CSED/AIGS 526: Data Mining

Your name: Jaeyoon Sim

Student ID: 20222421

Use late days quota(Yes/No): No

Problem 1 is a problem that implements my own Naive Bayes Classifier. To this end, we first created 1000 training instances and 400 testing instances.

```python
def make_trainig_data(num):
    X0_mu = [1, 0] # Mean 0
    X0_cov = [[1, 0.75], [0.75, 1]] # Covariance 0
    X0_train = np.random.multivariate_normal(X0_mu, X0_cov, num) # (num, 2)
    X0_train_label = np.zeros(num) # (num, )

    X1_mu = [0, 1.5] # Mean 1
    X1_cov = [[1, 0.75], [0.75, 1]] # Covariance 1
    X1_train = np.random.multivariate_normal(X1_mu, X1_cov, num) # (num, 2)
    X1_train_label = np.ones(num) # (num, )

    X_train = np.concatenate([X0_train, X1_train], axis=0) # (num * 2, 2)
    Y_train = np.concatenate([X0_train_label, X1_train_label], axis=0) # (num
* 2, )

    return X_train, Y_train

def make_testing_data(num):
    X0_mu = [1, 0] # Mean 0
    X0_cov = [[1, 0.75], [0.75, 1]] # Covariance 0
    X0_test = np.random.multivariate_normal(X0_mu, X0_cov, num) # (num, 2)
    X0_test_label = np.zeros(num) # (num, )

    X1_mu = [0, 1.5] # Mean 1
    X1_cov = [[1, 0.75], [0.75, 1]] # Covariance 1
    X1_test = np.random.multivariate_normal(X1_mu, X1_cov, num) # (num, 2)
    X1_test_label = np.ones(num) # (num, )

    X_test = np.concatenate([X0_test, X1_test], axis=0) # (num * 2, 2)
    Y_test = np.concatenate([X0_test_label, X1_test_label], axis=0) # (num *
2, )

    return X_test, Y_test
```

Listing 1: make training and testing data

In addition, we implemented my NB function that returns predicted labels, posterior probabilities, and error rate by receiving training data X, label Y, testing data X_test, and label Y_test as input. Naive Bayes Classifier was implemented as a method of giving the most likely label by comparing the posterior probability obtained by Bayes rule as follows.

$$\underset{h_k}{\operatorname{argmax}}\, P(h_k|x_1,\ldots,x_n) = \frac{P(h_k)\prod_{i=1}^{n} P(x_i|h_k)}{P(x_1,\ldots,x_n)} \tag{1}$$

```python
def myNB(X, Y, X_test, Y_test):
    pred = [] # Predicted labels
    posterior = [] # Posterior probability
    err = 0 # Error rate

    feature_num = X.shape[1] # The number of features

    labels_counter = Counter(Y) # Labels with the number of data - ex. {'0' :
500, '1' : 500} / {'G': 82, 'F': 49, 'G-F': 34, 'F-C': 26, 'C': 9}
    labels = list(labels_counter.keys()) # Labels - ex. (0.0, 1.0) / ('G', 'G-
F', 'F', 'F-C', 'C')
```

```
    labels_value = list(labels_counter.values()) # The number of data for labels - ex. [500, 500] / [34, 82, 59, 9, 26]

    label_num = len(labels) # The number of labels
    train_data_num = len(Y) # The number of train dataset
    test_data_num = len(Y_test) # The number of test dataset

    # Calculate prior probability
    p_prior = []
    for i in range(label_num):
        p_prior.append(labels_value[i] / train_data_num) # Prior probability : (# of features, )

    # Make dataframe for train data to make follow dictionary
    X_df = pd.DataFrame(X)
    Y_df = pd.DataFrame(Y)
    train_data_df = pd.concat([X_df, Y_df], axis=1)

    # Make dataframe for test data to make follow dictionary
    X_test_df = pd.DataFrame(X_test)
    Y_test_df = pd.DataFrame(Y_test)
    test_data_df = pd.concat([X_test_df, Y_test_df], axis=1)

    # Intialize dictionaries w.r.t. labels
    train_dict = {}
    test_dict = {}
    train_mu = {}
    train_std = {}
    for label in labels:
        train_dict.setdefault(label)
        test_dict.setdefault(label)
        train_mu.setdefault(label)
        train_std.setdefault(label)

    # Make dictionary for train data w.r.t. labels
    for label in labels:
        for i in range(train_data_num):
            if train_data_df.iloc[i,feature_num] == label:
                if train_dict[label] is None:
                    train_dict[label] = X_df.iloc[i,:]
                else:
                    train_dict[label] = pd.concat([train_dict[label], X_df.iloc[i,:]], axis=1)

    # Make dictionary for test data w.r.t. labels
    for label in labels:
        for i in range(test_data_num):
            if test_data_df.iloc[i,feature_num] == label:
                if test_dict[label] is None:
                    test_dict[label] = X_test_df.iloc[i,:]
                else:
                    test_dict[label] = pd.concat([test_dict[label], X_test_df.iloc[i,:]], axis=1)

    # Convert dataframe to list
    for label in labels:
        train_dict[label] = train_dict[label].transpose().values.tolist() # (# of train data, # of feature num)
        test_dict[label] = test_dict[label].transpose().values.tolist() # (#
```

```
of test data, # of feature num)

        # Calculate mean and standard deviation of train data w.r.t. labels
        for label in labels:
            train_mu[label] = np.mean(train_dict[label], axis=0)
            train_std[label] = np.std(train_dict[label], axis=0)

        # Naive Bayes Classifier
        for idx in range(test_data_num):
            p_likelihood = np.zeros((label_num, feature_num), dtype=np.float64)
            prod_p_likelihood = np.ones(label_num, dtype=np.float64)
            prior_likelihood = np.ones(label_num, dtype=np.float64)
            p_posterior = []

            data = X_test[idx, :]

            # Calculate likelihood probability
            for i, label in enumerate(labels):
                for j in range(feature_num):
                    p_likelihood[i, j] = norm.pdf(data[j], train_mu[label][j],
train_std[label][j])

            for i in range(label_num):
                for j in range(feature_num):
                    prod_p_likelihood[i] *= p_likelihood[i, j]
                prior_likelihood[i] = p_prior[i] * prod_p_likelihood[i]

            # Calculate evidence probability
            p_evidence = 1
            for i in range(label_num):
                p_evidence += prior_likelihood[i]

            # Calculate posterior probaiblity
            for i in range(label_num):
                p_posterior.append(prior_likelihood[i] / p_evidence)

            # Choose the label with highest posterior probability
            max_idx = np.argmax(p_posterior)
            pred.append(labels[max_idx])
            posterior.append(p_posterior[max_idx])

        err = 1.0 - accuracy_score(Y_test, pred) # Error rate

        return [pred, posterior, err]
```

Listing 2: my own Naive Bayes Classifier

The following is a function that calculates accuracy, precision, and recall in binary classification.

```
def perform_prediction(real, pred):
    TP = 0
    FP = 0
    TN = 0
    FN = 0

    for i in range(len(pred)):
        if pred[i] == 1 and real[i] == 1: # True positive
            TP += 1
        elif pred[i] == 1 and real[i] == 0: # False positive
```

```python
                FP += 1
            elif pred[i] == 0 and real[i] == 1: # False negative
                FN += 1
            elif pred[i] == 0 and real[i] == 0: # True negative
                TN += 1

        accuracy = (TP + TN) / (TP + FP + FN + TN) # Accuracy
        recall = TP / (TP + FN) # Recall
        precision = TP / (TP + FP) # Precision

        return accuracy, precision, recall
```

Listing 3: perform prediction on the testing data

In order to obtain meaningful results, the experiment was conducted a total of 10 times and then the average was taken to obtain the results.

```
[myNB with binary] Accuracy:0.925, Precision:0.929, Recall:0.920
[myNB with binary] Accuracy:0.945, Precision:0.954, Recall:0.935
[myNB with binary] Accuracy:0.963, Precision:0.960, Recall:0.965
[myNB with binary] Accuracy:0.963, Precision:0.947, Recall:0.980
[myNB with binary] Accuracy:0.953, Precision:0.941, Recall:0.965
[myNB with binary] Accuracy:0.958, Precision:0.964, Recall:0.950
[myNB with binary] Accuracy:0.955, Precision:0.933, Recall:0.980
[myNB with binary] Accuracy:0.938, Precision:0.949, Recall:0.925
[myNB with binary] Accuracy:0.932, Precision:0.918, Recall:0.950
[myNB with binary] Accuracy:0.945, Precision:0.959, Recall:0.930
[myNB with binary] Mean Accuracy:0.948, Mean Precision:0.946, Mean Recall:0.950
```

Figure 1: perform prediction on the testing data with myNB function

The following is a function that outputs the confusion matrix necessary to obtain accuracy, precision, and recall. The confusion matrix was constructed using the labels predicted by our model and actual labels.

```python
    def visualize_confusion_matrix(real, pred):
        cf_matrix = confusion_matrix(real, pred) # Confusion matrix

        # Visualize the confusion matrix
        ax = sns.heatmap(cf_matrix, annot=True, fmt='d', cmap='Blues')
        ax.set_title('Confusion Matrix\n', fontsize=16, fontweight='bold')
        ax.set_xlabel('Predicted')
        ax.set_ylabel('Actual')
        ax.xaxis.set_ticklabels(['class 0', 'class 1'])
        ax.yaxis.set_ticklabels(['class 0', 'class 1'])
        plt.show()
```

Listing 4: visualize confusion matrix

In addition, we implemented a function of drawing a scatter plot expressed samples corresponding to the same class in the same color.

```python
    def visualize_scatter_plot(data, pred):
        x1 = data[:,0] # (400, )
        x2 = data[:,1] # (400, )

        color_dict = {0:'red', 1:'blue'}

        # Visualiaze the scatter plot of data points whose labels are colore coded
     with prediction
        fig, ax = plt.subplots()
        ax.set_title("Scatter Plot of Data Points\n", fontsize=16, fontweight='
    bold')
        for c in np.unique(pred):
            idx = np.where(pred == c)
            ax.scatter(x1[idx], x2[idx], c = color_dict[c], label = c)
        ax.set_xlabel('x1')
        ax.set_ylabel('x2')
        ax.legend()
        plt.show()
```

Listing 5: visualize scatter plot of data points

A total of 10 experiments were conducted to output the results once. A confusion matrix and scatter plot were obtained for each result, but only the first to fourth results were inserted in the report.

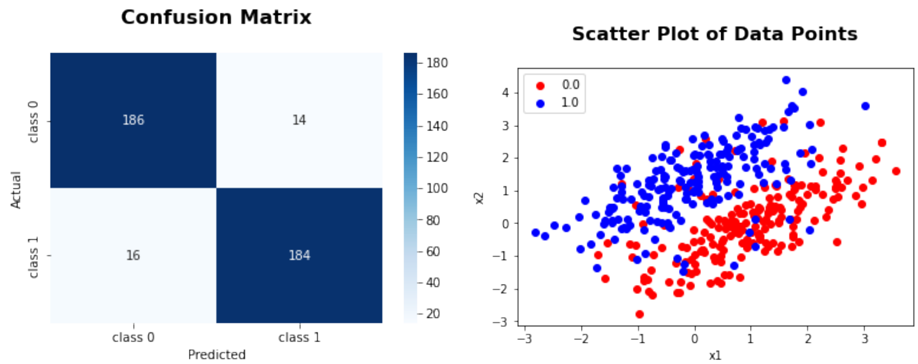

Figure 2: confusion matrix and scatter plot (1st experiment)

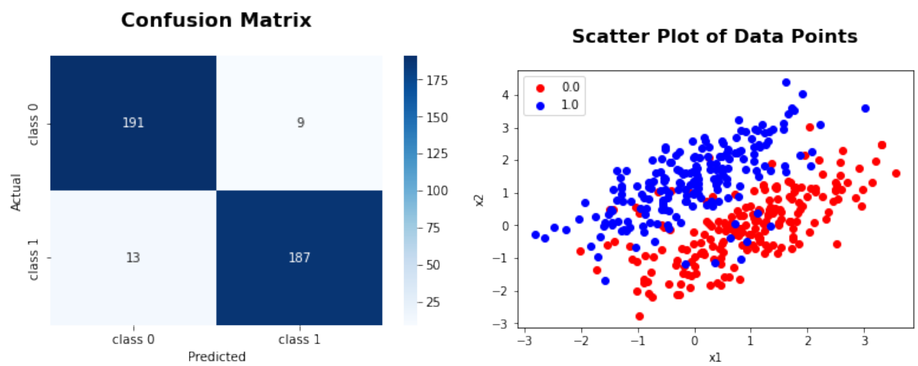

Figure 3: confusion matrix and scatter plot (2nd experiment)

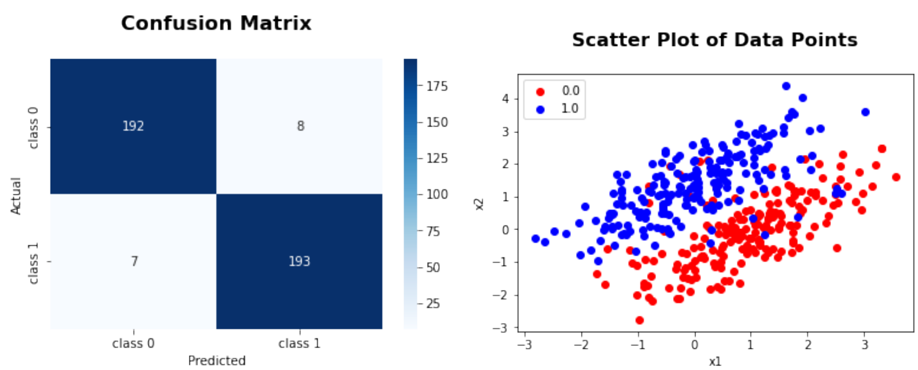

Figure 4: confusion matrix and scatter plot (3rd experiment)

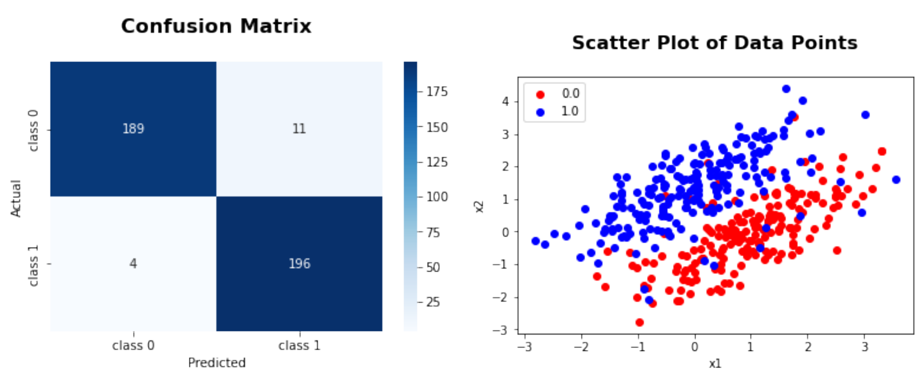

Figure 5: confusion matrix and scatter plot (4th experiment)

This time, the results were compared by changing the number of training samples. The function for this was implemented as follows. In the following function, the average result of a total of 10 experiments for each number of samples was used to see the change in accuracy.

```python
def change_training_data_and_plot_accuracies():
    training_samples = [10, 20, 50, 100, 300, 500]
    each_accuracy = [] # Accuracy for each # of samples
    each_precision = [] # Precision for each # of samples
    each_recall = [] # Recall for each # of samples
    all_accuracy = [] # Mean accuracy for each # of samples
    all_precision = [] # Mean precision for each # of samples
    all_recall = [] # Mean recall for each # of samples


    # Compute accuracy w.r.t. the number of training samples
    for num in training_samples:
        for i in range(10):
            X_train, Y_train = make_trainig_data(num)
            X_test, Y_test = make_testing_data(200)

            [pred, posterior, err] = myNB(X_train, Y_train, X_test, Y_test)

            accuracy, precision, recall = perform_prediction(Y_test, pred)

            each_accuracy.append(accuracy)
            each_precision.append(precision)
            each_recall.append(recall)

        mean_accuracy = np.mean(each_accuracy)
        mean_precision = np.mean(each_precision)
        mean_recall = np.mean(each_recall)

        all_accuracy.append(mean_accuracy)
        all_precision.append(mean_precision)
        all_recall.append(mean_recall)

        print(f"[myNB with {num} training samples] Mean Accuracy:{mean_accuracy:.3f}, Mean Precision:{mean_precision:.3f}, Mean Recall:{mean_recall:.3f}")

    font = {'family': 'Times New Roman',
            'color':  'blue',
            'weight': 'bold',
            'size': 12,
            'alpha': 0.8}

    # Show a plot of changes of accuracies w.r.t. the number of training samples
    plt.figure(figsize=(10,5))
    plt.title("Changes of Accuracies w.r.t. # of Samples\n", fontsize=16, fontweight='bold')
    plt.plot(training_samples, all_accuracy, marker='o', linestyle='--')
    plt.ylim([0.8, 1.0])
    plt.xlabel('# of training samples')
    plt.ylabel('Accuracy')
    plt.text(training_samples[0], all_accuracy[0], round(all_accuracy[0],3), fontdict=font)
    plt.text(training_samples[1], all_accuracy[1], round(all_accuracy[1],3), fontdict=font)
```

```python
        plt.text(training_samples[2], all_accuracy[2], round(all_accuracy[2],3), fontdict=font)
        plt.text(training_samples[3], all_accuracy[3], round(all_accuracy[3],3), fontdict=font)
        plt.text(training_samples[4], all_accuracy[4], round(all_accuracy[4],3), fontdict=font)
        plt.text(training_samples[5], all_accuracy[5], round(all_accuracy[5],3), fontdict=font)
        plt.show()
```

Listing 6: change training data and plot their accuracies

```
[myNB with 10 training samples] Mean Accuracy:0.865, Mean Precision:0.864, Mean Recall:0.875
[myNB with 20 training samples] Mean Accuracy:0.882, Mean Precision:0.875, Mean Recall:0.897
[myNB with 50 training samples] Mean Accuracy:0.899, Mean Precision:0.895, Mean Recall:0.908
[myNB with 100 training samples] Mean Accuracy:0.910, Mean Precision:0.906, Mean Recall:0.918
[myNB with 300 training samples] Mean Accuracy:0.915, Mean Precision:0.913, Mean Recall:0.921
[myNB with 500 training samples] Mean Accuracy:0.921, Mean Precision:0.919, Mean Recall:0.925
```

Figure 6: perform prediction on the testing data by changing the number of samples

When measuring the accuracy while changing the number of training samples, it can be seen that the overall accuracy improves as the number of training samples increases. It was confirmed that the accuracy was about 86% when there were 10 training samples, while the accuracy was easily over 90% when the number was increased to 500. This is because, as the number of training samples increases, more accurate distribution can be estimated, and better posteriors can be obtained based on this.

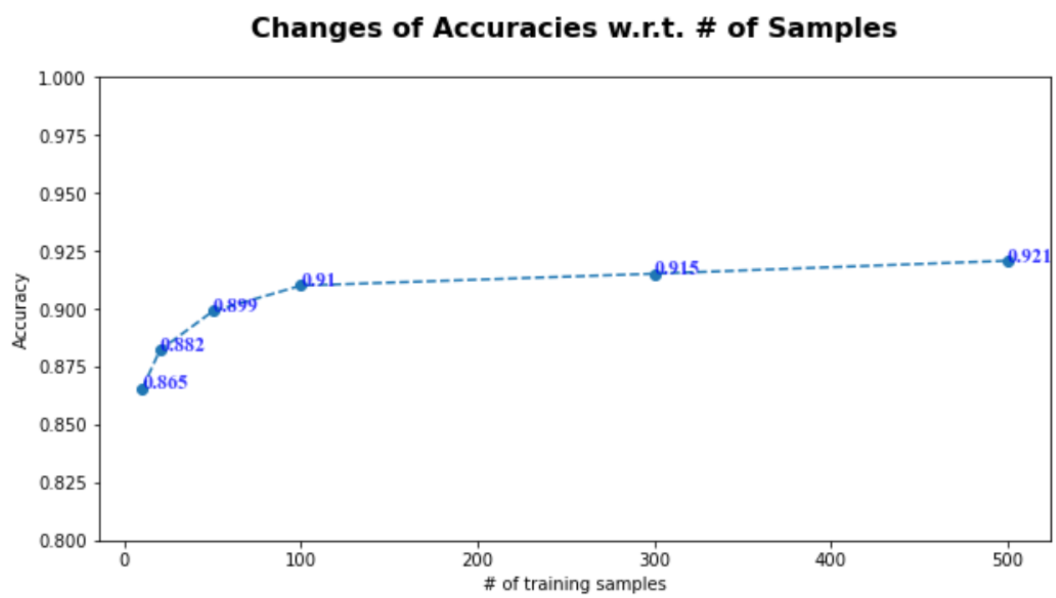

Figure 7: plot of changes of accuracies w.r.t. the number of samples

The following is the main function that can solve problem 1.1 using the functions written in this way.

```python
# ================= Problem 1.1 =================
all_accuracy = []
all_precision = []
all_recall = []
all_pred = []

for i in range(10):
    X_train, Y_train = make_trainig_data(500) # 500(0) + 500(1)
    X_test, Y_test = make_testing_data(200) # 200(0) + 200(1)

    [pred, posterior, err] = myNB(X_train, Y_train, X_test, Y_test) # My Naive Bayes Classfier

    accuracy, precision, recall = perform_prediction(Y_test, pred) # Perform prediction(accuracy, precision, recall)
    print(f"[myNB with binary] Accuracy:{accuracy:.3f}, Precision:{precision:.3f}, Recall:{recall:.3f}")

    all_accuracy.append(accuracy)
    all_precision.append(precision)
    all_recall.append(recall)
    all_pred.append(pred)

mean_accuracy = np.mean(all_accuracy, axis=0)
mean_precision = np.mean(all_precision, axis=0)
mean_recall = np.mean(all_recall, axis = 0)
print(f"[myNB with binary] Mean Accuracy:{mean_accuracy:.3f}, Mean Precision:{mean_precision:.3f}, Mean Recall:{mean_recall:.3f}")

for i in range(10):
    visualize_confusion_matrix(Y_test, all_pred[i])
    visualize_scatter_plot(X_test, all_pred[i])

change_training_data_and_plot_accuracies()
```

Listing 7: main function for ”problem 1.1”

This time, we actually applied it to the NBA dataset using our Naive Bayes Classifier made above. The difference from problem 1.1 was previously a binary classification that distinguished 0 and 1, but this problem 1.2 is a multi-class classification that distinguishes five positions. To this end, first, a function that creates data and labels from the 'nbastat2021.xlsx' file was implemented as follows.

```
def make_nba_data():
    file_name = 'nbastat2021.xlsx' # Input file
    nba_data = pd.read_excel(file_name, engine='openpyxl') # Input data with full-columns
    nba_data_9feat = nba_data.drop(['FULL NAME', 'TEAM', 'GP', 'MPG', 'POS'], axis =1) # Input data with pre-processing

    X_nba = nba_data_9feat.to_numpy() # (240, 9)
    Y_nba = nba_data['POS'].to_numpy() # (240, )

    return X_nba, Y_nba
```

Listing 8: make NBA data from 'nbastat2021.xlsx'

Since the conditions in question require a 6-fold classification using a total of 240 players, the function for 6-fold classification was created as follows.

```
def accuracy_6fold(X, Y):
    nba_data = np.concatenate([X, Y.reshape(len(Y), 1)], axis=1)  # (240, 10)
    nba_fold = KFold(n_splits=6, shuffle=False) # 6-fold

    i = 1
    all_accuracy = [] # All of accuracies

    # 6-fold cross validation (train:200 / test:40)
    for nba_train_index, nba_test_index in nba_fold.split(nba_data):
        X_train, Y_train = X[nba_train_index], Y[nba_train_index] # Train data of nba(200)
        X_test, Y_test = X[nba_test_index], Y[nba_test_index] # Test data of nba(40)

        [pred, posteriror, err] = myNB(X_train, Y_train, X_test, Y_test) # My Naive Bayes Classifier

        current_accuracy = accuracy_score(Y_test, pred) # Calculate accuracy
        all_accuracy.append(current_accuracy) # Append accuracy
        print(f"[myNB with NBA] Fold {i}) Accuracy:{current_accuracy:.3f}")
        i += 1

    mean_accuracy = np.mean(all_accuracy) # Mean of accuracy
    standard_deviation = np.std(all_accuracy) # Standard deviation of accuracy

    return mean_accuracy, standard_deviation
```

Listing 9: perform 6-fold classfication and report mean accuracy with standard deviation

The following is the main function that can solve problem 1.2 using the functions implemented in this way.

```
# ================= Problem 1.2 =================
X_nba, Y_nba = make_nba_data()

mean_accuracy, standard_deviation = accuracy_6fold(X_nba, Y_nba)
```

```
    print(f"[myNB with NBA] Mean Accuracy:{mean_accuracy:.3f}, Standard Deviation
    :{standard_deviation:.3f}")
```

Listing 10: main function for "problem 1.2"

As a result, the average of the accuracy output for each fold was taken and the standard deviation was output. The mean accuracy of 6-fold classification is about 0.567 and the standard deviation is 0.053.

```
[myNB with NBA] Fold 1) Accuracy:0.500
[myNB with NBA] Fold 2) Accuracy:0.625
[myNB with NBA] Fold 3) Accuracy:0.550
[myNB with NBA] Fold 4) Accuracy:0.550
[myNB with NBA] Fold 5) Accuracy:0.525
[myNB with NBA] Fold 6) Accuracy:0.650
[myNB with NBA] Mean Accuracy:0.567, Standard Deviation:0.053
```

Figure 8: perform 6-fold classification and report mean accuracy with standard deviation

Did it get better than clustering in the previous homework? Why?

K-means clustering can always satisfy convergence. However, we will not be able to guarantee that we have found a global optimal. Nevertheless, results close to the global optimal can be obtained. And in initialization, the center of the cluster is better as it is similar to the expected result, and it is better to set the k that we have to determine in the direction of reducing error if given our domain knowledge or supervision. K-means clustering creates a cluster based on distance and classifies it, so the performance is not that high. In addition, due to the problem of unsupervised learning, the performance was not very high because the label was not learned.

The Naive Bayes classifier is a conditional probabilistic model, with a strong assumption that each feature or attribute is independent of each other. Most attributes have correlation with each other, and in the case of Naive Bayes, all attributes have a conditionally independent assumption. Thanks to this reason, Naive Bayes can work well in reality. And we learn Naive Bayes using the maximum likelihood method, which requires relatively little training sets. The reason why Naive Bayes only needs a small training set is that this method uses a probabilistic model. And since distribution is used based on label information as supervised learning, the performance seems to be better than k-means.

Problem 2 is a problem that implements my own logistic regression. The same experiment was conducted with the same seed fixed. The overall result can be seen through the main function as follows. As we can see from the results afterwards, any seed will show good performance. So, it doesn't matter if we actually solve the seed and proceed with the experiment, but it was fixed and proceeded to include the same results in the report.

```
def main():
    seed_num = 0
    np.random.seed(seed_num)
    X_train, Y_train = make_trainig_data(500)
    X_test, Y_test = make_testing_data(200)

    learning_rate = [1, 0.1, 0.01] # Learning rate
    training_mode = ['batch', 'online'] # Training mode (Batch and Online)
    for lr in learning_rate:
        # ================= Problem 2.1 =================
        weight_batch, pred_batch, out_batch, loss_batch, accuracy_batch =
myLogisticRegression(X_train, Y_train, X_test, Y_test, lr, training_mode[0])
        plot_scatter_with_trained_decision_boundary(X_train, X_test,
weight_batch, pred_batch)
        plot_roc_curve(Y_test, out_batch, lr)
        # ================= Problem 2.2 =================
        weight_online, pred_online, out_online, loss_online, accuracy_online =
 myLogisticRegression(X_train, Y_train, X_test, Y_test, lr, training_mode[1])
        plot_scatter_with_trained_decision_boundary(X_train, X_test,
weight_online, pred_online)
        # ================= Problem 2.3 =================
        plot_changes_of_loss(loss_batch, loss_online, lr)
```

Listing 11: main function

The following is a function that creates training data and testing data. The characteristic here is that 0 and 1 are randomly assigned, so the data was constructed through shuffle, not just concatenation.

```
def make_trainig_data(num):
    X0_mu = [1, 0]
    X0_cov = [[1, 0.75], [0.75, 1]]
    X0_train = np.random.multivariate_normal(X0_mu, X0_cov, num) # (num, 2)
    X0_train_label = np.zeros(num) # (num, )

    X1_mu = [0, 1.5]
    X1_cov = [[1, 0.75], [0.75, 1]]
    X1_train = np.random.multivariate_normal(X1_mu, X1_cov, num) # (num, 2)
    X1_train_label = np.ones(num) # (num, )

    X_train = np.concatenate([X0_train, X1_train], axis=0) # (num * 2, 2)
    Y_train = np.concatenate([X0_train_label, X1_train_label], axis=0) # (num
* 2, )

    # Shuffle the train data
    XY_train = np.concatenate([X_train, Y_train.reshape(len(Y_train), 1)],
axis=1)
    np.random.shuffle(XY_train)

    X_train = XY_train[:,:-1]
    Y_train = XY_train[:,-1]

    return X_train, Y_train
```

```

    def make_testing_data(num):
        X0_mu = [1, 0]
        X0_cov = [[1, 0.75], [0.75, 1]]
        X0_test = np.random.multivariate_normal(X0_mu, X0_cov, num) # (num, 2)
        X0_test_label = np.zeros(num) # (num, )

        X1_mu = [0, 1.5]
        X1_cov = [[1, 0.75], [0.75, 1]]
        X1_test = np.random.multivariate_normal(X1_mu, X1_cov, num) # (num, 2)
        X1_test_label = np.ones(num) # (num, )

        X_test = np.concatenate([X0_test, X1_test], axis=0) # (num * 2, 2)
        Y_test = np.concatenate([X0_test_label, X1_test_label], axis=0) # (num *
   2, )

        # Shuffle the test data
        XY_test = np.concatenate([X_test, Y_test.reshape(len(Y_test), 1)], axis=1)
        np.random.shuffle(XY_test)

        X_test = XY_test[:,:-1]
        Y_test = XY_test[:,-1]

        return X_test, Y_test
```

Listing 12: make training and testing data

The following is an implementation of the activation function used in the logistic regression. We used sigmoid function here.

```
    def sigmoid_function(x):
        z = np.exp(-x)
        A = 1 / (1 + z)

        return A
```

Listing 13: sigmoid function

The following is an implementation of cross entropy loss. The characteristics here are that problem 2.1 and problem 2.2 are learned in different ways, so they are implemented separately accordingly.

```
    def cross_entropy_loss_batch(true, pred):
        N = len(true)
        loss = (-1) * (1 / N) * np.sum((true * np.log(pred)) + ((1 - true) * np.
   log(1 - pred)))

        return loss
```

Listing 14: cross entropy loss for batch training

```
    def cross_entropy_loss_online(true, pred):
        loss = (-1) * (true * np.log(pred) + (1 - true) * np.log(1 - pred))

        return loss
```

Listing 15: cross entropy loss for online training

The following is a function that proceeds with the logistic regression. Here, it is implemented in a function that allows learning to proceed differently according to the methods of batch training and online training. Each training technique was implemented along the pseudo code as follows.

given: network structure and a training set $D = \{(x^{(1)}, y^{(1)}), \cdots, (x^{(m)}, y^{(m)})\}$,
initialize all weights in $\boldsymbol{w}$ to small random numbers
until stopping criteria met do
    initialize the error $E(\boldsymbol{w}) = 0$
    for each $(\boldsymbol{x}^{(d)}, y^{(d)})$ in the training set
        input $\boldsymbol{x}^{(d)}$ to the network and compute output $o^{(d)}$
        increment the error $E(\boldsymbol{w}) = E(\boldsymbol{w}) + \frac{1}{2}\left(y^{(d)} - o^{(d)}\right)^2$
    calculate the gradient

$$\nabla E(\boldsymbol{w}) = \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \cdots, \frac{\partial E}{\partial w_n}\right]$$

    update the weights

$$\Delta \boldsymbol{w} = -\eta \, \nabla E(\boldsymbol{w})$$

Figure 9: pseudo code of batch training

given: network structure and a training set $D = \{(x^{(1)}, y^{(1)}), \cdots, (x^{(m)}, y^{(m)})\}$,
initialize all weights in $\boldsymbol{w}$ to small random numbers
until stopping criteria met do
    for each $(\boldsymbol{x}^{(d)}, y^{(d)})$ in the training set
        input $\boldsymbol{x}^{(d)}$ to the network and compute output $o^{(d)}$
        calculate the error $E(\boldsymbol{w}) = \frac{1}{2}\left(y^{(d)} - o^{(d)}\right)^2$
        calculate the gradient

$$\nabla E(\boldsymbol{w}) = \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \cdots, \frac{\partial E}{\partial w_n}\right]$$

        update the weights

$$\Delta \boldsymbol{w} = -\eta \, \nabla E(\boldsymbol{w})$$

Figure 10: pseudo code of online training

```python
def myLogisticRegression(X, Y, X_test, Y_test, learning_rate, mode):
    loss = [] # Loss values
    max_iteration = 10000 # Maximum iteration
    lr = learning_rate # Learning rate

    feature_num = X.shape[1] # The number of features : 2
    train_data_num = len(Y) # The number of train data : 1000
    test_data_num = len(Y_test) # The nubmer of test data : 400

    # Concatenate of data and label
    train_data = np.concatenate([X, Y.reshape(train_data_num, 1)], axis=1) #
```

```
(1000, 3)
    test_data = np.concatenate([X_test, Y_test.reshape(test_data_num, 1)], axis=1) # (400, 3)

    bias_train = np.ones((train_data_num,), dtype=np.int64)
    bias_test = np.ones((test_data_num,), dtype=np.int64)

    X_train = np.concatenate([bias_train.reshape(train_data_num, 1), X], axis=1) # (1000, 3)
    X_test = np.concatenate([bias_test.reshape(test_data_num, 1), X_test], axis=1) # (1000, 3)

    W = np.random.randn(feature_num, 1) * 0.01 # Initial weight : (2, 1)
    b = np.random.randn(1, 1) * 0.01 # Initial bias : (1, 1)
    W_b = np.concatenate([b, W], axis=0) # Initial weight with bias : (3, 1)
    #W_b = np.ones((3, 1), dtype=np.float64)

    if mode == 'batch': # Problem 2.1 - Batch training
        # Training
        previous_loss = 0
        iter_num = 0
        threshold = 1e-5
        for i in range(max_iteration):
            iter_num += 1
            gradient = [] # Gradient

            net = np.dot(X_train, W_b) # [X][W] : (1000, 1)
            net = net.flatten() # (1000, )
            out = sigmoid_function(net) # Activation function : (1000, )
            current_loss = cross_entropy_loss_batch(Y, out)

            # Calculate gradient
            for j in range(feature_num + 1):
                g_i = 0
                for k in range(train_data_num):
                    g_i += (1 / train_data_num) * (out[k] - Y[k]) * X_train[k][j]
                gradient.append(g_i) # (3, )

            # Update the weights
            for j in range(feature_num + 1):
                update_value = (-1) * lr * gradient[j]
                W_b[j, 0] += update_value

            # Termination conditions
            if np.abs(previous_loss - current_loss) < threshold:
                break;
            if np.linalg.norm(gradient) < threshold:
                break;

            previous_loss = current_loss
            loss.append(current_loss)

        # Test
        net = np.dot(X_test, W_b) # [X][W] : (1000, 1)
        net = net.flatten() # (1000, )
        out = sigmoid_function(net) # Activation function : (1000, )
        pred = np.where(out > 0.5, 1, 0)

```

```python
            accuracy = accuracy_score(Y_test, pred)
            print(f"[Batch  Training with lr={lr : <4}] Iteration : {iter_num : <5} Accuracy : {accuracy:.3f} Weight : {W_b.T}")
        elif mode == 'online': # Problem 2.2 - Online training
            # Training
            previous_loss = 0
            iter_num = 0
            threshold = 1e-5
            while True:
                flag = 0
                for d in range(train_data_num):
                    iter_num += 1
                    gradient = [] # Gradient

                    net = np.dot(X_train[d], W_b)
                    net = net.flatten()
                    out = sigmoid_function(net)
                    current_loss = cross_entropy_loss_online(Y[d], out[0])

                    # Calculate gradient
                    for j in range(feature_num + 1):
                        g_i = (out[0] - Y[d]) * X_train[d][j]
                        gradient.append(g_i) # (3, )

                    # Update the weights
                    for j in range(feature_num + 1):
                        update_value = (-1) * lr * gradient[j]
                        W_b[j, 0] += update_value

                    # Termination conditions
                    if np.abs(previous_loss - current_loss) < threshold:
                        flag = 1
                        break;
                    if np.linalg.norm(gradient) < threshold:
                        flag = 1
                        break;
                    if iter_num == 10000:
                        flag = 1
                        break;


                    previous_loss = current_loss
                    loss.append(current_loss)

                if flag == 1:
                    break;

            # Test
            net = np.dot(X_test, W_b) # [X][W] : (1000, 1)
            net = net.flatten() # (1000, )
            out = sigmoid_function(net) # Activation function : (1000, )
            pred = np.where(out > 0.5, 1, 0)

            accuracy = accuracy_score(Y_test, pred)
            print(f"[Online Training with lr={lr : <4}] Iteration : {iter_num : <5} Accuracy : {accuracy:.3f} Weight : {W_b.T}")
```

```python
        return W_b, pred, out, loss, accuracy
```

Listing 16: my logistic regression for batch / online training

The following is the result of learning with training set using logistic regression and verifying with testing set after fixing with seeds 0, 1, 2. We set the seed for the report, but it showed good performance even if it was random. In particular, batch and online learnings were conducted by changing the learning rate to 1, 0.1, and 0.01, and for each test data, the number of iterations to converge, the number of accuracy we got, and the edge weight that were learned were output.

```
[Batch  Training with lr=1   ] Iteration : 436   Accuracy : 0.963 Weight : [[-1.42254612 -4.72730219  4.88257847]]
[Online Training with lr=1   ] Iteration : 67    Accuracy : 0.973 Weight : [[-0.69338535 -4.33992083  4.31517816]]
[Batch  Training with lr=0.1 ] Iteration : 1457  Accuracy : 0.965 Weight : [[-1.03321363 -3.64551966  3.77878901]]
[Online Training with lr=0.1 ] Iteration : 1637  Accuracy : 0.970 Weight : [[-0.99862433 -3.84333582  3.9833383 ]]
[Batch  Training with lr=0.01] Iteration : 4061  Accuracy : 0.970 Weight : [[-0.58333432 -2.46929637  2.54215832]]
[Online Training with lr=0.01] Iteration : 7330  Accuracy : 0.968 Weight : [[-0.79291082 -2.99544307  3.08808836]]
```

Figure 11: logistic regression result with seed 0

```
[Batch  Training with lr=1   ] Iteration : 336   Accuracy : 0.958 Weight : [[-1.12735334 -4.25057968  4.34558155]]
[Online Training with lr=1   ] Iteration : 88    Accuracy : 0.905 Weight : [[-0.80039115 -4.89984159  2.85624374]]
[Batch  Training with lr=0.1 ] Iteration : 1289  Accuracy : 0.955 Weight : [[-0.89887958 -3.44867119  3.53786602]]
[Online Training with lr=0.1 ] Iteration : 1445  Accuracy : 0.960 Weight : [[-0.93063363 -3.6325731   3.67645427]]
[Batch  Training with lr=0.01] Iteration : 3910  Accuracy : 0.958 Weight : [[-0.57881103 -2.39946459  2.47229127]]
[Online Training with lr=0.01] Iteration : 2849  Accuracy : 0.960 Weight : [[-0.4853773  -2.11968672  2.21657587]]
```

Figure 12: logistic regression result with seed 1

```
[Batch  Training with lr=1   ] Iteration : 305   Accuracy : 0.983 Weight : [[-0.94484788 -4.06751003  4.14111258]]
[Online Training with lr=1   ] Iteration : 110   Accuracy : 0.938 Weight : [[ 0.0271637  -3.93847723  5.02453022]]
[Batch  Training with lr=0.1 ] Iteration : 1207  Accuracy : 0.980 Weight : [[-0.73108516 -3.35830825  3.41565412]]
[Online Training with lr=0.1 ] Iteration : 1459  Accuracy : 0.980 Weight : [[-0.64909469 -3.63913223  3.63483712]]
[Batch  Training with lr=0.01] Iteration : 3773  Accuracy : 0.980 Weight : [[-0.44842991 -2.38300624  2.42497275]]
[Online Training with lr=0.01] Iteration : 8344  Accuracy : 0.980 Weight : [[-0.62933933 -3.04820099  3.11258447]]
```

Figure 13: logistic regression result with seed 2

Compare the learned parameters and accuracy to the ones that you got from batch training. Are they the same?

Online training is mostly similar to batch training. In batch training, if an error is calculated using all samples and a gradient is obtained, in online training, a gradient is obtained as soon as an error is calculated with a sample. And then just update the weight. Therefore, the update process occurs as many as the number of samples in the training set. Overall, I think batch training and online training show similar accuracy. However, when the learning rate is the same, the two learning methods differ in terms of speed when the weight for loss converges.

The following is a function that implements the scatter plot and trained decision boundary of test data corresponding to both problems 2.1 and 2.2.

```
def plot_scatter_with_trained_decision_boundary(train_data, test_data, weight, pred):
    x1_test = test_data[:,0] # (400, )
    x2_test = test_data[:,1] # (400, )

    weight = weight.flatten() # (w_0, w_1, w_2)
    x1_train = train_data[:,0]
    x2_train = train_data[:,1]
    x1_min = np.min(x1_train)
    x1_max = np.max(x1_train)
    x2_min = np.min(x2_train)
    x2_max = np.min(x2_train)

    # Calculate the intercept and gradient of the decision boundary.
    c = (-1) * (weight[0] / weight[2])
    m = (-1) * (weight[1] / weight[2])

    # Calculate the decision boundary
    xd = np.array([x1_min, x1_max])
    yd = m * xd + c

    # Visualiaze the scatter plot of data points whose labels are colore coded with prediction
    color_dict = {0:'red', 1:'blue'}
    fig, ax = plt.subplots()
    ax.set_title("Scatter Plot of Test Data with Decision Boundary\n", fontsize=16, fontweight='bold')
    for c in np.unique(pred):
        idx = np.where(pred == c)
        ax.scatter(x1_test[idx], x2_test[idx], c = color_dict[c], label = c)
    ax.plot(xd, yd, 'k', lw=2, ls='--', label="decision boundary")
    ax.set_xlabel('x1', fontweight='bold')
    ax.set_ylabel('x2', fontweight='bold')
    ax.legend()
    plt.show()
```

Listing 17: plot scatter plot of test data with trained decision boundary

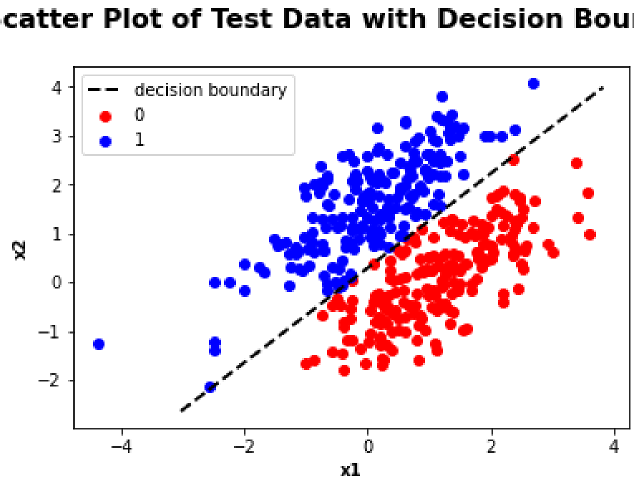

Figure 14: [batch training] plot scatter plot of test data with lr = 1

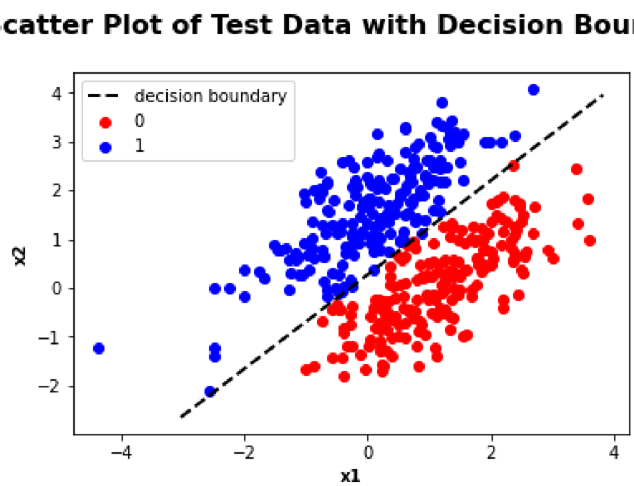

Figure 15: [batch training] plot scatter plot of test data with lr = 0.1

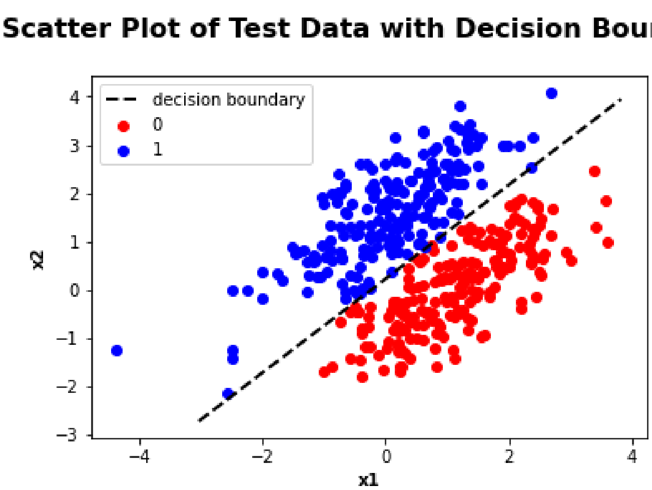

Figure 16: [batch training] plot scatter plot of test data with lr = 0.01

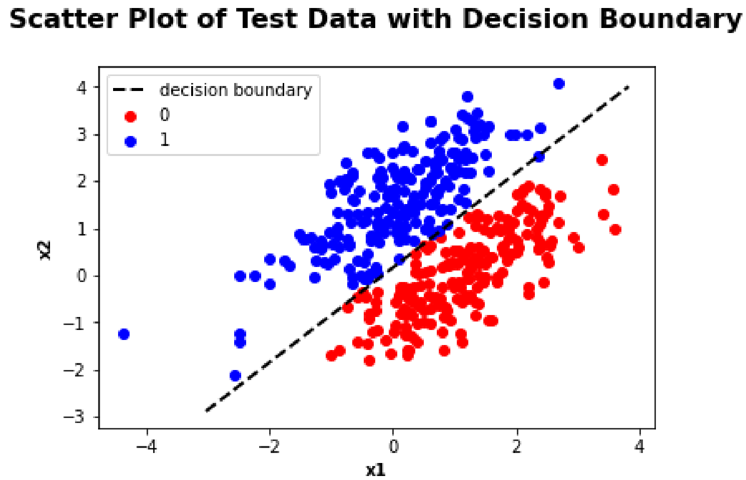

Figure 17: [online training] plot scatter plot of test data with lr = 1

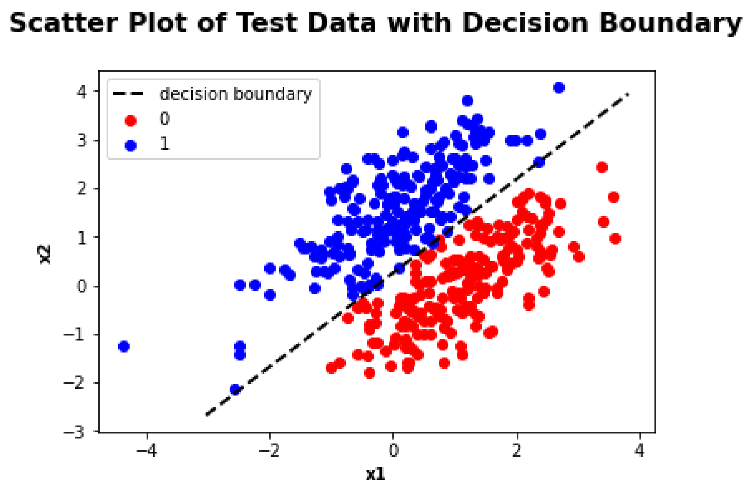

Figure 18: [online training] plot scatter plot of test data with lr = 0.1

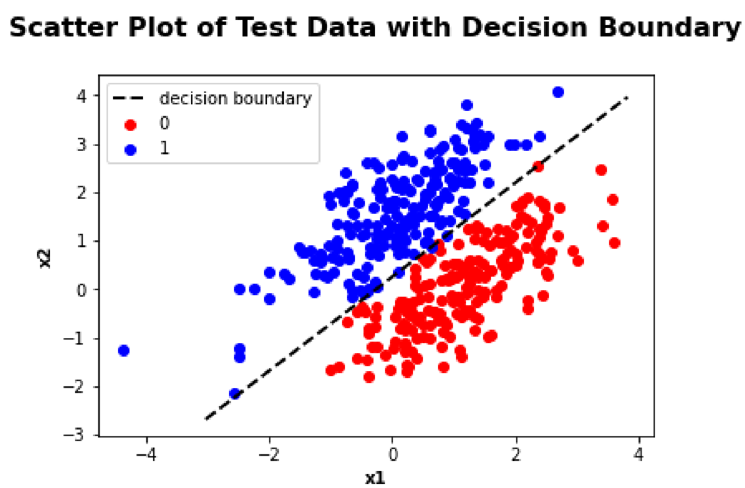

Figure 19: [online training] plot scatter plot of test data with lr = 0.01

The following is a function to plot the ROC curve corresponding to problem 2.1.

```
def plot_roc_curve(real, pred, lr):
    # Find the ROC curve by using built-in library
    fper, tper, thresholds = roc_curve(real, pred)

    # Plot the ROC curve
    plt.plot(fper, tper, color='red', label='ROC')
    plt.plot([0, 1], [0, 1], color='green', linestyle='--')
    plt.xlabel('False Positive Rate', fontweight='bold')
    plt.ylabel('True Positive Rate', fontweight='bold')
    plt.title(f"ROC Curve for Batch Training (lr={lr})\n", fontsize=16,
fontweight='bold')
    plt.legend()
    plt.show()
```

Listing 18: plot ROC curve

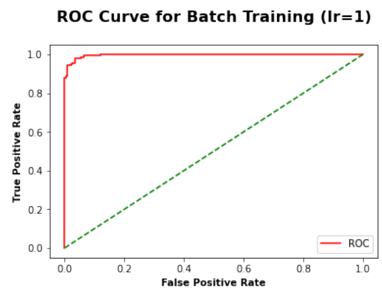

Figure 20: [batch training] plot ROC curve of batch training with lr = 1

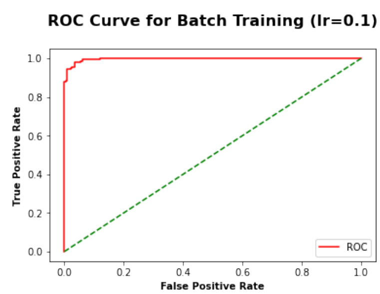

Figure 21: [batch training] plot ROC curve of batch training with lr = 0.1

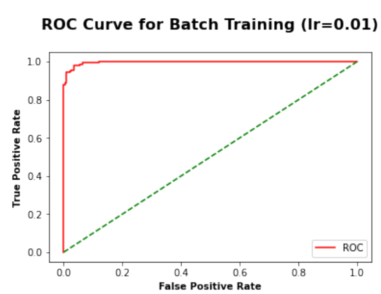

Figure 22: [batch training] plot ROC curve of batch training with lr = 0.01

The following is a function to plot changes of loss with respect to the number of iterations.

```
def plot_changes_of_loss(loss_batch, loss_online, learning_rate):
    loss_dict = {0:loss_batch, 1:loss_online}

    # Plot the changes of loss
    plt.subplots(1, 2, figsize=(12,5))
    plt.suptitle("Plot the changes of loss w.r.t. the # of itrations\n\n", fontsize=16, fontweight='bold')
    title_list = ['Batch Training', 'Online Training']
    for i in range(2):
        plt.subplot(1, 2, i+1)
        plt.plot(loss_dict[i], color='purple')
        plt.title(f"{title_list[i]} with lr={learning_rate}", fontsize = 12, fontweight='bold')
        plt.xlabel('The # of iterations', fontweight='bold')
        plt.ylabel('Loss', fontweight='bold')
    plt.show()
```

Listing 19: plot the changes of loss

Plot the changes of loss with respect to the number of iterations for gradient update for the above two cases and compare them. Briefly explain your observations.

Overall, it can be seen that batch training is more stable than online training. Obviously, batch training is based on all samples, while online training obtains gradients based on each sample, so we can find a phenomenon in which the loss value stands out. And there is a difference in the number of iterations depending on the size of the learning rate. It can be seen that the smaller the value of the learning rate, the longer the weight takes to converge.

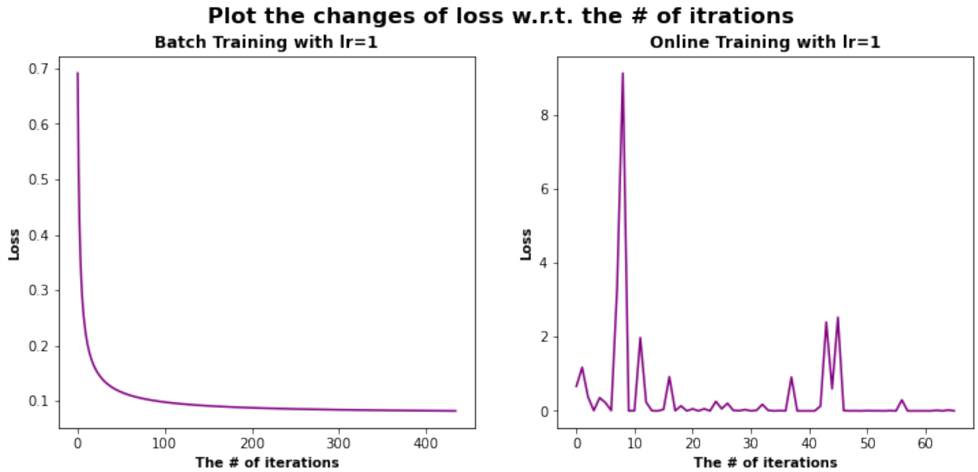

Figure 23: [batch / online] change of loss w.r.t. the number of iterations with lr = 1

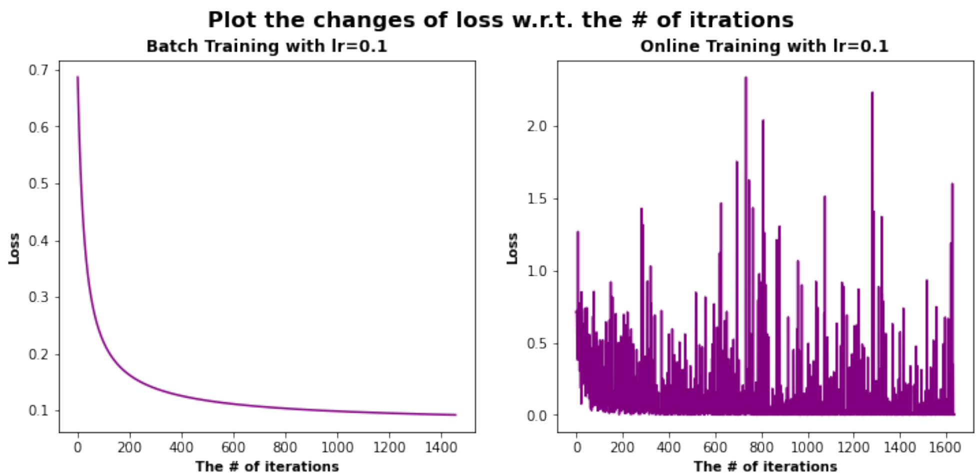

Figure 24: [batch / online] change of loss w.r.t. the number of iterations with lr = 0.1

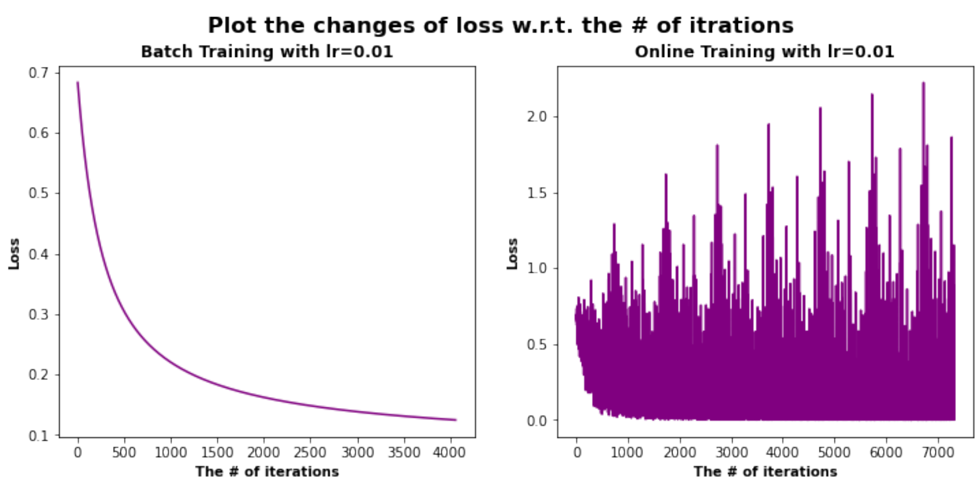

Figure 25: [batch / online] change of loss w.r.t. the number of iterations with lr = 0.01