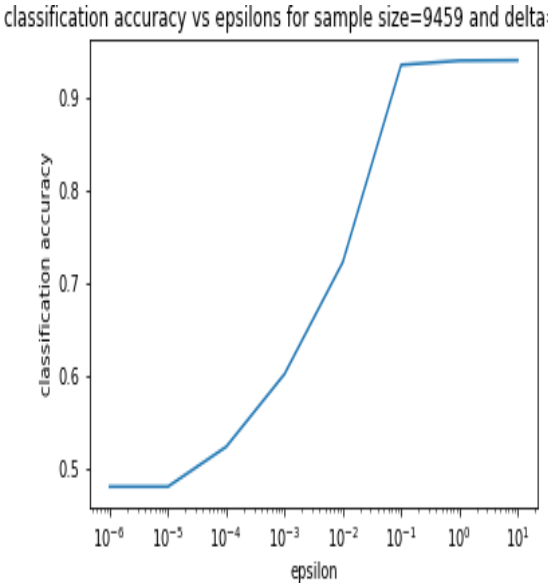
Now, we vary the epsilon values and calculate classification accuracy and ROC value for each of the epsilon values keeping delta and sample size fixed. For the non-differential privacy method, they have no effects as epsilon is only used in differential privacy methods. For differential privacy methods, with the decrease in epsilon value, decrease in classification accuracy and ROC value is observed. A decrease in epsilon value means an increase in privacy. So, the more private the algorithm is, the less accurate the model will be. The plots are shown in the figure 9 10 and code for calculating and plotting them is given afterward.



(a) Without Differential Privacy Method

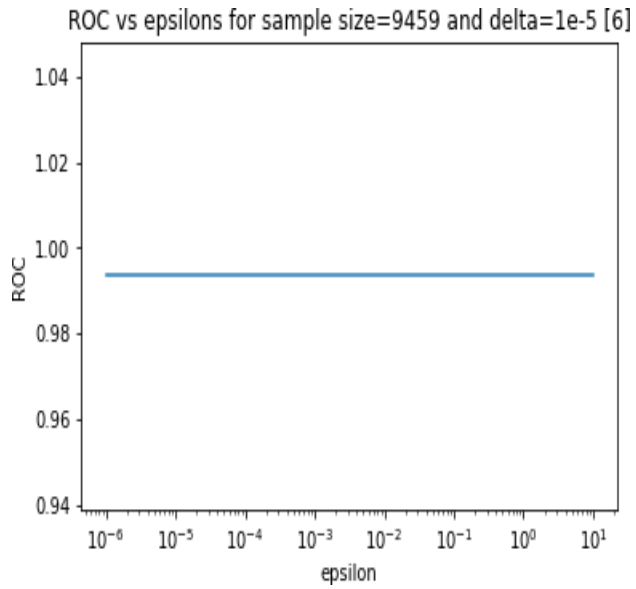


(b) Gaussian Differential Privacy Method

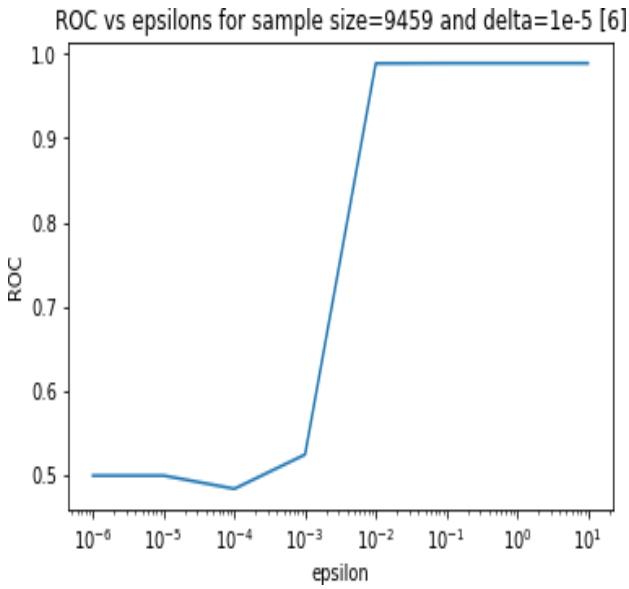


(c) Laplacian Differential Privacy Method

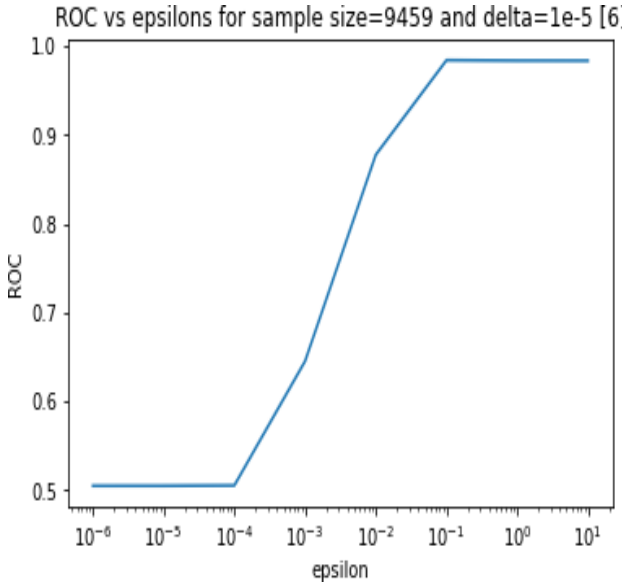
Figure 9: Classification accuracy vs epsilon values for delta=1e-5 and sample size=9459.



(a) Without Differential Privacy Method



(b) Gaussian Differential Privacy Method



(c) Laplacian Differential Privacy Method

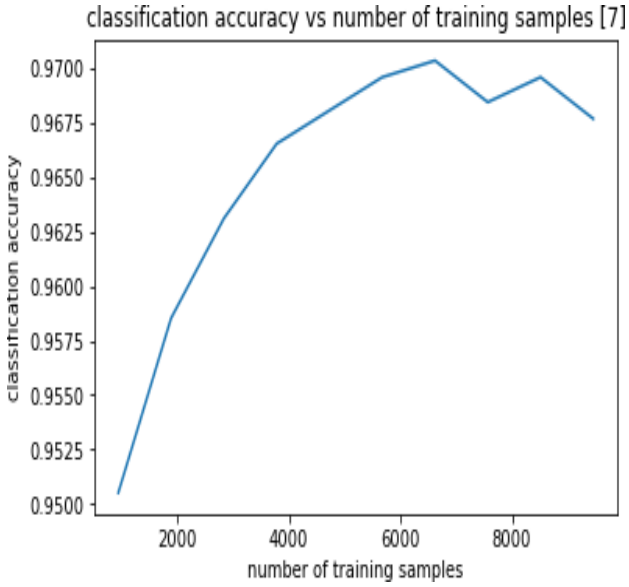
Figure 10: ROC vs epsilon values for delta=1e-5 and sample size=9459.

```
def check_acc_roc_vs_epsilon_effect(add_x0=True, iterations=5000,
    learning_rate=0.5, regularization_coefficient=0.001, method='non_dp',
           epsilon=0.01, delta=1e-5, data_norm=False, norm_order=2):

    all_epsilons = [10, 1, 0.1, 0.01, 0.001, 0.0001, 0.00001, 0.000001]
    X_train, y_train, X_validation, y_validation, X_test, y_test = get_data(
    data_norm, norm_order)
    if add_x0:
        X_test = np.hstack((np.ones((X_test.shape[0], 1)),X_test))

    all_acc=[];all_roc=[];
    for epsilon in all_epsilons:
        best_coefficients, _, _, _, _ = logistic_regression(
                                                X_train, y_train,
                                                X_validation, y_validation,
                                                add_x0,
                                                iterations,
                                                learning_rate,
```

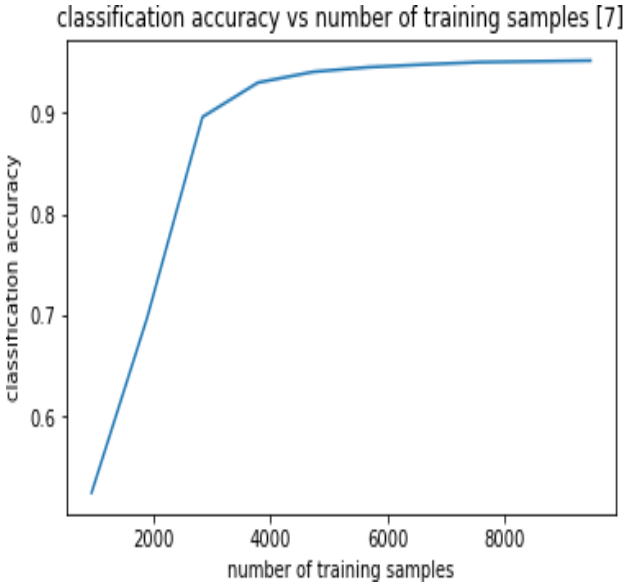
```
                                                regularization_coefficient,
                                                method,
                                                epsilon,
                                                delta,
                                                )

        logits = np.dot(X_test, best_coefficients)
        prediction_probabilites = sigmoid(logits)
        roc_score = roc_auc_score(y_test, prediction_probabilites, 'macro')
        y_pred = prediction_probabilites >= 0.5
        acc = accuracy_score(y_test, y_pred)
        all_acc.append(acc)
        all_roc.append(roc_score)

    os.makedirs('./logs/', exist_ok=True)
    plt.plot(all_epsilons, all_acc)
    plt.xscale('log')
    plt.xlabel('epsilon')
    plt.ylabel('classification accuracy')
    plt.title('classification accuracy vs epsilons for sample size={} and delta=1e-5 [6]'.format(int(X_train.shape[0])))
    plt.savefig("./logs/{}_6_classification_accuracy_vs_epsilon.png".format(method), bbox_inches='tight', pad_inches=0)
    plt.show()

    plt.plot(all_epsilons, all_roc)
    plt.xscale('log')
    plt.xlabel('epsilon')
    plt.ylabel('ROC')
    plt.title('ROC vs epsilons for sample size={} and delta=1e-5 [6]'.format(int(X_train.shape[0])))
    plt.savefig("./logs/{}_6_ROC_vs_epsilon.png".format(method), bbox_inches='tight', pad_inches=0)
    plt.show()

    log_dict = {
        'all_acc': np.array(all_acc),
        'all_epsilons': np.array(all_epsilons),
        'all_roc': np.array(all_roc)
        }

    np.save('./logs/{}_log_dict_5.npy'.format(method), log_dict)
if __name__ == '__main__':
    # fig 6 for non_dp, gaussian, laplace
    check_acc_roc_vs_epsilon_effect(iterations=5000, method='non_dp', data_norm=False)
    check_acc_roc_vs_epsilon_effect(iterations=5000, method='gaussian', data_norm=True, norm_order=2)
    check_acc_roc_vs_epsilon_effect(iterations=5000, learning_rate=5, regularization_coefficient=0.001, method='laplace', data_norm=True, norm_order=1)
```

4.9 Classification accuracy vs number of sample sizes for fixed epsilon and delta.

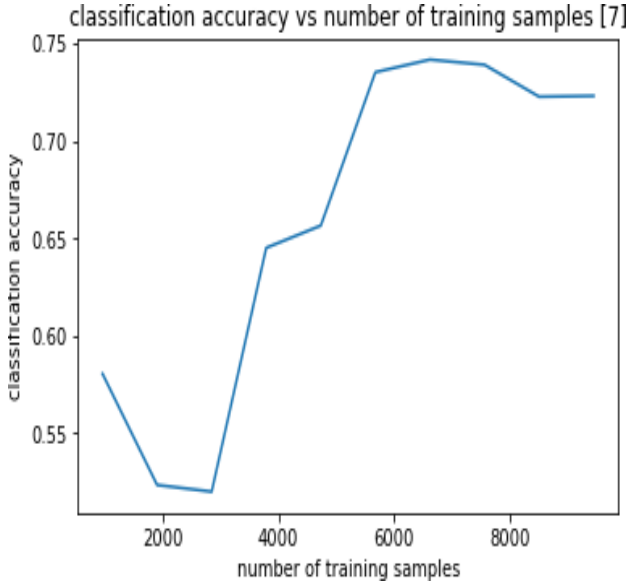
Now, we vary the number of training samples and calculate classification accuracy and ROC value for fixed epsilon and delta value. The plots are given in the figure 11 12. From these figures, it is evident that with an increase in the number of samples, the classification accuracy and ROC value increase for both non-differential and differential privacy methods. Code for calculating and plotting them is given after the figures.



(a) Without Differential Privacy Method

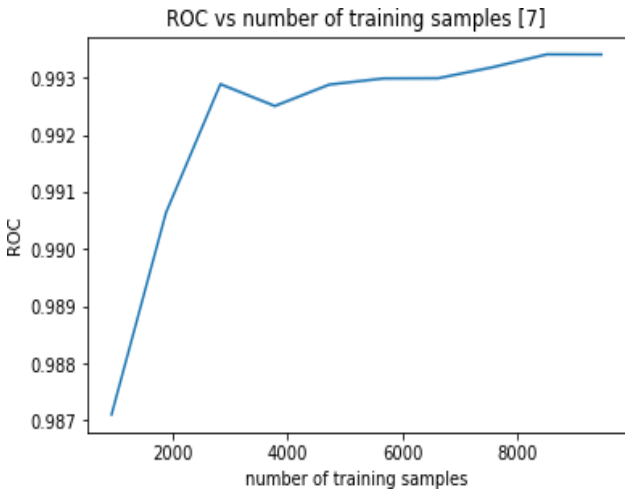


(b) Gaussian Differential Privacy Method

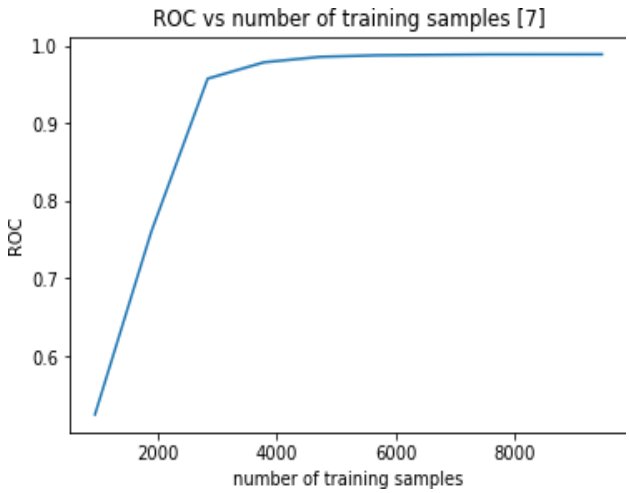


(c) Laplacian Differential Privacy Method

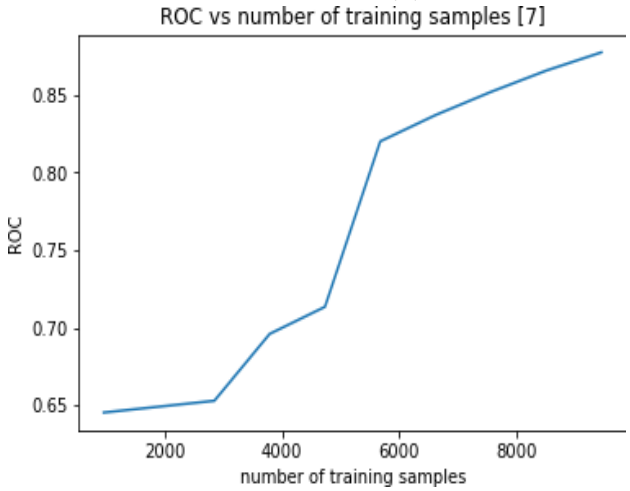
Figure 11: Classification accuracy vs number of sample sizes for epsilon=0.01 and delta=1e-5.



(a) Without Differential Privacy Method



(b) Gaussian Differential Privacy Method



(c) Laplacian Differential Privacy Method

Figure 12: ROC vs number of sample sizes for epsilon=0.01 and delta=1e-5.

```
def check_acc_roc_vs_num_samples_effect(add_x0=True, iterations=5000,
    learning_rate=0.5, regularization_coefficient=0.001,
                                    method='non_dp', epsilon=0.01, delta=1e
    -5, data_norm=False, norm_order=2):

     keep_train_percentage = [1, 0.9, 0.8, 0.7, 0.6, 0.5, 0.4, 0.3, 0.2, 0.1]
     all_number_training_samples=[];all_acc=[];all_roc=[];
     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(data_norm, norm_order)
          new_len = int(X_train.shape[0]*percentage)
          X_train = X_train[0:new_len,...]
          y_train = y_train[0:new_len]
          best_coefficients, _, _, _, _ = logistic_regression(
                                                  X_train, y_train,
                                                  X_validation, y_validation,
                                                  add_x0,
                                                  iterations,
                                                  learning_rate,
                                                  regularization_coefficient,
                                                  method,
                                                  epsilon,
                                                  delta,
```

```
                                                )

        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)
        roc_score = roc_auc_score(y_test, prediction_probabilites, 'macro')
        y_pred = prediction_probabilites >= 0.5
        acc = accuracy_score(y_test, y_pred)
        all_acc.append(acc)
        all_roc.append(roc_score)
        all_number_training_samples.append(new_len)
        time.sleep(1)

    all_acc = np.array(all_acc)
    all_roc = np.array(all_roc)
    all_number_training_samples = np.array(all_number_training_samples)
    idx = np.argmax(all_acc)
    print('Best acc achieved is: {} at num_samples: {}'.format(all_acc[idx],
    all_number_training_samples[idx]))
    idx = np.argmax(all_roc)
    print('Best roc achieved is: {} at num_samples: {}'.format(all_roc[idx],
    all_number_training_samples[idx]))

    os.makedirs('./logs/', exist_ok=True)
    # plot of 7 accuracy vs number of training samples
    plt.plot(all_number_training_samples, all_acc)
    plt.xlabel('number of training samples')
    plt.ylabel('classification accuracy')
    plt.title('classification accuracy vs number of training samples [7]')
    plt.savefig("./logs/{}
    _7_classification_accuracy_vs_number_of_training_samples.png".format(
    method), bbox_inches='tight', pad_inches=0)
    plt.show()

    os.makedirs('./logs/', exist_ok=True)
    # plot of 7 ROC vs number of training samples
    plt.plot(all_number_training_samples, all_roc)
    plt.xlabel('number of training samples')
    plt.ylabel('ROC')
    plt.title('ROC vs number of training samples [7]')
    plt.savefig("./logs/{}_7_roc_vs_number_of_training_samples.png".format(
    method), bbox_inches='tight', pad_inches=0)
    plt.show()

    log_dict = {
        'all_acc': all_acc,
        'all_roc': all_roc,
        'all_number_training_samples': all_number_training_samples,
        }
    np.save('./logs/{}_log_dict_6.npy'.format(method), log_dict)
if __name__ == '__main__':
    # fig 7 for non_dp, gaussian, laplace
    check_acc_roc_vs_num_samples_effect(iterations=5000, method='non_dp',
    data_norm=False)
    check_acc_roc_vs_num_samples_effect(iterations=5000, method='gaussian',
    data_norm=True, norm_order=2)
    check_acc_roc_vs_num_samples_effect(iterations=5000, learning_rate=5,
    regularization_coefficient=0.001, method='laplace', data_norm=True,
    norm_order=1)
```