Polish Companies Bankruptcy Classification

Dan-Refining Masters

119020071 Haotian Zhai

119020012 Yuxin Guo

119020054 Xinyang Wang

SPRING || 2022

FTE 4560

Basic Machine Learning

Contents

1 Introduction

1.1 Data Description

The Polish bankruptcy data set contains information of the companies. The total data set is split into a training set with 949 samples and a testing set with 151 samples. There are 64 financial features and an indicator on whether those companies went bankruptcy. Class label 0 indicates that the company is ”not bankrupted” and 1 indicates ”bankrupted”. There are many extreme values in the companies’ financial features, which do not follow Gaussian distribution. Besides, the number of samples of ”not bankrupted” is much larger than that of ”bankrupted”, so there exists a data imbalance problem in the data set, which can be seen from Figure 1.

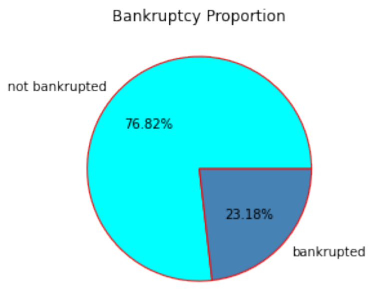

Figure 1: Imbalanced Data Set

1.2 Data Preprocessing

1.2.1 Dealing with Missing Data

There are some missing values in the data set. Attributes that contain missing values and the missing percentages are displayed in Figure 2.

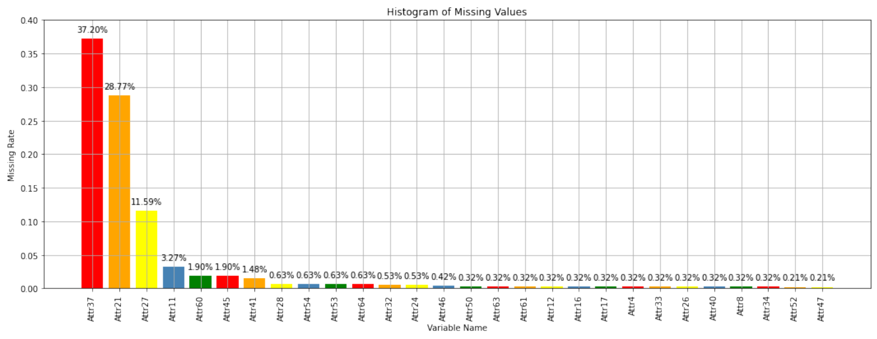

Figure 2: Imbalanced Data Set

It can be seen that missing rates of all attributes are within 40%, three of which are much higher than the others. There are many extreme values in those attributes, which can make the mean highly skewed. Based on Figure 3, it is obvious that the distribution of original data does not follow Gaussian distribution. As a result, to add more flexibility to the missing value imputation, we choose values around the median instead of a fixed median value to fill those missing data.

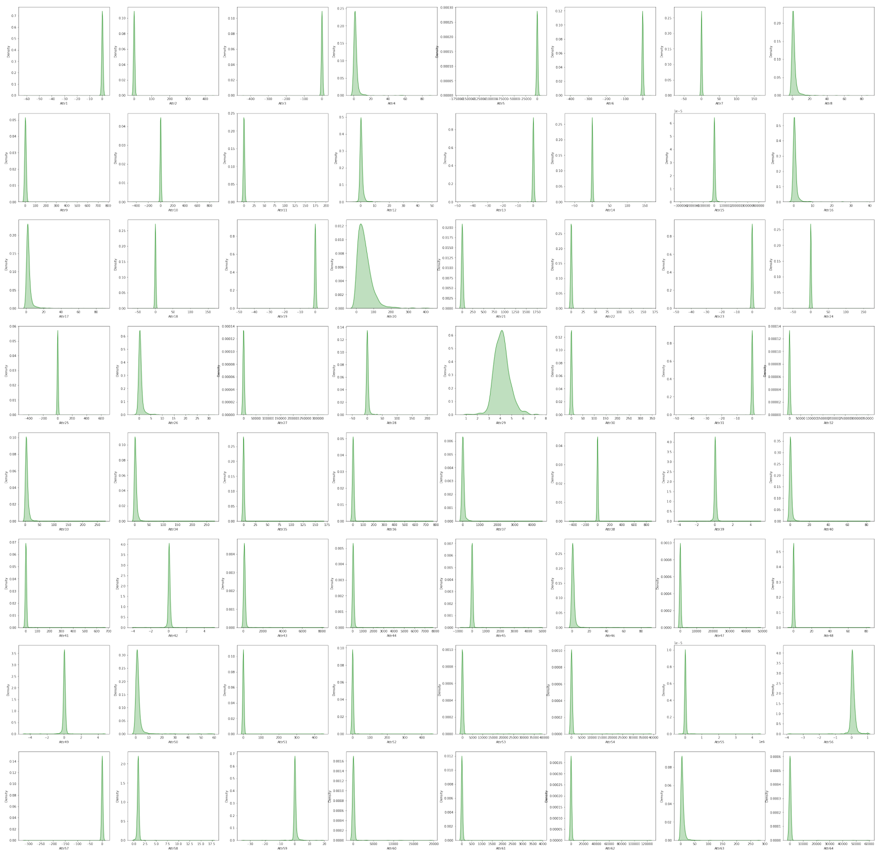

Figure 3: Density Curve of All Features

1.2.2 Data Normalization

Apply z-score normalization to the data set, which makes the mean of each attribute 0 and variance 1. Formulas to do the normalization is displayed as below:

$$\tilde{x}_{ij} = \frac{x_{ij} - \overline{x}_i}{s_i} \tag{1}$$

$$\overline{x}_i = \frac{1}{n} \sum_{j=1}^{n} x_{ij} \tag{2}$$

$$s_i = \frac{1}{n} \sum_{j=1}^{n} (x_{ij} - x_i)^2 \tag{3}$$

1.2.3 Feature Selection

Since the number of features is too large, this may cause overfitting problem or make the data matrix singular, we use a simple random forest (random forest classifier will be talked later) to calculate the weight of each feature, part of which is displayed in Figure 4:

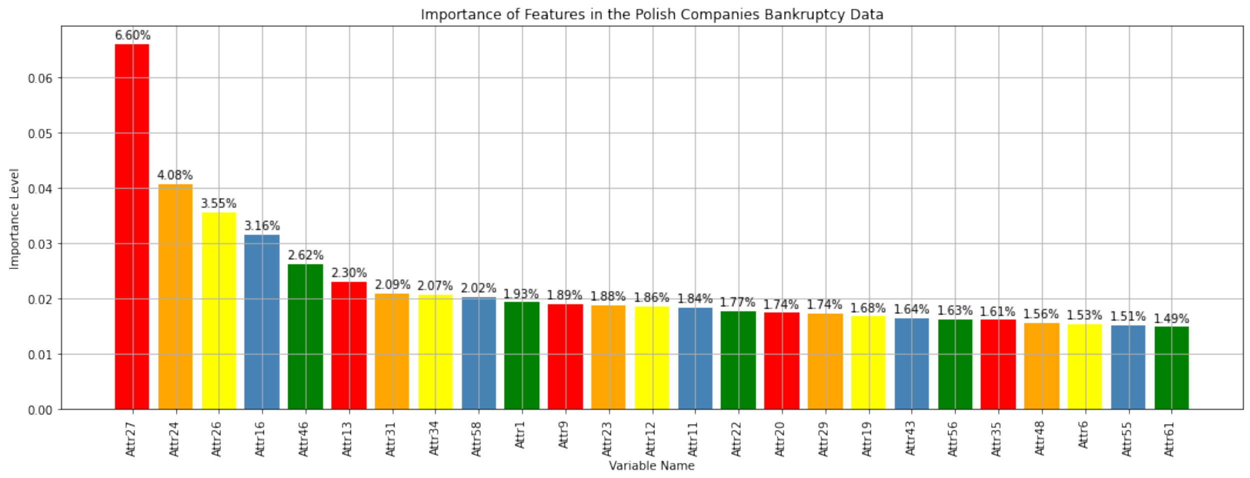

Figure 4: Feature Weights

Considering the feature selection based on random forest, Disha,R. A. has pointed out that choosing 40% of all features is generally accepted [**1**]. We follow the similar criterion and select 25 features which have higher weights.

The heat map of original features are displayed in Figure 5 while the heat map of selected features are displayed in Figure 6. As the picture tells. our feature selection has ensured the relatively low correlation between different features.

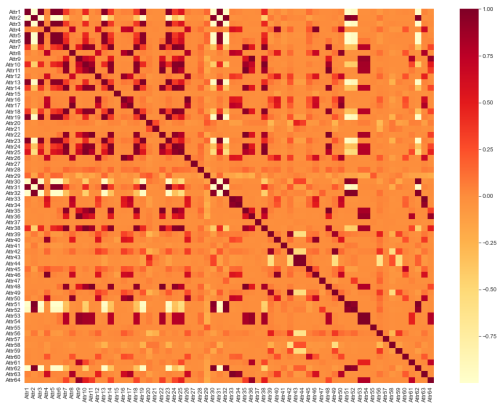

Figure 5: Feature Weights

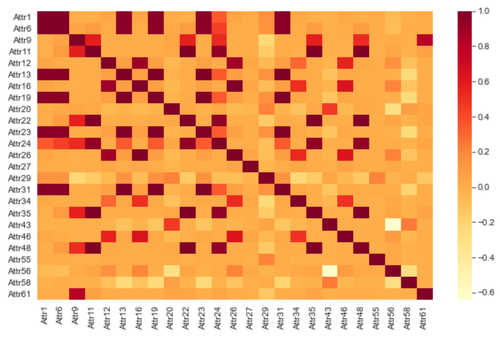

Figure 6: Feature Weights

1.3 Models with Different Data Sets

Based on requirements of different models, we perform data preprocessing with different procedures:

- Missing data imputation: decision tree, random forest, neural network
- Missing data imputation and normalization: KNN, classifiers with regularization (L2), least squares classifier with Gaussian basis functions
- Missing data imputation, normalization, and feature selection: LDA, classifiers without regularization (L2)

1.4 Evaluation Metrics

Symbols used in the following discussions are listed in Table 1:

Symbol	Definition
TP	True Positive: class label is 1 and classified as 1
TN	True Negative: class label is 0 and classified as 0
FP	False Positive: class label is 0 and classified as 1
FN	False Negative: class label is 1 and classified as 0
TPR	True Positive Rate
FPR	False Positive Rate
ROC	Receiver Operating Characteristic

Table 1: Notations

Metrics can be computed as below:

$$precision = \frac{TP}{TP+FP} \tag{4}$$

$$recall = \frac{TP}{TP+FN} \tag{5}$$

$$F1\ score = 2 \cdot \frac{precision \cdot recall}{precision + recall} \tag{6}$$

$$accuracy = \frac{TP+TN}{TP+FP+TN+FN} \tag{7}$$

$$TPR = recall = \frac{TP}{TP+FN} \tag{8}$$

$$FPR = \frac{FP}{FP+TN} \tag{9}$$

Precision is the percentage of true positive samples among samples which are predicted as class 1. *Recall* is the percentage of true positive samples which are actually in class 1. $F1$ *score* is a harmonic average of precision and recall.

ROC is receiver operating characteristic curve, of which the x-axis is the false positive rate and y-axis is the true positive rate. *AUC* is the area under the *ROC* curve. The higher *AUC* is, the better the model can perform.

Comparing $F1$ *score* and *AUC*, we can find that they both concentrate on samples in class 1. Besides, *AUC* tries to reduce false positive rate, but $F1$ *score* tries to improve the proportion of true samples among positive ones. Both two metrics are important for classification. Since Carrington, A. M. pointed out that *AUC* could work well with unbalanced data [**2**], we will evaluate models based on *AUC* first. If models are judged to have similar *AUCs*, we will choose the one with higher $F1$ *score*.

To better understand *AUC*, we implement the calculation procedure manually:

```
# AUC approximation
integral_list = []
for i in range(10):
    xlist = FPR_df[i]
    ylist = TPR_df[i]
    xpoints = sorted(list(set(xlist)))
    ypoints = []
    for x in xpoints:
        yindex = xlist.index(x)
        ypoints.append(ylist[yindex])
    integral = 0
    for j in range(len(xpoints)-1):
        integral += 1/2*(ypoints[j]+ypoints[j+1])*(xpoints[j+1]-xpoints[j]
            )
    integral_list.append(integral)
print(integral_list)
```

We can also try to calculate *AUC* by the existing package:

```
# AUC
from sklearn.metrics import roc_curve, auc
FPR, TPR, threshold = roc_curve(y_true, y_score, pos_label)
roc_auc = auc(FPR, TPR)
```

In the function *roc_curve*(), parameter *y_true* is the true label, *y_score* is the probability of the sample in the positive class, and *pos_label* is the positive class label. According to FPR and TPR, we can derive *AUC* based on function *auc*().

It should be noted that for models with hyperparameters, for example λ, metrics are computed by using 10-fold cross validation, because this method can reduce the possibility of overfitting and approach the real result of test data set.

2 Data Models

We use following models in the analysis of Polish companies bankruptcy rate:

- KNN classifier with different k
- Least squares classification with/without regularization (L2)
- Least squares classification with Gaussian basis functions
- Linear discriminant analysis (LDA)
- Logistic regression with/without regularization (L2)
- Softmax regression with/without regularization (L2)
- Decision tree
- Random forest
- Neural network

2.1 KNN Classifier

In the code below, for a specific *k*, we perform 10-fold cross validation to calculate the probability that one sample is in class 1.

```
def knn_train(trainX, trainY, k):
    Prob = []
    folds = 10
    for i in range(0, folds):
        for j in range(0, vali_x.shape[0]):
            dis = np.sqrt((diff**2).sum(axis=1))
            sort_dis = sorted(dis_dict.items(), key=lambda item:item[1])
            neigh = trainY["class"][[a[0] for a in sort_dis[:k]]]
            true = trainY["class"][vali_x.iloc[[j]].index[0]]
            num = neigh_l.count(1)
            ratio = num/k
            Prob.append([ratio,true])
    return(Prob)
```

Then, we choose different parameters k = 1, 3, 5, 7, 9, 11 (we do not choose even numbers to avoid a tie) and calculate metrics for each element. The parameter k here denotes the number of neighbours that the test point should choose.

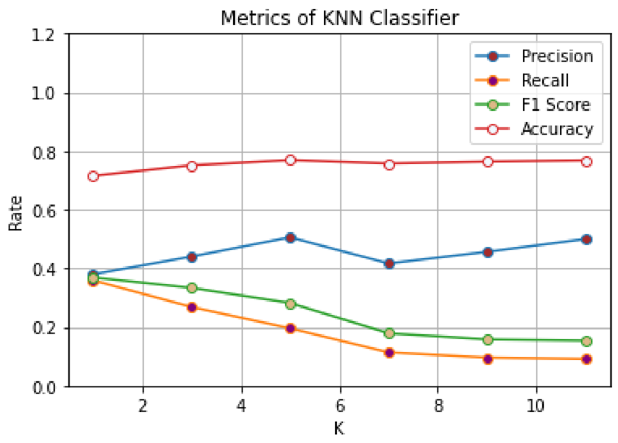

Figure 7: Metrics of KNN Classifier

Metrics of KNN classifier are displayed in Figure 7.

ROC curve of KNN classifier is displayed in Figure 8. It can be seen that F1-score is decreas-

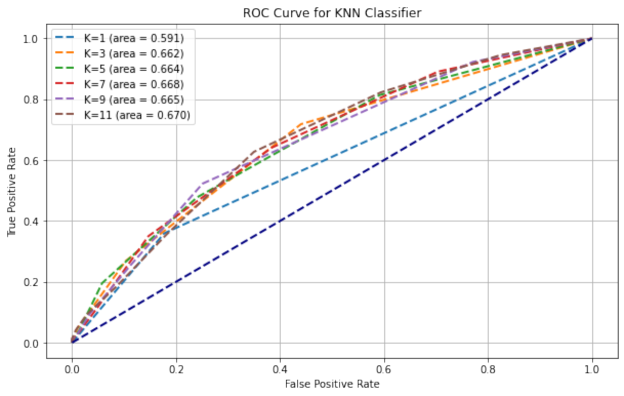

Figure 8: ROC Curve for KNN Classifier

ing when k gets larger. AUC has a rapid increase when $k = 3$ and does not increase much when k gets larger. As a result, we choose 3 as the optimal k and use it to predict the test data.

```
def knn_test(trainX, trainY, testX, testY, k=3):
    row = testX.shape[0]
    Prob = []
    for i in range(row):
        sample = testX.iloc[[i]]
        diff = np.tile(sample, (trainX.shape[0], 1)) - trainX
        dis = np.sqrt((diff**2).sum(axis=1))
        neigh = trainY["class"][[a[0] for a in sort_dis[:k]]]
        true = testY["class"][i]
        num = neigh_l.count(1)
        ratio = num/k
    return(Prob)
```

Metrics of the test data when $k = 3$ are displayed in Table 2.

Accuracy	**Precision**	**Recall**	**F1-score**
0.7086	0.5813	0.4901	0.5319

Table 2: Test Result for KNN Classifier

ROC curve of the test data when $k = 3$ are displayed in Figure 9.

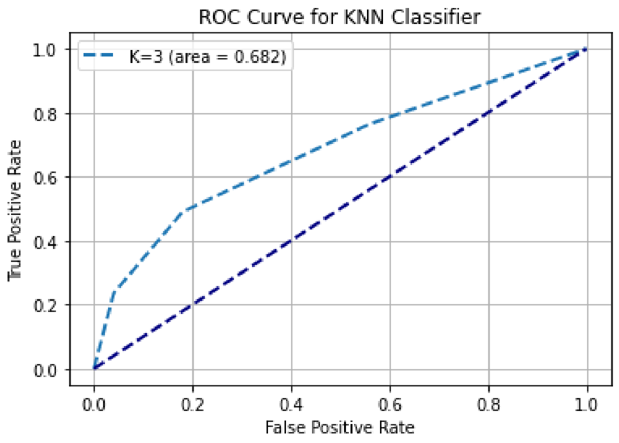

Figure 9: Test ROC Curve for KNN Classifier

2.2 Least Squares Classifier

2.2.1 Least Squares Classification

First, we perform data transformation to change data into the required format, and then we make predictions on two classes. Assign the class with higher probability to the sample.

```
# Least squares without regularization
def LS(trainX, trainY):
    W = np.dot(np.dot(trainY, trainX.T), np.linalg.inv(np.dot(trainX,
                                                  trainX.T)))
    prob = np.dot(W, trainX).T
    pred = [int(i[1]>i[0]) for i in prob]
    label = [list(i).index(1) for i in trainY.T]
    precision = TP/(TP+FP)
    recall = TP/(TP+FN)
    F1 = 2*precision*recall/(precision+recall)
    accuracy = (TP+TN)/(TP+TN+FP+FN)
    return((precision,recall,F1,accuracy))
```

Metrics on the train and test data are displayed in Table 3.

Data	Accuracy	Precision	Recall	F1-score
Train	0.7787	0.6923	0.0818	0.1463
Test	0.5364	0.3733	0.5490	0.4444

Table 3: Results for LS Classifier

During the training process, we find that least squares without regularization may lead to the overfitting problem because the parameter matrix W has many extreme values, some of which even exceed the limit computer can handle. As a result, there are many extreme values in the predicted probability, which makes it impossible to scale the range to [0,1] and no ROC curve can be drawn.

2.2.2 Least Squares Classification with Regularization (L2)

In the code below, we perform 10-fold cross validation for a specific λ. Generally, the larger λ is, the smaller parameters will be. In each iteration, we get the probability of class 1, the predicted class, and the true class.

```
# Least squares with regularization
def LS_regu(trainX, trainY, para):
    # Divide indices of data into 10 folds randomly
    index0 = [i for i in range(len(trainY[0])) if trainY[0][i]==1]
    index1 = [i for i in range(len(trainY[1])) if trainY[1][i]==1]
    # Use 10-fold cross validation to estimate the error rate
    folds = 10
    for i in range(0, folds):
        # For each fold, calculate the accuracy
        # Train set has 9 folds and validation set has 1 fold
        if np.linalg.det(np.dot(train_x, train_x.T)) != 0:
            left = np.dot(train_y, train_x.T)
            right = np.dot(train_x, train_x.T)+diagonal
            W = np.dot(left, np.linalg.inv(right))
            prob = np.dot(W, vali_x).T
            pred = [int(i[1]>i[0]) for i in prob]
    return(np.array(Prob),np.array(Pred),np.array(L))
```

Metrics of the regularized least squares classifier are displayed in Figure 10.

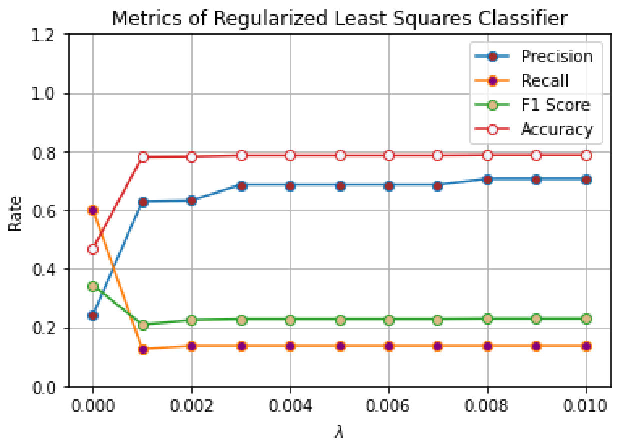

Figure 10: Metrics of Regularized Least Squares Classifier

ROC of the regularized least squares classifier are displayed in Figure 11.

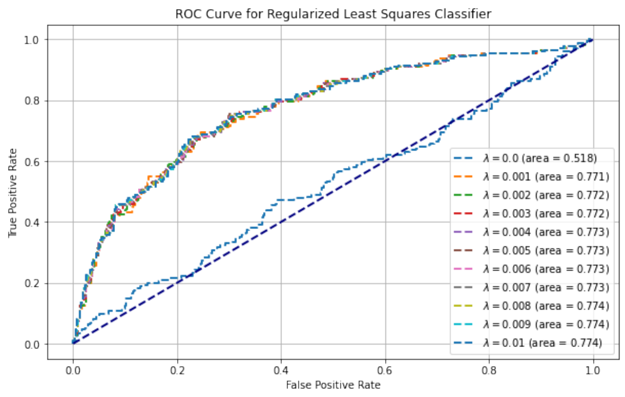

Figure 11: ROC Curve for Regularized Least Squares Classifier

From the pattern of ROC curve, it can be seen that adding a regularization term contributes a lot to the AUC performance, combined with F1 score, we choose 0.02 as the optimal λ to predict the test data.

```
def LS_test(trainX, trainY, testX, testY, para=0.02):
    left = np.dot(trainY, trainX.T)
    right = np.dot(trainX, trainX.T)+diagonal
    W = np.dot(left, np.linalg.inv(right))
    prob = np.dot(W, testX).T
    list = [list(i).index(1) for i in testY.T]
    return(np.array(Prob),np.array(Pred),np.array(L))
```

Test metrics of regularized least squares are displayed in Table 4.

Accuracy	**Precision**	**Recall**	**F1-score**
0.5894	0.4367	0.7450	0.5507

Table 4: Test Result for LS Classifier with L2

Test ROC curve of regularized least squares is displayed in Figure 12.

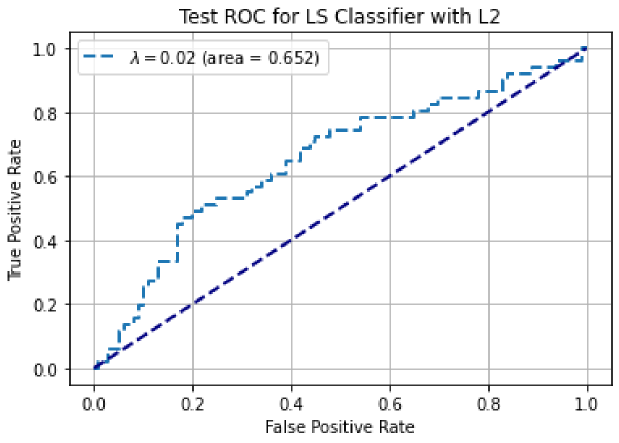

Figure 12: Test ROC for LS Classifier with L2

After adding the L2 regularization term, we find that all parameters are scaled to a relatively bounded range instead of infinity, which makes the model more robust.

2.3 Least Squares Classification with Gaussian Basis Functions

We calculate the mean and variance of each attribute as the parameter for Gaussian distribution. Based on the distribution, we generate 100 Gaussian variables. For the covariance matrix, it is calculated as the average of L2 norm of all data. Finally, we transform X by using the Gaussian basis function to $\phi(X)$.

```
# Data transformation
np.random.seed(1)
row = trainDX.shape[0]
col = trainDX.shape[1]
meanMatrix = np.random.normal(0,1,size=(100,col))
kernelX = np.zeros((row, 100))
sigma = np.diag([np.sqrt((trainDX**2).sum(axis=1)).mean()]*col)
a = 1/(2*math.pi)**(col/2)
b = 1/(np.linalg.det(sigma))**0.5
```

```
for i in range(kernelX.shape[0]):
    sample = np.array(trainDX.iloc[i])
    for j in range(kernelX.shape[1]):
        mean = meanMatrix[j]
        c = np.dot((sample-mean).T,np.linalg.inv(sigma))
        d = sample-mean
        kernelX[i][j] = a*b*np.exp(-0.5*np.dot(c,d))
```

After training the model, the train and test metrics are showed in Table 5.

Data	Accuracy	Precision	Recall	F1-score
Train	0.8229	0.7203	0.3863	0.5029
Test	0.6754	0.5357	0.2941	0.3797

Table 5: Results for LS Classifier with Gaussian Basis Function

ROC curves for train and test data are drawn in Figure 13.

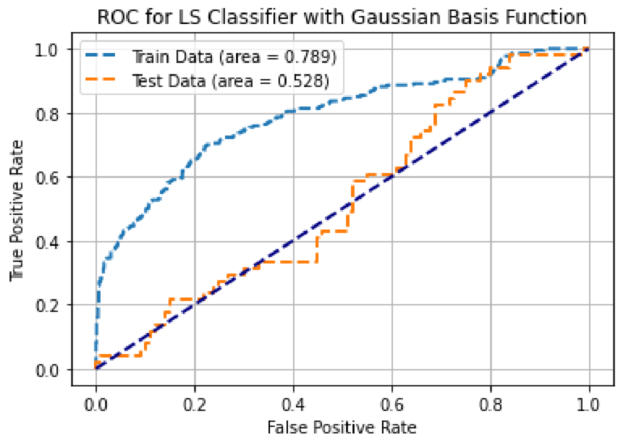

Figure 13: ROC for LS Classifier with Gaussian Basis Function

From those metrics, we find that least squares regression with Gaussian basis function can perform well on the train data, but the test results are not as good as expected.

2.4 Linear Discriminant Analysis

First, we define a function for projection to calculate the within-class scatter matrix, the between-class scatter matrix, the eigenvector corresponding to the largest eigenvalue, and the mean after projection, which are parameters of Linear Discriminant Analysis later.

Then we perform 10-fold cross validation for LDA classifier. Since LDA is sensitive to the distribution of labels in data set, we use cross validation method to make up for the possible bias. Note that in this method, feature selection has been finished during data pre-processing to prevent from singular matrix calculation error. For each LDA classifier in cross validation, we calculate precision, recall, F1-score, and accuracy. Besides, we obtain the parameters of each LDA classifier.

Matrices of the Linear Discriminant Analysis classifier (10-fold validation) are displayed in Figure 14.

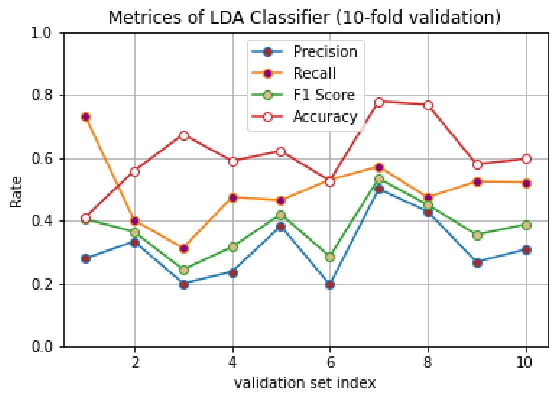

Figure 14: Metrics of Linear Discriminant Analysis

According to above image, we can see that the different LDA classifiers' performances vary a little. Therefore, from those metrics, We take arithmetic mean of 10-fold cross validation parameters, and choose it as the parameter of final model to make predictions on the test data. In this way, can the algorithm be more robust to the distribution of the labels in data set.

```python
def LDA_test(pred, test_x, test_y):
    TP = len([j for j in range(len(pred)) if np.array(pred)[j]==1 and
                                           test_y[j]==1])
    FP = len([j for j in range(len(pred)) if np.array(pred)[j]==1 and
                                           test_y[j]==0])
    TN = len([j for j in range(len(pred)) if np.array(pred)[j]==0 and
                                           test_y[j]==0])
    FN = len([j for j in range(len(pred)) if np.array(pred)[j]==0 and
                                           test_y[j]==1])
    Precision = TP / (TP+FP)
    Recall = TP / (TP+FN)
    F1score = 2*Precision*Recall / (Precision+Recall)
    Accuracy = (TP+TN) / (TP+TN+FP+FN)
    return Precision, Recall, F1score, Accuracy
```

ROC curves of LDA Classifier for test data are drawn in Figure 15.

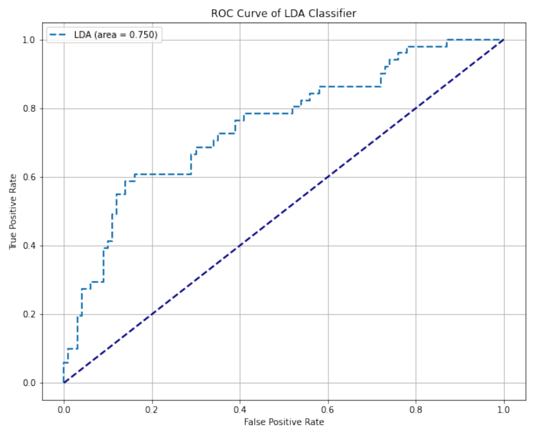

Figure 15: ROC for LDA Classifier

Data	Accuracy	Precision	Recall	F1-score
Train	0.6165	0.3245	0.5326	0.3763
Test	0.6689	0.5386	0.7255	0.5968

Table 6: Results for Linear Discriminant Analysis Classifier

Metrics of LDA classifier for train data and test data are summarized in Table 6. We can see that test set's performance is much better than train set's. From this can we conclude that the method of ”taking-arithmetic-mean-of-cv-parameters” makes sense. The outperformance of test set may be because the test data is much further away from the decision boundaries, which means the LDA classifier is more suitable for test set.

2.5 Logistic Regression

2.5.1 Logistic Regression

We first define the sigmoid function needed to perform data transformation.

```
# Define sigmoid function
def sigmoid(x):
    return 1.0/(1+np.exp(-x))
```

Then we define the function for gradient descent, which is the most common algorithm for optimization of cross-entropy error function.

```
# Define Gradient Descent Method without regularization
```

```
def gradientDescent(X,Y):
    while(num_iter < max_iter):
        num_iter+=1
        weights2 = weights1
        y = sigmoid(dataMatrix*weights1)
        error = y - classMatrix
        weights1 = weights1 - (step_size)*dataMatrix.transpose()*error/len
                                                    (classMatrix)
        if all(abs(weights2-weights1) < episilon):
            break
    return weights1
```

Then, according to the parameter weight we get, we classify the training set and testing set into 0 or 1. Then we compute the corresponding precision, recall, and F1-score.

```
def logistic(train_X,train_Y,test_X,test_Y):
    temp_weight = gradientDescent(train_X,train_Y)
    pred_train_Y = pd.DataFrame(sigmoid(train_X.to_numpy()*temp_weight))
    #classify the prediction results for training set
    for i in range(len(pred_train_Y)):
        if pred_train_Y.iloc[i,0] <=0.5:
            pred_train_Y.iloc[i,0] = 0
        else:
            pred_train_Y.iloc[i,0] = 1

    train_precision = TP_train/(TP_train + FP_train)
    train_recall = TP_train/(TP_train + FN_train)
    train_F1_score = 2*train_precision*train_recall/(train_precision+
                                                train_recall)
```

Metrics of logistic regression for train data and test data are summarized in Table 7.

Data	**Accuracy**	**Precision**	**Recall**	**F1-score**
Train	0.7882	0.8276	0.1091	0.1928
Test	0.7152	0.5667	0.6667	0.6126

Table 7: Results for Logistic Regression

The ROC curve for logistic regression is shown in Figure 16.

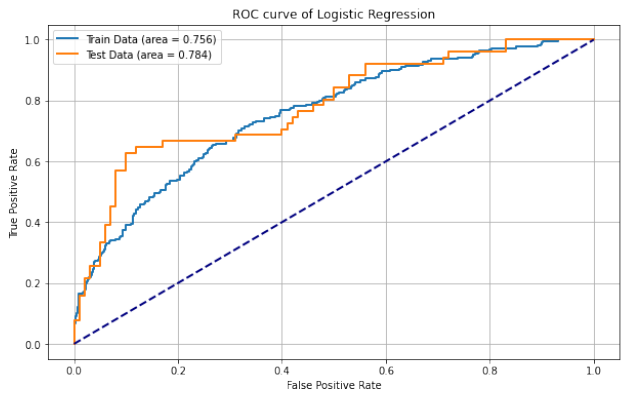

Figure 16: ROC for Logistic Regression

2.5.2 Logistic Regression with Regularization (L2)

Different from logistic regression without regularization, we re-define the function inside gradient descent to ensure the change brought by L2-norm regularization.

```
#define gradient descent with regularization
def gradientDescent_reg(X,Y,lam):
    while(num_iter < max_iter):
        num_iter+=1
        weights2 = weights1
        y = sigmoid(dataMatrix*weights1)
        error = y - classMatrix
        weights1 = weights1 - (step_size)*(dataMatrix.transpose()*error+
                                              lam*weights1)/len(classMatrix
                                              )
        if all(abs(weights2-weights1) < episilon):
            break
    return weights1
```

In the code below, we perform 10-fold cross validation for a specific λ. This function returns training set precision, training set recall, training set F1-score, validation set precision, validation set recall, validation set F1-score separately.

```
#Define cross-validation for logistic with regularization to test accuracy
def cross_validation_regularize(X,Y,lam):
    # Use 10-fold cross validation to estimate the error rate
    folds = 10
    for i in range(0, folds):
        # For each fold, calculate the accuracy
        vali_index = index[int(row/folds*i):int(row/folds*(i+1))]
        # Train set has 9 folds and validation set has 1 fold
        vali_x = X.iloc[vali_index]
        vali_y = Y.iloc[vali_index]
        train_x = X.iloc[list(set(index)-set(vali_index))]
        train_y = Y.iloc[list(set(index)-set(vali_index))]
        temp_weight = gradientDescent_reg(train_x,train_y,lam)
        #predict result for the train set
        pred_train_y = sigmoid(train_x.to_numpy()*temp_weight)
        pred_train_y = pd.DataFrame(pred_train_y)
        for j in range(len(pred_train_y)):
            if pred_train_y.iloc[j,0] <= 0.5:
                pred_train_y.iloc[j,0] = 0
            else:
                pred_train_y.iloc[j,0] = 1
```

The result is shown in the Figure 17.

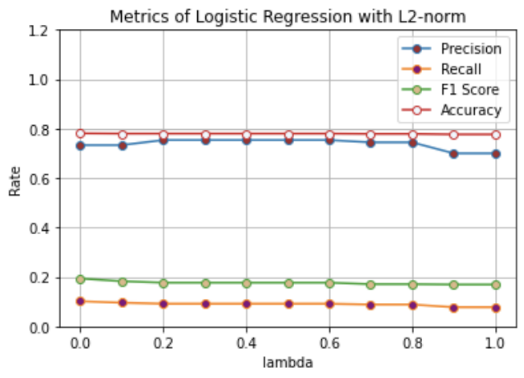

Figure 17: Metrics of Logistic Regression with L2-norm(range 0 to 1)

According to the graph, there seems to be little improvement of regularization to the original model.

As λ becomes larger between the range of 0 to 1, a trade-off between precision and recall occurs. The indicator for overall performance, F1-score, however, decreases. This phenomenon is possibly due to that the number of training data is large enough, the model does not have over-fitting issue. Adding λ, instead, would lower the performance of the model.

However, we still use $\lambda = 0.2$ to perform logistic regression with L2-norm to show the impact, as this choice of λ improves predicting precision.

Metrics of Logistic Regression with L2-norm are summarized in the following Table 8.

Data	Accuracy	Precision	Recall	F1-score
Train	0.7871	0.8214	0.1045	0.1855
Test	0.6911	0.5182	0.6434	0.5718

Table 8: Results for Logistic Regression with L2-norm Regularization

For the testing data, there seems to be no difference in accuracy, precision, recall and F1-score if we add the regularization term, which means that L2-norm regularization is not effective when it comes to improving logistic regression in this case.

The ROC curve for logistic regression with L2-norm Regularization is shown in Figure 18.

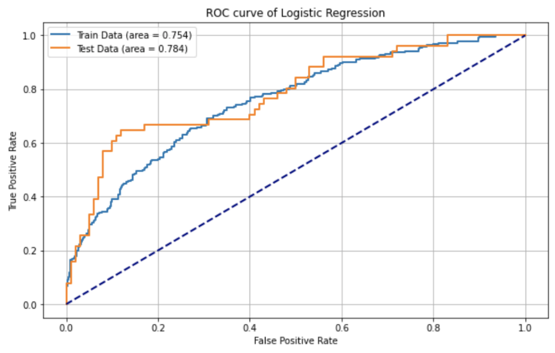

Figure 18: ROC for Logistic Regression with Regularization(L2)

2.6 Softmax Regression

2.6.1 Softmax Regression

We first transform the class from one-dimension to two-dimension.

```
# Transform the trainDY and testDY into two columns
trainDY["class2"] = 0
testDY["class2"] = 0

for i in range(949):
    if trainDY.iloc[i,0] == 0.0:
        trainDY.iloc[i,0] = 1
        trainDY.iloc[i,1] = 0
    else:
        trainDY.iloc[i,0] = 0
        trainDY.iloc[i,1] = 1
```

Then, we define the softmax function needed.

```
# Define the softmax function
def softmax(X):

    num_sample, num_class = X.shape
    for i in range(num_sample):
        denom = np.exp(X[i]).sum()
        X[i] = np.exp(X[i])/denom
    return(X)
```

Then we define the function for gradient descent, which is the most common algorithm for optimization of cross-entropy error function for softmax regression.

```
# Define gradient descent for softmax without regularization
# k implies the number of classes
def gradientDescent(X,Y,k):
    while(num_iter < max_iter):
        num_iter += 1
        weight2 = weight1
        y = softmax(dataMatrix*weight1)
        error = y - classMatrix
        weight1 = weight1 - (step_size)*dataMatrix.transpose()*error/
                                                     num_sample
        if (abs(weight2[:,0]-weight1[:,0]) < episilon).all():
            break
    return weight1
```

Then, we compute the training and testing accuracy for softmax regression, with maximum iteration equals to 2000. The result is shown in the following table 9.

Data	**Accuracy**	**Precision**	**Recall**	**F1-score**
Train	0.7819	0.7097	0.1000	0.1753
Test	0.6755	0.5128	0.7843	0.6202

Table 9: Results for Softmax Regression

The ROC curve for softmax regression is shown below in Figure 19.

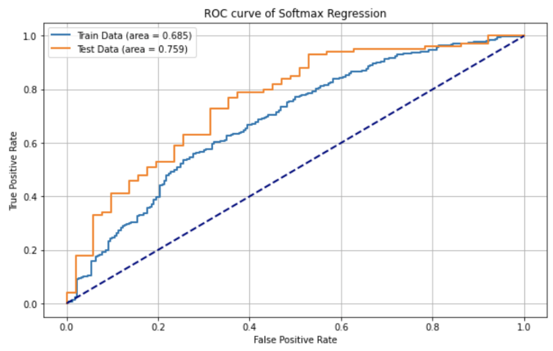

Figure 19: ROC for Softmax Regression

2.6.2 Softmax Regression with Regularization (L2)

Different from softmax regression without regularization, we re-define the function inside gradient descent to ensure the change brought by L2-norm regularization.

```
# Define gradient descent for softmax with regularization
# k implies the number of classes
def gradientDescent_reg(X,Y,k,lam):
    while(num_iter < max_iter):
        num_iter += 1
        weight2 = weight1
        y = softmax(dataMatrix*weight1)
        error = y - classMatrix
        weight1 = weight1 - (step_size)*(dataMatrix.transpose()*error+lam*
                                                      weight1)/num_sample
        if (abs(weight2[:,0]-weight1[:,0]) < episilon).all():
            break
    return weight1
```

Then, we perform 10-fold CV to choose the λ to optimize the predicting result.

```
#Define cross-validation for logistic with regularization to test accuracy
def cross_validation_regularize(X,Y,lam):
    # Use 10-fold cross validation to estimate the error rate
    folds = 10
    for i in range(0, folds):
        # For each fold, calculate the accuracy
        vali_index = index[int(row/folds*i):int(row/folds*(i+1))]
        # Train set has 9 folds and validation set has 1 fold
        vali_x = X.iloc[vali_index]
        vali_y = Y.iloc[vali_index]
        train_x = X.iloc[list(set(index)-set(vali_index))]
        train_y = Y.iloc[list(set(index)-set(vali_index))]
        temp_weight = gradientDescent_reg(train_x,train_y,2,lam)
        #predict result for the train set
        pred_train_y = softmax(train_x.to_numpy()*temp_weight)
        pred_train_y = pd.DataFrame(pred_train_y)
        temp_row, temp_col = pred_train_y.shape
        for j in range(temp_row):
            if pred_train_y.iloc[j,0] <= pred_train_y.iloc[j,1]:
                pred_train_y.iloc[j,0] = 0
                pred_train_y.iloc[j,1] = 1
            else:
                pred_train_y.iloc[j,0] = 1
                pred_train_y.iloc[j,1] = 0
```

Metrics of softmax regression with L2-norm regularization(10-fold CV) is shown in Figure 20.

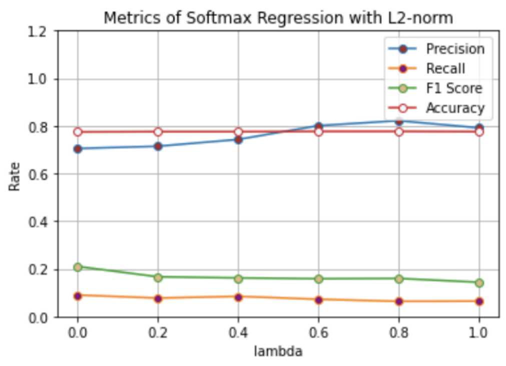

Figure 20: Metrics of Softmax Regression with L2-norm(10-fold CV)

According to the graph, as λ becomes larger between the range of 0 to 1, a trade-off between precision and recall occurs. The indicator for overall performance, F1-score, however, decreases. This phenomenon is possibly due to that the number of training data is large enough, the model does not have over-fitting issue. Adding *lambda*, instead, would lower the performance of the model.

However, we still choose $\lambda = 0.6$ to make precision better. The accuracy, precision, recall, F1-score of regularized Softmax Regression is shown in Table 10.

Data	Accuracy	Precision	Recall	F1-score
Train	0.7829	0.8182	0.0818	0.1488
Test	0.6490	0.4868	0.7255	0.5827

Table 10: Results for Regularized Softmax Regression

As can be concluded from the table, L2-norm regularization is not effective to improve softmax regression model's performance.

The ROC curve can be plotted as followed in Figure 21.

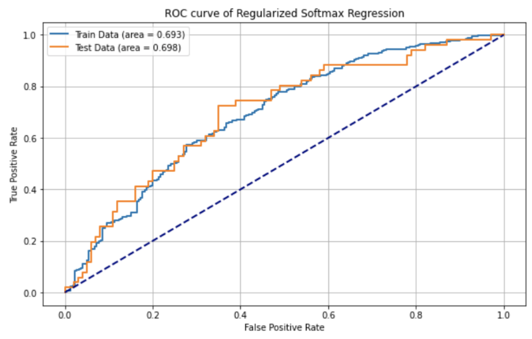

Figure 21: ROC for Regularized Softmax Regression

2.7 Decision Tree

Scikit-learn package is used for the realization of Decision Tree algorithm. In this classifier, there are some parameters to choose:

- max_features: the maximum features needed to split the tree;
- max_depth: the maximum depth of the decision tree;
- min_sample_split: minimum amount of samples needed to split inner nodes.

We perform the algorithm with different max_depth, then we calculate precision, recall ,F1-score, and accuracy.

```
def tree(para):
    for i in range(1,para):
        clf = DecisionTreeClassifier(max_depth=para)
        clf = clf.fit(trainDX,trainDY)
        train_predict = clf.predict(trainDX)
        test_predict = clf.predict(testDX)
        train_acc.append(Accuracy_train)
        train_pre.append(Precision_train)
        train_rec.append(Recall_train)
        train_f1.append(F1_train)
    return(train_acc,train_pre,train_rec,train_f1,test_acc,test_pre,
                                        test_rec,test_f1)
```

Metrics of the decision tree classifier are displayed in the following Figure 22.

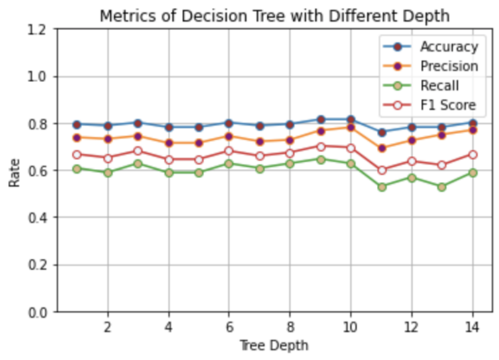

Figure 22: Metrics of Decision Tree

According to the graph, when the maximum tree depth is set to be ten, the tree has better performance in accuracy, precision, recall and F1 Score than the tree with maximum tree depth equal to fourteen. This phenomenon is possibly due to that when the tree depth gets larger, overfitting may occur, which influences the model's performance.

In order to get good testing performance as well as maintain easier interpretability, we choose maximum tree depth to be 10. We then train the model and get the accuracy, precision, recall and F1-score for training and testing data, as in Table 11.

Data	**Accuracy**	**Precision**	**Recall**	**F1-score**
Train	1.0	1.0	1.0	1.0
Test	0.8146	0.7805	0.6275	0.6957

Table 11: Results for Decision Tree Classifier

From the table, we can see that decision tree has an excellent performance on the training data, with relatively good testing data performance.

2.8 Random Forest

The package contains the random forest classifier. In this classifier, there are some parameters to choose:

- n_estimators: the number of trees;
- max_features: number of features which should be chosen, for classification problem, *sqrt* is the general option;
- oob_score: whether leaving some samples when building the model, for the generality of the model, we choose True.

```
# External package
from sklearn.ensemble import RandomForestClassifier
```

We perform 10-fold cross validation for a specific number of trees. For each number of trees, we calculate precision, recall, F1-score, and accuracy.

```
def rf_train(trainX, trainY, num_tree):
    # Use 10-fold cross validation
    folds = 10
    for i in range(0, folds):
        RandomForestClassifier(n_estimators=num_tree,max_features="sqrt",
                                                     oob_score=True,random_state=0
                                                     )
        model = forest.fit(train_x, train_y)
        pred = model.predict(vali_x)
        Pred += list(pred)
        Label += list(vali_y)
    return((P,R,F1,A))
```

Metrics of the random forest classifier are displayed in Figure 23.

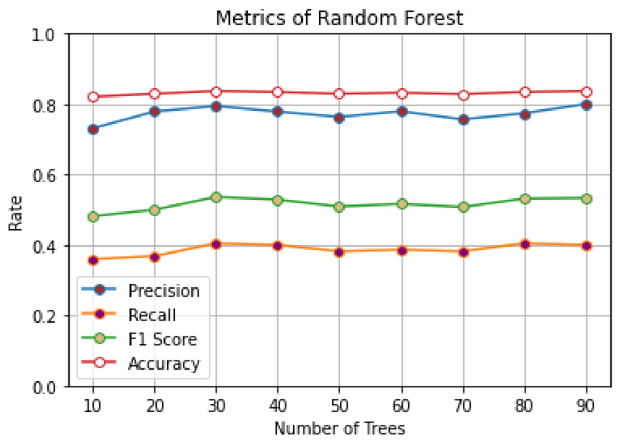

Figure 23: Metrics of Random Forest

From those metrics, We choose 30 as the optimal number of trees, which is used to train the model and make predictions on the test data.

Data	Accuracy	Precision	Recall	F1-score
Train	0.9989	1.0	0.9954	0.9977
Test	0.8211	0.8529	0.5686	0.6823

Table 12: Results for Random Forest Classifier

From Table 12, we conclude that random forest classifier has a good performance on both the train and test data.

2.9 Neural Network

The package contains fully-connected neural network classifier. The structure is introduced from tensorflow. In this classifier, there are some parameters to choose:

- batch_size: the number of samples selected for once training;
- learning_rate: the speed of model training;
- iteration_times: the total number of times that the model was trained.

First, we define the neural network structure. After performing 10-fold cross validation, we choose 2 as the optimal number of layers. We also find that the most significant activation function is sigmoid function. Therefore, we construct the neural network using sigmoid function in both hidden layers.

```
def Fully_neural_network(X):
    layer_1 = tf.nn.sigmoid(tf.add(tf.matmul(X, w1), b1))
    layer_2 = tf.nn.sigmoid(tf.add(tf.matmul(layer_1, w2), b2))
    layer_out = tf.matmul(layer_2, w_out) + b_out
    return layer_out
```

Then, we define the input structure, i.e. placeholder, which is helpful in training the model.

```
# features
xs=tf.placeholder(tf.float32,[None,64])
# labels
ys=tf.placeholder(tf.float32,[None,1])
# weight matrix
w1 = tf.Variable(tf.random_normal([len(trainDX[0]), 64], stddev=1, seed=1)
                                                    )
w2 = tf.Variable(tf.random_normal([64, 64], stddev=1, seed=1))
w_out = tf.Variable(tf.random_normal([64, 2], stddev=1, seed=1))
# bias matrix
b1 = tf.Variable(tf.random_normal([64]))
b2 = tf.Variable(tf.random_normal([64]))
b_out = tf.Variable(tf.random_normal([2]))
```

We use different parameters to train the neural network model. From training, we choose the optimal parameters as follows. The batch_size is 40, the learning_rate is 0.001, and the iteration_time is 1000.

ROC curves of Neural Network Classifier for train data and test data are drawn in Figure 24.

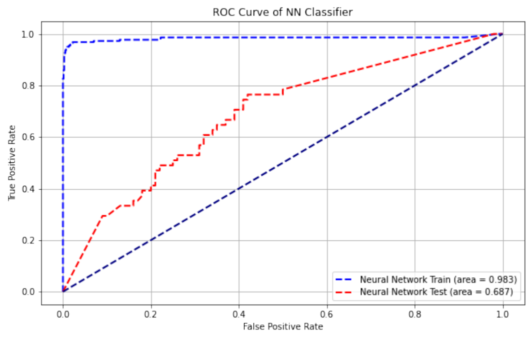

Figure 24: ROC for NN Classifier

Matrices of NN classifier for train data and test data are summarized in Table 13. We conclude that NN Classifier has an excellent performance on train data, but has a relatively poor performance

Data	**Accuracy**	**Precision**	**Recall**	**F1-score**
Train	0.9800	0.9631	0.9500	0.9565
Test	0.7102	0.6123	0.5798	0.5347

Table 13: Results for Neural Network Classifier

on test data. This may be resulted from overfitting. Or the sigmoid activation function is not suitable for test set.

3 Further Study

3.1 Synthetic Minority Oversampling Technique (SMOTE)

To improve the performance of models with imbalanced data set, we can oversample the minority class. Since oversampling randomly by duplicating samples may lead to the over-fitting problem, we try to synthesize new samples from existing ones, which is referred to as the SMOTE.

The procedure of SMOTE is displayed as below:

- For each sample x in the minority class, calculate the Euclidean distance between x and other samples in the minority class and get k nearest neighbours.
- Set an sampling ratio N, for each sample x, randomly choose $N-1$ samples from k nearest neighbours.
- For each neighbour $\tilde{x}$, construct a new sample $x_{new} = \tilde{x} + rand(0,1) \cdot (\tilde{x} - x)$

We use the existing package to generate SMOTE data set.

```
# Oversampling by existing packages
from imblearn.over_sampling import SMOTE
smo = SMOTE(random_state=1)
X_smo, y_smo = smo.fit_resample(trainX, trainY)
X_smo["class"] = y_smo["class"]
X_smo.to_csv("overSample.csv",index=False)
```

After splitting the data set into X and Y and passing them into the function, this package can automatically generate new samples and the ratio of two classes will be 1:1.

3.2 Model Performance on SMOTE Data Set

From relevant research, we find that oversampling method can sometimes improve the performance of models with regularization terms, random forest, and neural network, so we use the new data set to rebuild those models.

3.2.1 Least Squares Classification with Regularization (L2)

We use the SMOTE data set to perform 10-fold cross validation and choose the best λ, metrics are shown in Figure 25.

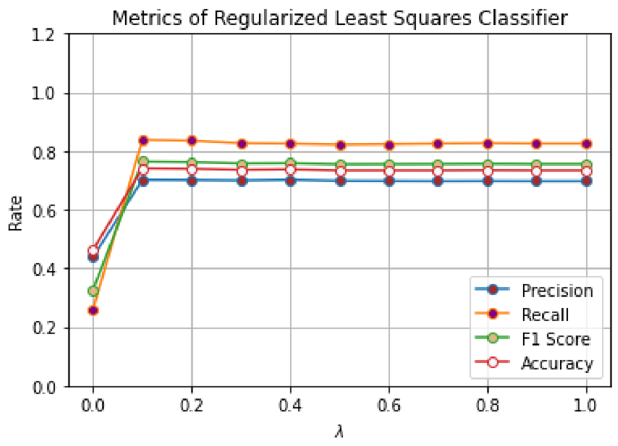

Figure 25: Metrics of Regularized Least Squares (SMOTE)

Train and test results on the optimal λ are displayed in Table 14. Although train metrics improve a lot, test metrics do not have much improvement compared with the original data set. It is judged that least squares classifier with regularization does not work well with this SMOTE data set.

Data	**Accuracy**	**Precision**	**Recall**	**F1-score**
Train	0.7215	0.6924	0.7969	0.7410
Test	0.5960	0.4431	0.7647	0.5611

Table 14: Results for Regularized Least Squares (SMOTE)

3.2.2 Logistic Regression with Regularization (L2)

We use the SMOTE data set to perform 10-fold cross validation and choose the best λ, metrics are shown in Figure 26.

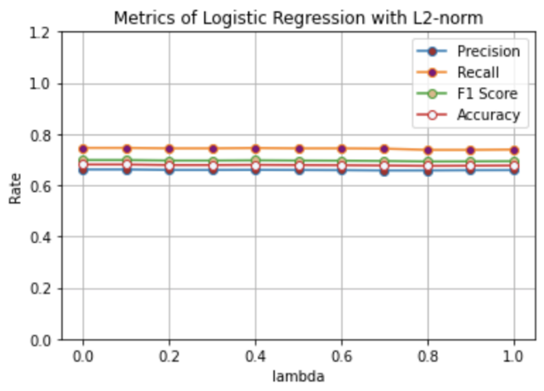

Figure 26: Metrics of Regularized Logistic Regression (SMOTE)

Train and test results on the optimal λ are displayed in Table 15. Regularized least squares classifier has some improvement on F1-score and recall when using SMOTE data set compared with the original data set.

Data	Accuracy	Precision	Recall	F1-score
Train	0.6996	0.6790	0.7572	0.7160
Test	0.6291	0.4684	0.7255	0.5692

Table 15: Results for Regularized Logistic Regression (SMOTE)

3.2.3 Random Forest

We perform the 10-fold cross validation to choose the best number of trees.

Compared with the original data set, the SMOTE data declares more variation with different number of trees. From Figure 27, we choose 40 as the optimal number of trees.

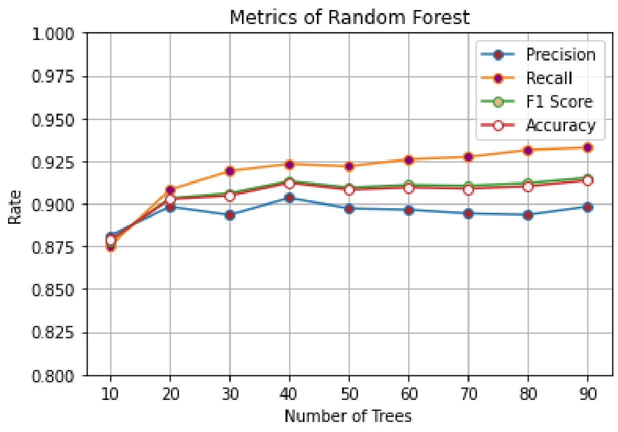

Figure 27: Metrics of Random Forest (SMOTE)

Data	Accuracy	Precision	Recall	F1-score
Train	1.0	1.0	1.0	1.0
Test	0.8412	0.8796	0.6102	0.7205

Table 16: Results for Random Forest Classifier (SMOTE)

Train and test results of random forest using SOMTE data are displayed in Table 16. It can be seen that random forest with SMOTE data set has some improvement on F1-score, the increase is 0.0382, which is impressive. As a result, we think that oversampling can work well with random forest under this specific data set.

3.2.4 Neural Network

We use the SMOTE data set to train the neural network and adjust the parameters to optimize the result. ROC curves of Neural Network Classifier for train data and test data are drawn in Figure 28. The final model metrics are shown in table 17.

Data	Accuracy	Precision	Recall	F1-score
Train	0.9658	0.9821	0.9575	0.9748
Test	0.7289	0.6381	0.6792	0.6003

Table 17: Results for Neural Network Classifier (SMOTE)

Train and test results of neural network using SMOTE data are displayed in Table 17. The

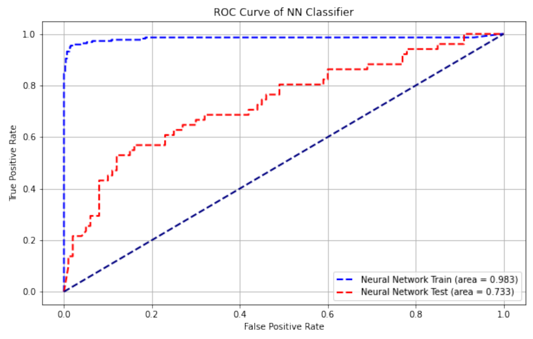

Figure 28: Metrics of Neural Network (SMOTE)

test metrics have some improvement on Recall and F1-score. It seems like that both train and test metrics do not make much improvement on prediction accuracy and precision. To some extent, we can conclude that the neural network does not fit this data set.

3.3 Possible Explanation for Better Performance of Certain Models on Test Data Compared with Train Data

Surprising as it might seem, in the models we discussed above, we can find that for least squares classification, regularized least squares classification, LDA, logistic regression, regularized logistic regression, softmax regression, regularized softmax regression, their testing recalls and F1-scores are higher than their training recalls and F1-scores.

Obviously and reasonably, the train data we use must be far away from the decision boundary, which may be caused by that the difference of distributions between train data and test data. To verify this point of view, we plot the distribution of train data and test data in Figure 29.

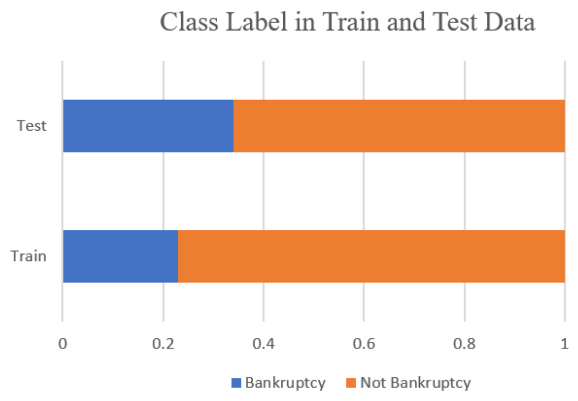

Figure 29: Class Label in Train and Test Data

3.4 Capture of Most Contributory Features

Coming back to reality, it is essential for us to understand which of the features contribute the most to the prediction of models. We conduct related research on the features, to identify which of the features is the most important one, and give related advice to certain companies.

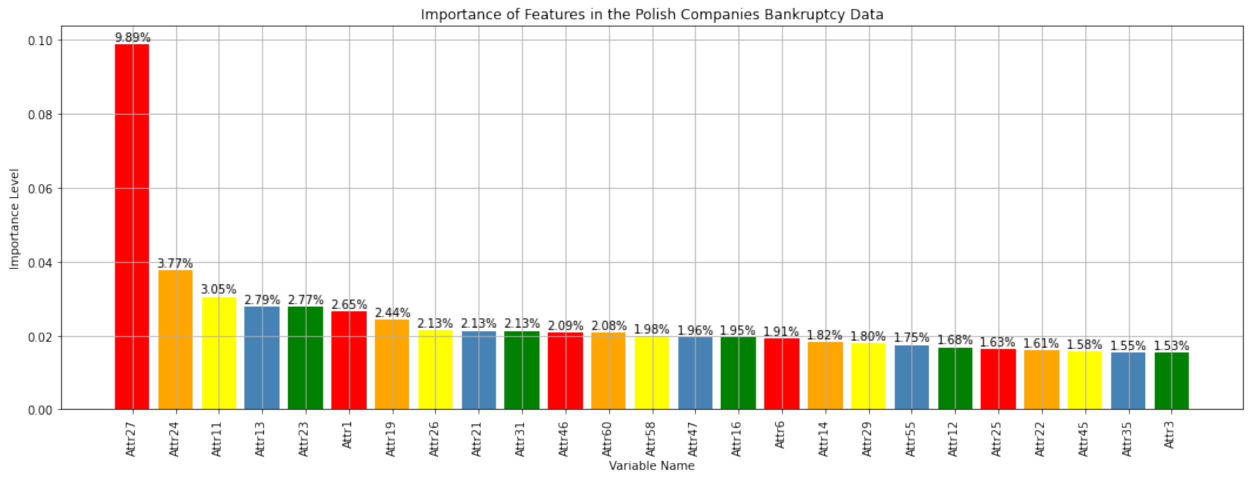

Figure 30: Importance of Different Features

Most contributory six features:

- Attr27: profit on operating activities / financial expenses
- Attr24: gross profit (in 3 years) / total assets
- Attr11: (gross profit + extraordinary items + financial expenses) / total assets
- Attr13: (gross profit + depreciation) / sales
- Attr23: net profit / sales
- Attr1: net profit / total assets

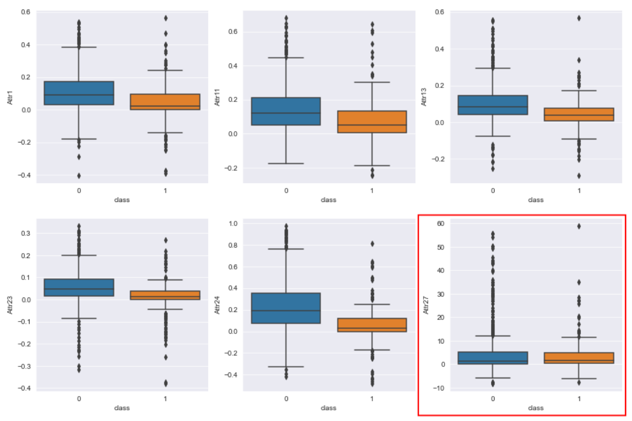

Figure 31: Box Plot of Contributory Plot

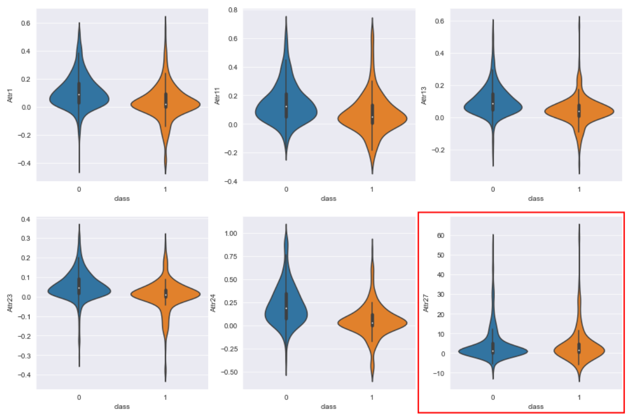

Figure 32: Violin Plot of Contributory Features

Distribution of the six features can be seen from Figure 31 and 32 (some outliers are eliminated to plot the figure). Feature with the red circle is the most contributory one.

Least contributory six features:

- Attr51: short-term liabilities / total assets
- Attr16: (gross profit + depreciation) / total liabilities
- Attr7: EBIT / total assets
- Attr32: (current liabilities * 365) / cost of products sold
- Attr62: (short-term liabilities *365) / sales
- Attr31: (gross profit + interest) / sales

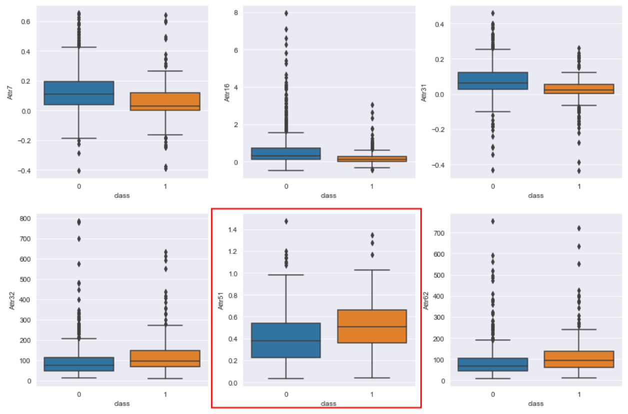

Figure 33: Box Plot of Less Contributory Features

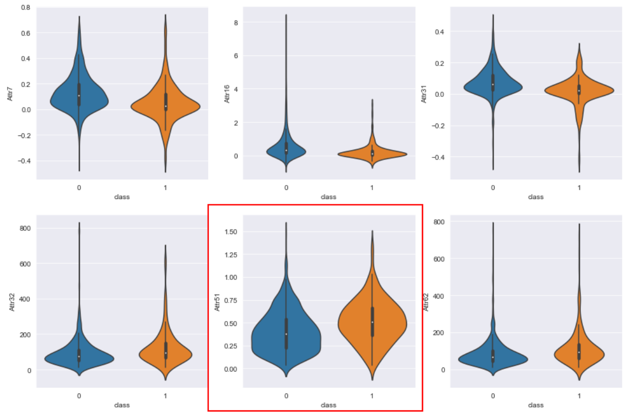

Figure 34: Violin Plot of Less Contributory Features

Distribution of the six features can be seen from Figure 33 and 34 (some outliers are eliminated to plot the figure). Feature with the red circle is the least contributory one.

4 Algorithms Comparison

In order to choose the most suited model, we make a table to demonstrate the models' performance.

Model	**Accuracy**	**Precision**	**Recall**	**F1-score**
KNN	0.7086	0.5813	0.4901	0.5319
LS	0.5364	0.3733	0.5490	0.4444
LS with L2	0.5894	0.4367	0.7450	0.5507
LS Basis Function	0.6754	0.5357	0.2941	0.3797
LDA	0.6689	0.5386	0.7255	0.5968
Logistic	0.7152	0.5667	0.6667	0.6126
Logistic with L2	0.6911	0.5182	0.6434	0.5718
Softmax	0.6755	0.5128	0.7843	0.6202
Softmax with L2	0.6490	0.4868	0.7255	0.5827
Decision Tree	0.8146	0.7805	0.6275	0.6957
Random Forest	0.8211	0.8529	0.5686	0.6823
Neural Network	0.7102	0.6123	0.5798	0.5347

Table 18: Test Results Comparison on Model

As is shown in Table 18, with its high F1-score, recall and easy interpretability, we select decision tree as the most suited model. Here, we visualize the decision tree to show its simplicity and accuracy in Figure 35.

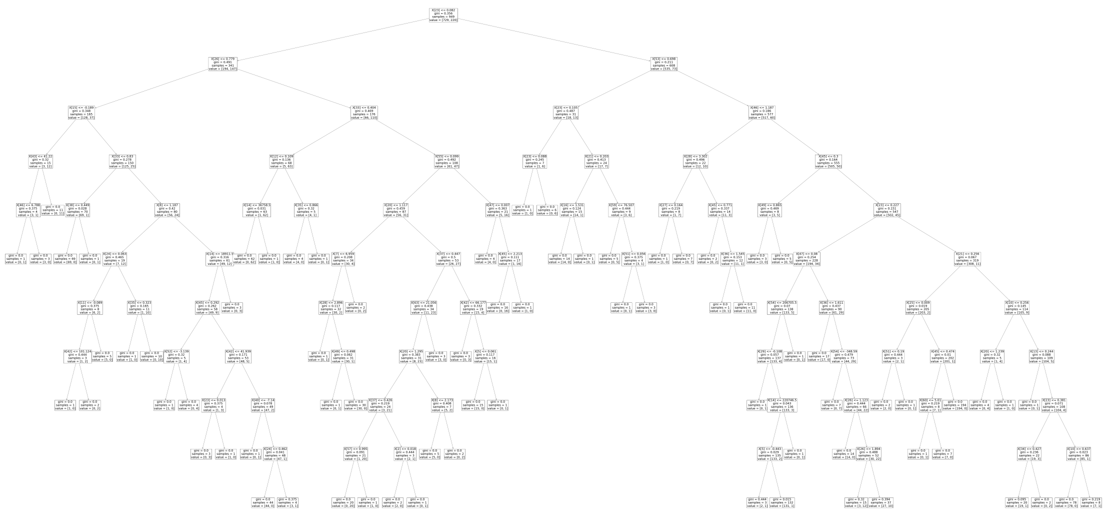

Figure 35: Decision Tree

The SMOTE data set can also be used to improve the model. Compared with models trained with original data, we can find that random forest has a large improvement with its recall and F1-score. And the result is satisfied.

Model	Data	Accuracy	Precision	Recall	F1-score
LS with L2	Original	0.5894	0.4367	0.7450	0.5507
LS with L2	SMOTE	0.5960	0.4431	0.7647	0.5611
Logistic with L2	Original	0.7152	0.5667	0.6667	0.6126
Logistic with L2	SMOTE	0.6291	0.4684	0.7255	0.5692
Random Forest	Original	0.8211	0.8529	0.5686	0.6823
Random Forest	SMOTE	0.8412	0.8796	0.6102	0.7205
Neural Network	Original	0.7102	0.6123	0.5798	0.5347
Neural Network	SMOTE	0.7289	0.6381	0.6792	0.6003

Table 19: Test Results Comparison

To summarize, considering the models' accuracy, precision, recall, F1-score and interpretability, we choose decision tree and random forest(SMOTE) as our final model.

5 Conclusion

We have tried different feature engineering techniques to process the data. Then, we apply the data to all those models and evaluate their performance. We also make some further study to make up for the bias resulted from data imbalance by SMOTE and improve models to some extent.

After comprehensive comparison between different models' performance, we choose best models for this data set. Advice has been given to help certain companies to avoid bankruptcy and help investors to distinguish between reliable and unreliable companies.

6 Appendix

- KNN

```
thresholds = np.linspace(0,1,100)
k_choices = np.linspace(1,11,6)
# False positive rate
FPR_df = []
# True positive rate
TPR_df = []
precision = []
recall = []
F1 = []
accuracy = []
# Evaluate the model for each k
for k in k_choices:
    result = knn_train(trainDX, trainDY, int(k))
    prob = [i[0] for i in result]
    label = [i[1] for i in result]
    FPR_list = []
    TPR_list = []
    pred = [int(i>=0.5) for i in prob]
    TP=len([i for i in range(len(pred)) if pred[i]==1 and label[i]==1
                                          ])
    FP=len([i for i in range(len(pred)) if pred[i]==1 and label[i]==0
                                          ])
    TN=len([i for i in range(len(pred)) if pred[i]==0 and label[i]==0
                                          ])
    FN=len([i for i in range(len(pred)) if pred[i]==0 and label[i]==1
                                          ])
    P = TP/(TP+FP)
    precision.append(P)
    R = TP/(TP+FN)
    recall.append(R)
    F1.append(2*P*R/(P+R))
    accuracy.append((TP+TN)/(TP+TN+FP+FN))
    # Calculate FPR and TPR for each threshold
    for t in thresholds:
        pred = [int(i>=t) for i in prob]
        TP = len([i for i in range(len(pred)) if pred[i]==1 and label
                                              [i]==1])
        FP = len([i for i in range(len(pred)) if pred[i]==1 and label
                                              [i]==0])
        TN = len([i for i in range(len(pred)) if pred[i]==0 and label
                                                    [i]==0])
        FN = len([i for i in range(len(pred)) if pred[i]==0 and label
                                                    [i]==1])
        FPR = FP/(FP+TN)
        TPR = TP/(TP+FN)
        FPR_list.append(FPR)
        TPR_list.append(TPR)
    FPR_df.append(FPR_list)
    TPR_df.append(TPR_list)
```

```
thresholds = np.linspace(0,1,100)
k_choices = np.linspace(1,11,6)
# False positive rate
FPR_df = []
# True positive rate
TPR_df = []
precision = []
recall = []
F1 = []
accuracy = []
# Evaluate the model for each k
for k in k_choices:
    result = knn_train(trainDX, trainDY, int(k))
    prob = [i[0] for i in result]
    label = [i[1] for i in result]
    FPR_list = []
    TPR_list = []
    pred = [int(i>=0.5) for i in prob]
    TP=len([i for i in range(len(pred)) if pred[i]==1 and label[i]==1
                                         ])
    FP=len([i for i in range(len(pred)) if pred[i]==1 and label[i]==0
                                         ])
    TN=len([i for i in range(len(pred)) if pred[i]==0 and label[i]==0
                                         ])
    FN=len([i for i in range(len(pred)) if pred[i]==0 and label[i]==1
                                         ])
    P = TP/(TP+FP)
    precision.append(P)
    R = TP/(TP+FN)
    recall.append(R)
    F1.append(2*P*R/(P+R))
    accuracy.append((TP+TN)/(TP+TN+FP+FN))
    # Calculate FPR and TPR for each threshold
    for t in thresholds:
        pred = [int(i>=t) for i in prob]
        TP = len([i for i in range(len(pred)) if pred[i]==1 and label
                                                [i]==1])
        FP = len([i for i in range(len(pred)) if pred[i]==1 and label
                                                [i]==0])
        TN = len([i for i in range(len(pred)) if pred[i]==0 and label
                                                [i]==0])
        FN = len([i for i in range(len(pred)) if pred[i]==0 and label
                                                [i]==1])
        FPR = FP/(FP+TN)
        TPR = TP/(TP+FN)
        FPR_list.append(FPR)
        TPR_list.append(TPR)
    FPR_df.append(FPR_list)
    TPR_df.append(TPR_list)
```

```
def knn_test(trainX, trainY, testX, testY, k=3):
    row = testX.shape[0]
    Prob = []
    for i in range(row):
        sample = testX.iloc[[i]]
        diff = np.tile(sample, (trainX.shape[0], 1)) - trainX
        dis = np.sqrt((diff**2).sum(axis=1))
        dis_dict = pd.DataFrame(dis).to_dict()[0]
        sort_dis = sorted(dis_dict.items(), key=lambda item:item[1])
        neigh = trainY["class"][[a[0] for a in sort_dis[:k]]]
        neigh_l = list(neigh)
        true = testY["class"][i]
        num = neigh_l.count(1)
        ratio = num/k
        Prob.append([ratio,true])
    return(Prob)
```

- Least Squares Classifier

```
# Least squares without regularization
def LS(trainX, trainY):
    W = np.dot(np.dot(trainY, trainX.T), np.linalg.inv(np.dot(trainX,
                                            trainX.T)))
    prob = np.dot(W, trainX).T
    pred = [int(i[1]>i[0]) for i in prob]
    label = [list(i).index(1) for i in trainY.T]
    TP = len([i for i in range(len(pred)) if pred[i]==1 and label[i]=
                                                   =1])
    TN = len([i for i in range(len(pred)) if pred[i]==0 and label[i]=
                                                   =0])
    FP = len([i for i in range(len(pred)) if pred[i]==1 and label[i]=
                                                   =0])
    FN = len([i for i in range(len(pred)) if pred[i]==0 and label[i]=
                                                   =1])
    precision = TP/(TP+FP)
    recall = TP/(TP+FN)
    F1 = 2*precision*recall/(precision+recall)
    accuracy = (TP+TN)/(TP+TN+FP+FN)
    return((precision,recall,F1,accuracy))
```

- Least Squares Classifier with Regularization

```
# Least squares with regularization
def LS_regu(trainX, trainY, para):
    # Set the seed
    random.seed(1)
    # Divide indices of data into 10 folds randomly
    index0 = [i for i in range(len(trainY[0])) if trainY[0][i]==1]
    index1 = [i for i in range(len(trainY[1])) if trainY[1][i]==1]
    index = index0+index1
    random.shuffle(index0)
    random.shuffle(index1)
    diagonal = np.diag([para]*trainX.shape[0])
    # Use 10-fold cross validation to estimate the error rate
    folds = 10
    Prob = []
    Pred = []
    L = []
    for i in range(0, folds):
        # For each fold, calculate the accuracy
        vali_index = index0[int(len(index0)/folds*i):int(len(index0)/
                                        folds*(i+1))] + index1[
                                        int(len(index1)/folds*i)
                                        :int(len(index1)/folds*(
                                        i+1))]
        # Train set has 9 folds and validation set has 1 fold
        vali_x = trainX[:,vali_index]
        train_x = trainX[:,list(set(index)-set(vali_index))]
        vali_y = trainY[:,vali_index]
        train_y = trainY[:,list(set(index)-set(vali_index))]
        if np.linalg.det(np.dot(train_x, train_x.T)) != 0:
            left = np.dot(train_y, train_x.T)
            right = np.dot(train_x, train_x.T)+diagonal
            W = np.dot(left, np.linalg.inv(right))
            prob = np.dot(W, vali_x).T
            label = [list(i).index(1) for i in vali_y.T]
            Prob += [i[1] for i in prob]
            Pred += [int(i[1]>i[0]) for i in prob]
            L += label
    return(np.array(Prob),np.array(Pred),np.array(L))
```

```
# Cross validation for different lambda
lam_choices = np.linspace(0,0.01,11)
for lam in lam_choices:
    result = LS_regu(trainDX, trainDY, lam)
    Prob = result[0]
    Pred = result[1]
    Label = result[2]
```

```
def LS_test(trainX, trainY, testX, testY, para=0.02):
    diagonal = np.diag([para]*trainX.shape[0])
    L = []
    left = np.dot(trainY, trainX.T)
    right = np.dot(trainX, trainX.T)+diagonal
    W = np.dot(left, np.linalg.inv(right))
    prob = np.dot(W, testX).T
    list = [list(i).index(1) for i in testY.T]
    Prob = [i[1] for i in prob]
    Pred = [int(i[1]>i[0]) for i in prob]
    return(np.array(Prob),np.array(Pred),np.array(L))
```

- Least Squares Classifier with Gaussian Basis Function

```
# Data transformation
np.random.seed(1)
row = trainDX.shape[0]
col = trainDX.shape[1]
meanMatrix = np.random.normal(0,1,size=(100,col))
kernelX = np.zeros((row, 100))
sigma = np.diag([np.sqrt((trainDX**2).sum(axis=1)).mean()]*col)
a = 1/(2*math.pi)**(col/2)
b = 1/(np.linalg.det(sigma))**0.5
```

```
for i in range(kernelX.shape[0]):
    sample = np.array(trainDX.iloc[i])
    for j in range(kernelX.shape[1]):
        mean = meanMatrix[j]
        c = np.dot((sample-mean).T,np.linalg.inv(sigma))
        d = sample-mean
        kernelX[i][j] = a*b*np.exp(-0.5*np.dot(c,d))
```

```
# Train Model
def LSG(X, trainY):
    DX = X.T
    DY = np.zeros((trainY.shape[0],2))
    for i in range(trainY.shape[0]):
        DY[i][trainY[i]] = 1
    DY = DY.T
    left = np.dot(DY,DX.T)
    right = np.dot(DX,DX.T)
    W = np.dot(left,np.linalg.inv(right))
    prob = np.dot(W, DX).T
    P = [i[1] for i in prob]
    pred = [int(i[1]>i[0]) for i in prob]
    label = [list(i).index(1) for i in DY.T]
    TP = len([i for i in range(len(pred)) if pred[i]==1 and label[i]=
                                            =1])
    TN = len([i for i in range(len(pred)) if pred[i]==0 and label[i]=
                                            =0])
    FP = len([i for i in range(len(pred)) if pred[i]==1 and label[i]=
                                            =0])
    FN = len([i for i in range(len(pred)) if pred[i]==0 and label[i]=
                                            =1])
    precision = TP/(TP+FP)
    recall = TP/(TP+FN)
    F1 = 2*precision*recall/(precision+recall)
    accuracy = (TP+TN)/(TP+TN+FP+FN)
    return((accuracy,precision,recall,F1,P,label))
```

```
# Test Model
def LSG_test(X, trainY, testX, testY):
    DX = X.T
    DY = np.zeros((trainY.shape[0],2))
    for i in range(trainY.shape[0]):
        DY[i][trainY[i]] = 1
    DY = DY.T
    left = np.dot(DY,DX.T)
    right = np.dot(DX,DX.T)
    W = np.dot(left,np.linalg.inv(right))
    DXT = testX.T
    prob = np.dot(W, DXT).T
    P = [i[1] for i in prob]
    pred = [int(i[1]>i[0]) for i in prob]
    label = [i for i in testY]
    TP = len([i for i in range(len(pred)) if pred[i]==1 and label[i]=
                                                =1])
    TN = len([i for i in range(len(pred)) if pred[i]==0 and label[i]=
                                                =0])
    FP = len([i for i in range(len(pred)) if pred[i]==1 and label[i]=
                                                =0])
    FN = len([i for i in range(len(pred)) if pred[i]==0 and label[i]=
                                                =1])
    precision = TP/(TP+FP)
    recall = TP/(TP+FN)
    F1 = 2*precision*recall/(precision+recall)
    accuracy = (TP+TN)/(TP+TN+FP+FN)
    return((accuracy,precision,recall,F1,P,label))
```

- Linear Discriminant Analysis

```
def projection(x,x0,x1,y):
    # mean of each class
    np.set_printoptions(precision=6)
    u = np.array(np.mean(x,axis=0))
    u_0 = np.mean(x0,axis=0)
    u_1 = np.mean(x1,axis=0)

    # within-class scatter matrix
    S_W = np.zeros((np.array(x).shape[1], np.array(x).shape[1]))
    X0 = np.array(x)[y == 0] - u_0
    S_W += np.mat(X0).T * np.mat(X0)
    X1 = np.array(x)[y == 1] - u_1
    S_W += np.mat(X1).T * np.mat(X1)

    # between-class scatter matrix
    S_B = np.zeros((np.array(x).shape[1], np.array(x).shape[1]))
    mu = np.mean(np.array(x), axis=0)
    N0 = len(np.array(x)[y == 0])
    S_B += N0 * np.mat(u_0 - mu).T * np.mat(u_0 - mu)
    N1 = len(np.array(x)[y == 1])
    S_B += N1 * np.mat(u_1 - mu).T * np.mat(u_1 - mu)

    # eigenvals, eigenvectors
    eigvals, eigvecs = np.linalg.eig(np.linalg.inv(S_W) * S_B)
    # sorting the eigenvectors by decreasing eigenvalues
    eig_vecs = sorted(eigvecs, key=lambda k: k[0], reverse=True)
    # eigenvector corresponding to the largest eigenvalue
    w = eig_vecs[0]

    # mean after projection
    m0 = np.dot(np.transpose(w),u_0)
    m1 = np.dot(np.transpose(w),u_1)
    w0 = -(m0+m1)/2
    return w,w0
```

```
# Use 10-fold cross validation to estimate the error rate
folds = 10
kf = KFold(n_splits = folds)
TX, TY, VX, VY, Train, Vali = [], [], [], [], [], []
for train, vali in kf.split(trainData):
    # Train set has 9 folds and validation set has 1 fold
    train = trainData.loc[list(train),]
    vali = trainData.loc[list(vali),]
    train_x = train[train.columns.values[:-1]]
    train_y = np.concatenate(train[["class"]].values)
    vali_x = vali[vali.columns.values[:-1]]
    vali_y = np.concatenate(vali[["class"]].values)
    TX.append(train_x)
    TY.append(train_y)
    VX.append(vali_x)
    VY.append(vali_y)

def LDA_train(TX, VX, TY, VY):
    Precision, Recall, F1score, Accuracy = [], [], [], []
    W0lst, Wlst = [], []
    for i in range(len(TX)):
        training_data, vali_data = TX[i], VX[i]
        training_data_0 = np.array(TX[i])[TY[i] == 0]
        training_data_1 = np.array(TX[i])[TY[i] == 1]
        w,w0 = projection(training_data,training_data_0,
                                              training_data_1,TY[i])
        W0lst.append(w0)
        Wlst.append(w)
        pred = []
        gl = np.dot(w.T,np.array(VX[i]).T)
        Prob = [-i for i in gl]
        pred = [abs(0-int(i>=w0)) for i in Prob]
        TP = len([j for j in range(len(pred)) if np.array(pred)[j]==1
                                               and VY[i][j]==1])
        FP = len([j for j in range(len(pred)) if np.array(pred)[j]==1
                                               and VY[i][j]==0])
        TN = len([j for j in range(len(pred)) if np.array(pred)[j]==0
                                               and VY[i][j]==0])
        FN = len([j for j in range(len(pred)) if np.array(pred)[j]==0
                                               and VY[i][j]==1])
        P = TP / (TP+FP)
        Precision.append(P)
        R = TP / (TP+FN)
        Recall.append(R)
        F = 2*P*R / (P+R)
        F1score.append(F)
        A = (TP+TN) / (TP+TN+FP+FN)
        Accuracy.append(A)
        return Precision, Recall, F1score, Accuracy, W0lst, Wlst
```

```
def LDA_test(W0lst, Wlst, test_x, test_y):
    w0test = np.mean(W0lst)
    Wlst = pd.DataFrame(Wlst)
    wtest = np.array(np.mean(Wlst[list(range(trainData.shape[1]-1))])
                                     )
    pred = []
    gl = np.dot(wtest.T,np.array(test_x).T)
    Prob = [-i for i in gl]
    pred = [abs(0-int(i>=w0test)) for i in Prob]
    TP = len([j for j in range(len(pred)) if np.array(pred)[j]==1 and
                                          test_y[j]==1])
    FP = len([j for j in range(len(pred)) if np.array(pred)[j]==1 and
                                          test_y[j]==0])
    TN = len([j for j in range(len(pred)) if np.array(pred)[j]==0 and
                                          test_y[j]==0])
    FN = len([j for j in range(len(pred)) if np.array(pred)[j]==0 and
                                          test_y[j]==1])
    Precision = TP / (TP+FP)
    Recall = TP / (TP+FN)
    F1score = 2*Precision*Recall / (Precision+Recall)
    Accuracy = (TP+TN) / (TP+TN+FP+FN)
    return Precision, Recall, F1score, Accuracy
```

- Logistic Regression

```
# Define sigmoid function
def sigmoid(x):
    return 1.0/(1+np.exp(-x))
```

```
# Define Gradient Descent Method without regularization
def gradientDescent(X,Y):
    dataMatrix = np.mat(X)
    classMatrix = np.mat(Y)
    num_sample, num_parameter = np.shape(dataMatrix)
    #initialize the weights
    weights1 = np.zeros((num_parameter,1))
    weights2 = np.zeros((num_parameter,1))
    step_size = 0.01
    max_iter = 50000
    episilon = 0.0000001
    num_iter = 0
    while(num_iter < max_iter):
        num_iter+=1
        weights2 = weights1
        y = sigmoid(dataMatrix*weights1)
        error = y - classMatrix
        weights1 = weights1 - (step_size)*dataMatrix.transpose()*
                                        error/len(classMatrix)
        if all(abs(weights2-weights1) < episilon):
            break
    return weights1
```

```
def logistic(train_X,train_Y,test_X,test_Y):
    temp_weight = gradientDescent(train_X,train_Y)
    pred_train_Y = pd.DataFrame(sigmoid(train_X.to_numpy()*
                                    temp_weight))
    #classify the prediction results for training set
    for i in range(len(pred_train_Y)):
        if pred_train_Y.iloc[i,0] <=0.5:
            pred_train_Y.iloc[i,0] = 0
        else:
            pred_train_Y.iloc[i,0] = 1
    TP_train = 0
```

```
FP_train = 0
TN_train = 0
FN_train = 0
#calculate TP,FP,TN,FN
for j in range(len(pred_train_Y)):
    if (pred_train_Y.iloc[j,0] == 1 and train_Y.iloc[j,0] == 1):
        TP_train += 1
    elif (pred_train_Y.iloc[j,0] == 1 and train_Y.iloc[j,0] == 0)
                                                           :
        FP_train += 1
    elif (pred_train_Y.iloc[j,0] == 0 and train_Y.iloc[j,0] == 0)
                                                           :
        TN_train += 1
    else:
        FN_train += 1

train_precision = TP_train/(TP_train + FP_train)
train_recall = TP_train/(TP_train + FN_train)
train_F1_score = 2*train_precision*train_recall/(train_precision+
                                      train_recall)

#classify the prediction results for the test set
pred_test_Y = pd.DataFrame(sigmoid(test_X.to_numpy()*temp_weight)
                                      )
for m in range(len(pred_test_Y)):
    if pred_test_Y.iloc[m,0] <=0.5:
        pred_test_Y.iloc[m,0] = 0
    else:
        pred_test_Y.iloc[m,0] = 1
TP_test = 0
FP_test = 0
TN_test = 0
FN_test = 0
#calculate TP,FP,TN,FN
for n in range(len(pred_test_Y)):
    if (pred_test_Y.iloc[n,0] == 1 and test_Y.iloc[n,0] == 1):
        TP_test += 1
    elif (pred_test_Y.iloc[n,0] == 1 and test_Y.iloc[n,0] == 0):
        FP_test += 1
    elif (pred_test_Y.iloc[n,0] == 0 and test_Y.iloc[n,0] == 0):
        TN_test += 1
    else:
        FN_test += 1
test_precision = TP_test/(TP_test + FP_test)
test_recall = TP_test/(TP_test + FN_test)
test_F1_score = 2*test_precision*test_recall/(test_precision+
                                      test_recall)

return(train_precision,train_recall,train_F1_score,test_precision
                                      ,test_recall,test_F1_score)
```

- Logistic Regression with Regularization

```
#define gradient descent with regularization
def gradientDescent_reg(X,Y,lam):
    dataMatrix = np.mat(X)
    classMatrix = np.mat(Y)
    num_sample, num_parameter = np.shape(dataMatrix)
    #initialize the weights
    weights1 = np.zeros((num_parameter,1))
    weights2 = np.zeros((num_parameter,1))
    step_size = 0.01
    max_iter = 50000
    episilon = 0.0000001
    num_iter = 0
    while(num_iter < max_iter):
        num_iter+=1
        weights2 = weights1
        y = sigmoid(dataMatrix*weights1)
        error = y - classMatrix
        weights1 = weights1 - (step_size)*(dataMatrix.transpose()*
                                            error+lam*weights1)/len(
                                            classMatrix)
        if all(abs(weights2-weights1) < episilon):
            break
    return weights1
```

```
#Define cross-validation for logistic with regularization to test
                                     accuracy
def cross_validation_regularize(X,Y,lam):
    #Create the fold to save precision
    precision_train = []
    precision_vali = []
    recall_train = []
    recall_vali = []
```

```
# Set the seed
random.seed(1)
# Divide indices of data into 10 folds randomly
row = X.shape[0]
index = [int(i) for i in np.linspace(0, row-1, row)]
random.shuffle(index)
# Use 10-fold cross validation to estimate the error rate
folds = 10
for i in range(0, folds):
    # For each fold, calculate the accuracy
    vali_index = index[int(row/folds*i):int(row/folds*(i+1))]
    # Train set has 9 folds and validation set has 1 fold
    vali_x = X.iloc[vali_index]
    vali_y = Y.iloc[vali_index]
    train_x = X.iloc[list(set(index)-set(vali_index))]
    train_y = Y.iloc[list(set(index)-set(vali_index))]
    temp_weight = gradientDescent_reg(train_x,train_y,lam)
    #predict result for the train set
    pred_train_y = sigmoid(train_x.to_numpy()*temp_weight)
    pred_train_y = pd.DataFrame(pred_train_y)
    for j in range(len(pred_train_y)):
        if pred_train_y.iloc[j,0] <= 0.5:
            pred_train_y.iloc[j,0] = 0
        else:
            pred_train_y.iloc[j,0] = 1
    TP_train = 0
    FP_train = 0
    FN_train = 0
    TN_train = 0
    for k in range(len(pred_train_y)):
        if pred_train_y.iloc[k,0] == 1 and train_y.iloc[k,0] == 1
                                                                :
            TP_train+=1
        elif pred_train_y.iloc[k,0] == 1 and train_y.iloc[k,0] ==
                                                                0:
            FP_train+=1
        elif pred_train_y.iloc[k,0] == 0 and train_y.iloc[k,0] ==
                                                                1:
            FN_train+=1
        else:
            TN_train+=1
    precision_train.append(TP_train/(TP_train+FP_train))
    recall_train.append(TP_train/(TP_train + FN_train))
```

```
        #predict result for the test set
        pred_vali_y = sigmoid(vali_x.to_numpy()*temp_weight)
        pred_vali_y = pd.DataFrame(pred_vali_y)
        for j in range(len(pred_vali_y)):
            if pred_vali_y.iloc[j,0] <= 0.5:
                pred_vali_y.iloc[j,0] = 0
            else:
                pred_vali_y.iloc[j,0] = 1
        TP_vali = 0
        FP_vali = 0
        FN_vali = 0
        TN_vali = 0
        for k in range(len(pred_vali_y)):
            if pred_vali_y.iloc[k,0] == 1 and vali_y.iloc[k,0] == 1:
                TP_vali+=1
            elif pred_vali_y.iloc[k,0] == 1 and vali_y.iloc[k,0] == 0
                                                                :
                FP_vali+=1
            elif pred_vali_y.iloc[k,0] == 0 and vali_y.iloc[k,0] == 1
                                                                :
                FN_vali+=1
            else:
                TN_vali+=1
        if(TP_vali+FP_vali != 0):
            precision_vali.append(TP_vali/(TP_vali+FP_vali))
        else:
            precision_vali.append(None)
        recall_vali.append(TP_vali/(TP_vali+FN_vali))



    F1_train = [2*precision_train[i]*recall_train[i]/(precision_train
                                         [i]+recall_train[i]) for i
                                         in range(len(precision_train
                                         ))]
    F1_vali = [2*precision_vali[i]*recall_vali[i]/(precision_vali[i]+
                                         recall_vali[i]) for i in
                                         range(len(precision_vali))
                                         if precision_vali != None
                                         and (precision_vali[i]+
                                         recall_vali[i]) != 0]
    precision_vali = list(filter(None,precision_vali))
    return(np.mean(precision_train),np.mean(recall_train),np.mean(
                                         F1_train),np.mean(
                                         precision_vali),np.mean(
                                         recall_vali),np.mean(F1_vali
                                         ))
```

- Softmax Regression

```
# Transform the trainDY and testDY into two columns
trainDY["class2"] = 0
testDY["class2"] = 0

for i in range(949):
    if trainDY.iloc[i,0] == 0.0:
        trainDY.iloc[i,0] = 1
        trainDY.iloc[i,1] = 0
    else:
        trainDY.iloc[i,0] = 0
        trainDY.iloc[i,1] = 1

for j in range(151):
    if testDY.iloc[j,0] == 0.0:
        testDY.iloc[j,0] = 1
        testDY.iloc[j,1] = 0
    else:
        testDY.iloc[j,0] = 0
        testDY.iloc[j,1] = 1
```

```
# Define the softmax function
def softmax(X):

    num_sample, num_class = X.shape
    for i in range(num_sample):
        denom = np.exp(X[i]).sum()
        X[i] = np.exp(X[i])/denom
    return(X)
```

```
# Define gradient descent for softmax without regularization
# k implies the number of classes
def gradientDescent(X,Y,k):
    dataMatrix = np.mat(X)
    classMatrix = np.mat(Y)
    num_sample, num_parameter = dataMatrix.shape
    weight1 = np.ones([num_parameter,k])
    weight2 = np.ones([num_parameter,k])
    step_size = 1
    max_iter = 50000
    episilon = 0.000001
    num_iter = 0
    los = []
    while(num_iter < max_iter):
        num_iter += 1
        weight2 = weight1
        y = softmax(dataMatrix*weight1)
        error = y - classMatrix
        weight1 = weight1 - (step_size)*dataMatrix.transpose()*error/
                                                    num_sample
        if (abs(weight2[:,0]-weight1[:,0]) < episilon).all():
            break
    return weight1
```

- Softmax Regression with Regularization

```
# Define gradient descent for softmax with regularization
# k implies the number of classes
def gradientDescent_reg(X,Y,k,lam):
    dataMatrix = np.mat(X)
    classMatrix = np.mat(Y)
    num_sample, num_parameter = dataMatrix.shape
    weight1 = np.ones([num_parameter,k])
    weight2 = np.ones([num_parameter,k])
    step_size = 1
    max_iter = 2000
    episilon = 0.000001
    num_iter = 0
    los = []
    while(num_iter < max_iter):
        num_iter += 1
        weight2 = weight1
        y = softmax(dataMatrix*weight1)
        error = y - classMatrix
        weight1 = weight1 - (step_size)*(dataMatrix.transpose()*error
                                                    +lam*weight1)/num_sample
        if (abs(weight2[:,0]-weight1[:,0]) < episilon).all():
            break
    return weight1
```

```
#Define cross-validation for logistic with regularization to test
                                    accuracy
def cross_validation_regularize(X,Y,lam):
    #Create the fold to save precision
    precision_train = []
    precision_vali = []
    recall_train = []
    recall_vali = []
    accuracy_train = []
    accuracy_vali = []
    # Set the seed
    np.random.seed(1)
    # Divide indices of data into 10 folds randomly
    row = X.shape[0]
    index = [int(i) for i in np.linspace(0, row-1, row)]
    random.shuffle(index)
    # Use 10-fold cross validation to estimate the error rate
    folds = 10
    for i in range(0, folds):
        # For each fold, calculate the accuracy
        vali_index = index[int(row/folds*i):int(row/folds*(i+1))]
        # Train set has 9 folds and validation set has 1 fold
        vali_x = X.iloc[vali_index]
        vali_y = Y.iloc[vali_index]
        train_x = X.iloc[list(set(index)-set(vali_index))]
        train_y = Y.iloc[list(set(index)-set(vali_index))]
        temp_weight = gradientDescent_reg(train_x,train_y,2,lam)
        #predict result for the train set
        pred_train_y = softmax(train_x.to_numpy()*temp_weight)
        pred_train_y = pd.DataFrame(pred_train_y)
        temp_row, temp_col = pred_train_y.shape
        for j in range(temp_row):
            if pred_train_y.iloc[j,0] <= pred_train_y.iloc[j,1]:
                pred_train_y.iloc[j,0] = 0
                pred_train_y.iloc[j,1] = 1
            else:
                pred_train_y.iloc[j,0] = 1
                pred_train_y.iloc[j,1] = 0

        TP_train = 0
        FP_train = 0
        FN_train = 0
        TN_train = 0
        for k in range(temp_row):
            if pred_train_y.iloc[k,0] == 0 and train_y.iloc[k,0] == 0
                                                :
                TP_train+=1
            elif pred_train_y.iloc[k,0] == 0 and train_y.iloc[k,0] ==
                                                 1:
                FP_train+=1
            elif pred_train_y.iloc[k,0] == 1 and train_y.iloc[k,0] ==
                                                 0:
                FN_train+=1
            else:
                TN_train+=1
        precision_train.append(TP_train/(TP_train+FP_train))
        recall_train.append(TP_train/(TP_train + FN_train))
        accuracy_train.append((TP_train+TN_train)/(TP_train+TN_train+
                                            FP_train+FN_train))
        #predict result for the test set
        pred_vali_y = softmax(vali_x.to_numpy()*temp_weight)
        pred_vali_y = pd.DataFrame(pred_vali_y)
        temp_row2,temp_col2 = pred_vali_y.shape
        for j in range(temp_row2):
            if pred_vali_y.iloc[j,0] <= pred_vali_y.iloc[j,1]:
                pred_vali_y.iloc[j,0] = 0
                pred_vali_y.iloc[j,1] = 1
            else:
                pred_vali_y.iloc[j,0] = 1
                pred_vali_y.iloc[j,1] = 0
        TP_vali = 0
        FP_vali = 0
        FN_vali = 0
        TN_vali = 0
        for k in range(temp_row2):
            if pred_vali_y.iloc[k,0] == 0 and vali_y.iloc[k,0] == 0:
                TP_vali+=1
            elif pred_vali_y.iloc[k,0] == 0 and vali_y.iloc[k,0] == 1
                                                :
                FP_vali+=1
            elif pred_vali_y.iloc[k,0] == 1 and vali_y.iloc[k,0] == 0
                                                :
                FN_vali+=1
            else:
                TN_vali+=1
        if(TP_vali+FP_vali != 0):
            precision_vali.append(TP_vali/(TP_vali+FP_vali))
        else:
            precision_vali.append(0)
        recall_vali.append(TP_vali/(TP_vali+FN_vali))
        accuracy_vali.append((TP_vali+TN_vali)/(TP_vali+TN_vali+
                                             FP_vali+FN_vali))


    F1_train = [2*precision_train[i]*recall_train[i]/(precision_train
                                     [i]+recall_train[i]) for i
                                     in range(len(precision_train
                                     ))]
    F1_vali = [2*precision_vali[i]*recall_vali[i]/(precision_vali[i]+
                                     recall_vali[i]) for i in
                                     range(len(precision_vali))
                                     if precision_vali != None
                                     and (precision_vali[i]+
                                     recall_vali[i]) != 0]
    precision_vali = list(filter(None,precision_vali))
    return(np.mean(accuracy_train),np.mean(precision_train),np.mean(
                                     recall_train),np.mean(
                                     F1_train),np.mean(
                                     accuracy_vali),np.mean(
                                     precision_vali),np.mean(
                                     recall_vali),np.mean(F1_vali
                                     ))
```

- Decision Tree

```
def tree(para):
    train_acc = []
    train_pre = []
    train_rec = []
    train_f1 = []
    test_acc = []
    test_pre = []
    test_rec = []
    test_f1 = []
    for i in range(1,para):
        clf = DecisionTreeClassifier(max_depth=para)
        clf = clf.fit(trainDX,trainDY)
        train_predict = clf.predict(trainDX)
```

```
test_predict = clf.predict(testDX)
#Compute the training accuracy, precision, recall and F1-
                                                     score
TP_train = 0
TN_train = 0
FP_train = 0
FN_train = 0
for i in range(len(train_predict)):
    if (train_predict[i] == 0 and trainDY[i] == 0):
        TN_train += 1
    elif (train_predict[i] == 1 and trainDY[i] == 1):
        TP_train += 1
    elif (train_predict[i] == 1 and trainDY[i] == 0):
        FP_train += 1
    else:
        FN_train += 1
Accuracy_train = (TP_train+TN_train)/(TP_train+TN_train+
                                                     FP_train+FN_train)
Precision_train = TP_train/(TP_train + FP_train)
Recall_train = TP_train/(TP_train + FN_train)
F1_train = 2*Precision_train*Recall_train/(Precision_train +
                                                     Recall_train)
train_acc.append(Accuracy_train)
train_pre.append(Precision_train)
train_rec.append(Recall_train)
train_f1.append(F1_train)
#Compute the testing accuracy, precision, recall and F1-score
TP_test = 0
TN_test = 0
FP_test = 0
FN_test = 0
for i in range(len(test_predict)):
    if (test_predict[i] == 0 and testDY[i] == 0):
        TN_test += 1
    elif (test_predict[i] == 1 and testDY[i] == 1):
        TP_test += 1
    elif (test_predict[i] == 1 and testDY[i] == 0):
        FP_test += 1
    else:
        FN_test += 1
Accuracy_test = (TP_test+TN_test)/(TP_test+TN_test+FP_test+
                                                     FN_test)
Precision_test = TP_test/(TP_test + FP_test)
```

```
        Recall_test = TP_test/(TP_test + FN_test)
        F1_test = 2*Precision_test*Recall_test/(Precision_test +
                                                 Recall_test)
        test_acc.append(Accuracy_test)
        test_pre.append(Precision_test)
        test_rec.append(Recall_test)
        test_f1.append(F1_test)
    return(train_acc,train_pre,train_rec,train_f1,test_acc,test_pre,
                                             test_rec,test_f1)
```

- Random Forest

```
# External package
from sklearn.ensemble import RandomForestClassifier
```

```
def rf_train(trainX, trainY, num_tree):
    # Set the seed
    random.seed(1)
    # Divide indices of data into 5 folds randomly
    index0 = [i for i in range(len(trainY)) if trainY[i]==0]
    index1 = [i for i in range(len(trainY)) if trainY[i]==1]
    index = index0+index1
    random.shuffle(index0)
    random.shuffle(index1)
    Pred = []
    Label = []
    # Use 10-fold cross validation
    folds = 10
    for i in range(0, folds):
        # For each fold, calculate the accuracy
        vali_index = index0[int(len(index0)/folds*i):int(len(index0)/
                                                 folds*(i+1))] + index1[
                                                 int(len(index1)/folds*i)
                                                 :int(len(index1)/folds*(
                                                 i+1))]
        # Train set has 10 folds and validation set has 1 fold
        vali_x = trainX.iloc[vali_index]
        vali_y = trainY[vali_index]
        train_x = trainX.iloc[list(set(index)-set(vali_index))]
        train_y = trainY[list(set(index)-set(vali_index))]
        forest = RandomForestClassifier(n_estimators=num_tree,
                                                 max_features="sqrt",
                                                 oob_score=True,
                                                 random_state=0)
        model = forest.fit(train_x, train_y)
        pred = model.predict(vali_x)
        Pred += list(pred)
        Label += list(vali_y)
    TP = len([i for i in range(len(Pred)) if Pred[i]==1 and Label[i]=
                                                  =1])
    FP = len([i for i in range(len(Pred)) if Pred[i]==1 and Label[i]=
                                                  =0])
    TN = len([i for i in range(len(Pred)) if Pred[i]==0 and Label[i]=
                                                  =0])
    FN = len([i for i in range(len(Pred)) if Pred[i]==0 and Label[i]=
                                                  =1])
    P = TP/(TP+FP)
    R = TP/(TP+FN)
    F1 = 2*P*R/(P+R)
    A = (TP+TN)/(TP+TN+FP+FN)
    return((P,R,F1,A))
```

```
# Random forest testing
forest = RandomForestClassifier(n_estimators=30, max_features="sqrt",
                                      oob_score=True, random_state=0)
model = forest.fit(trainX,trainY)
pred = model.predict(testX)
TP = len([i for i in range(len(pred)) if pred[i]==1 and testY[i]==1])
FP = len([i for i in range(len(pred)) if pred[i]==1 and testY[i]==0])
TN = len([i for i in range(len(pred)) if pred[i]==0 and testY[i]==0])
FN = len([i for i in range(len(pred)) if pred[i]==0 and testY[i]==1])
P = TP/(TP+FP)
R = TP/(TP+FN)
F1 = 2*P*R/(P+R)
acc = (TP+TN)/(TP+TN+FP+FN)
print(acc,P,R,F1)
```

- Neural Network

```
import tensorflow.compat.v1 as tf
tf.compat.v1.disable_eager_execution()
```

```
def Fully_neural_network(X):
    layer_1 = tf.nn.sigmoid(tf.add(tf.matmul(X, w1), b1))
    layer_2 = tf.nn.sigmoid(tf.add(tf.matmul(layer_1, w2), b2))
    layer_out = tf.matmul(layer_2, w_out) + b_out
    return layer_out
```

```
# features
xs=tf.placeholder(tf.float32,[None,64])
# labels
ys=tf.placeholder(tf.float32,[None,1])
# weight matrix
w1 = tf.Variable(tf.random_normal([len(trainDX[0]), 64], stddev=1,
                                     seed=1))
w2 = tf.Variable(tf.random_normal([64, 64], stddev=1, seed=1))
w_out = tf.Variable(tf.random_normal([64, 2], stddev=1, seed=1))
# bias matrix
b1 = tf.Variable(tf.random_normal([64]))
b2 = tf.Variable(tf.random_normal([64]))
b_out = tf.Variable(tf.random_normal([2]))
```

```
net_out = Fully_neural_network(xs)
# Probability of 2 classes
pre = tf.nn.softmax(net_out)
# Label output
pre1 = tf.argmax(pre, 1)
# Cross Entropy
loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=
                                     net_out, labels=ys))
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
train_op = optimizer.minimize(loss)
correct_pre = tf.equal(tf.argmax(pre, 1), tf.argmax(ys, 1))
accuracy = tf.reduce_mean(tf.cast(correct_pre, tf.float32))
init = tf.global_variables_initializer()
with tf.Session() as sess:
    sess.run(init)
    for i in range(1, iteration_times + 1):
        start = (i * batch_size) % len(trainDX)
        end = min(start + batch_size, len(trainDX))
        batch_x = trainDX[start:end]
        batch_y = trainDY[start:end]
        sess.run(train_op, feed_dict={xs: batch_x, ys: batch_y})
        if i % 100 == 0 or i == 1:
            l, acc = sess.run([loss, accuracy], feed_dict={xs:
                                                 trainDX, ys: trainDY
                                                 })
            print("Iteration " + str(i) + ", Cross Entropy= " + "{:.
                                                 4f}".format(
                l) + ", Training Accuracy= " + "{:.3f}".format(acc))
            w1nn,w2nn,wnn_out,b1nn,b2nn,bnn_out = sess.run([w1,w2,
                                              w_out,b1,b2,b_out])
```

```
with tf.Session() as sess:
    layer_1 = sess.run(tf.nn.sigmoid(tf.add(tf.matmul(testDX, w1nn),
                                      b1nn)))
    layer_2 = sess.run(tf.nn.sigmoid(tf.add(tf.matmul(layer_1, w2nn),
                                       b2nn)))
    layer_out = sess.run(tf.matmul(layer_2, wnn_out) + bnn_out)

    pre = sess.run(tf.nn.softmax(layer_out))
    p1 = sess.run(tf.argmax(pre, 1))
    correct_pre = sess.run(tf.equal(tf.argmax(pre, 1), tf.argmax(
                                      testDY, 1)))
```

```
from sklearn.metrics import roc_curve, auc
Prob1 = [prob[i][1] for i in range(len(prob1))]
Prob2 = [prob[i][1] for i in range(len(prob))]
plt.figure(figsize=(10,6))
FPR1, TPR1, threshold1 = roc_curve(testDY.reshape(151,), Prob1,
                                     pos_label=1)
FPR2, TPR2, threshold2 = roc_curve(trainDY.reshape(949,), Prob2,
                                     pos_label=1)
roc_auc1 = auc(FPR1, TPR1)
roc_auc2 = auc(FPR2, TPR2)
plt.plot(FPR1, TPR1, color='b',lw=2, label='Neural Network Test'+' (
                                     area = %0.3f)'%roc_auc1,
                                     linestyle='--')
plt.plot(FPR2, TPR2, color='r',lw=2, label='Neural Network Train'+' (
                                     area = %0.3f)'%roc_auc2,
                                     linestyle='--')
plt.plot([0,1], [0,1], color = 'navy', lw=2, linestyle='--')
plt.grid()
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("ROC Curve of NN Classifier")
plt.legend()
plt.show()
```

- SMOTE

```
# Hand written oversampling method
trainData = pd.read_table("NormalTrain.csv",sep=",")
unblcData = trainData[trainData["class"]==1][trainData.columns.values
                                     [:-1]]
```

```
N = unblcData.shape[0]
random.seed(1)
k = 2
new_df = pd.DataFrame(columns=trainData.columns.values)
count = 0
for i in range(N):
    sample = unblcData.iloc[[i]]
    s = list(unblcData.iloc[i])
    others = unblcData.iloc[[j for j in range(N) if j != i]]
    diff = np.tile(sample, (N-1, 1)) - others
    dis = np.sqrt((diff**2).sum(axis=1))
    dis_dict = pd.DataFrame(dis).to_dict()[0]
    sort_dis = sorted(dis_dict.items(), key=lambda item:item[1])
    for a in range(k):
        r = random.uniform(0,1)
        ele = list(trainData.iloc[sort_dis[a][0]][trainData.columns.
                                                     values[:-1]])
        add_ele = r*np.array(s)+(1-r)*np.array(ele)
        new_df.loc[count] = list(add_ele)+[1]
        count += 1
overSample = trainData.append(new_df)
overSample["class"] = [int(i) for i in overSample["class"]]
overSample.to_csv("OverSample.csv",index=False)
```

References

[1] Disha, R. A. Waheed, S. (2022) Performance analysis of machine learning models for intrusion detection system using Gini Impurity-based Weighted Random Forest (GIWRF) feature selection technique. Cybersecurity. [Online] 5 (1), 1–22.

[2] Carrington, A. M. et al. (2020) A new concordant partial AUC and partial c statistic for imbalanced data in the evaluation of machine learning algorithms. BMC medical informatics and decision making. [Online] 20 (1), 4–12.