**Individual Project Report**

Zekai Zhang 1930026166

Date: 2021/11/20

目录

[Workflow 2](#_Toc88316347)

[The models adopted 6](#_Toc88316348)

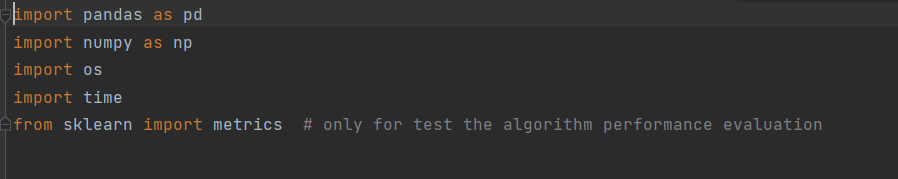
[KNN algorithm 6](#_Toc88316349)

[Naïve Bayes algorithm 9](#_Toc88316350)

[Perceptron algorithm 12](#_Toc88316351)

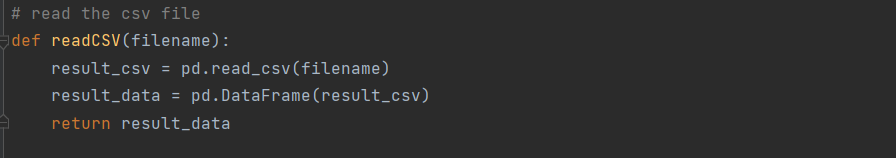
# Workflow

Step 1, we import the library we need to use

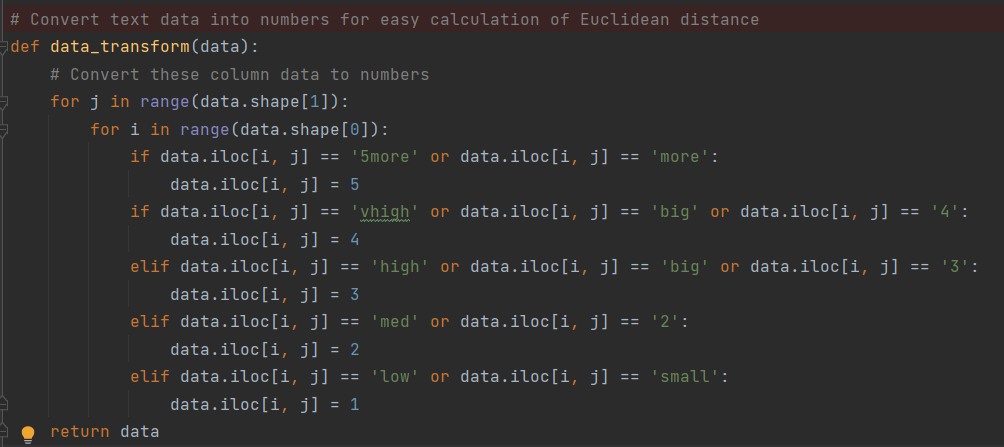


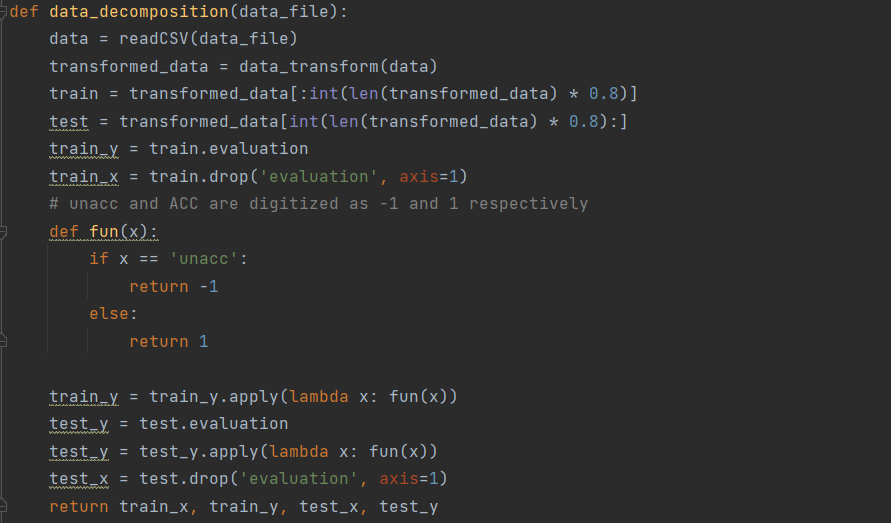
There are some libraries that need to be downloaded. We can use pip install -i

Step 2, write a function that reads a CSV file and converts it to a DataFrame

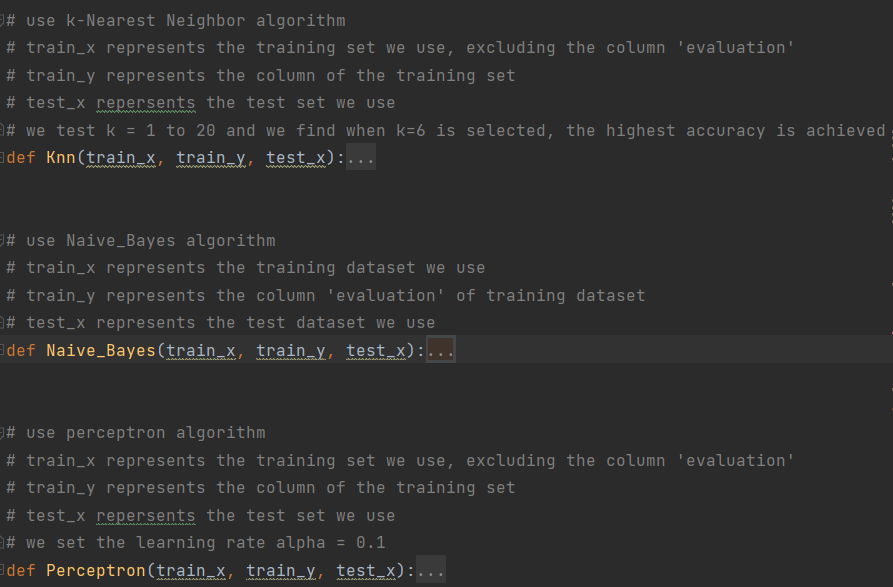


Step 3, We quantified the non-digital data in the data set into numbers for the convenience of algorithm calculation

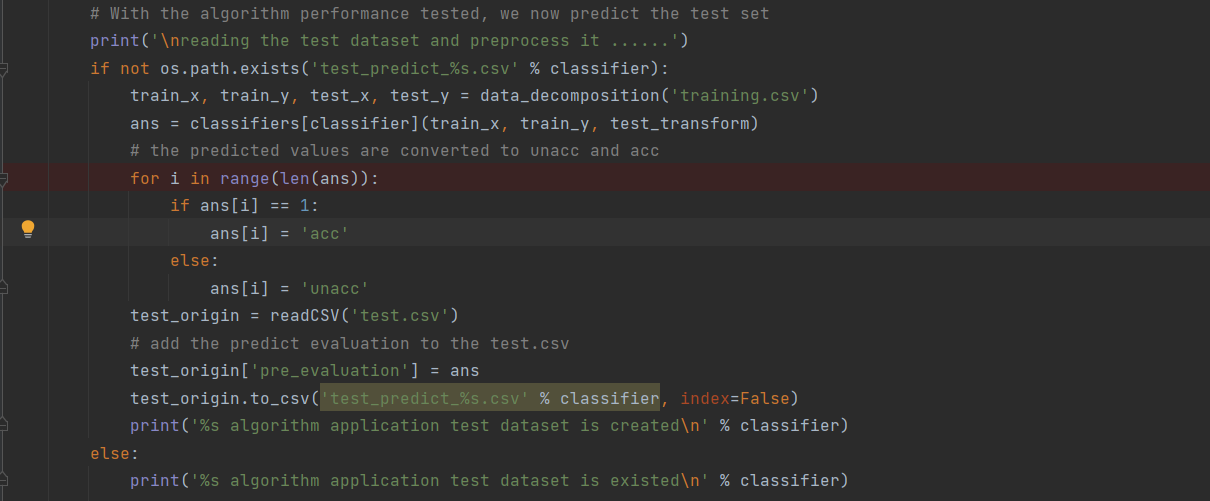


Step 4, We split the data set into training set and test set in an 8:2 ratio, then take ‘unacc’ as a negative case and ‘acc’ as a positive case, and convert to -1 and 1

Step 5, Independent implementation of three classification algorithm code



Step 6, write the main function, the first step is to read the training set and test set, then use a for loop to run the three models and output the performance evaluation results, and finally determine whether the test set prediction file has been exported, if not, export the prediction results file



# The models adopted

## KNN algorithm

Input:

train\_x: the training set we use, excluding the column 'evaluation'

train\_y: the column ‘evaluation’ of the training set

test\_x: the test set we use

Step 0, we convert the format of the dataset to a matrix for subsequent calculation

train\_group = train\_x.values  
labels = train\_y.tolist()  
test\_x = test\_x.values

Step 1, through the definition of KNN algorithm to write the program, first calculate the distance between the test group and all the tuples of the training set, then square the results and add them up, finally take the root.

newInput = test\_x[0]  
for i in range(test\_x.shape[0]):  
 newInput = test\_x[i]

# Step 1: the following copy numSamples rows for train\_group  
diff = np.tile(newInput, (train\_group.shape[0], 1)) - train\_group # diff by element

squareDiff = diff \*\* 2  
squareSum = squareDiff.sum(axis=1) # add up by row  
distance = squareSum \*\* 0.5 # Take the square root and get the Euclidean distance

Step 2, Sort the results

# Step 2: Sort these distances  
sortedDistance = distance.argsort()

classCount = {}

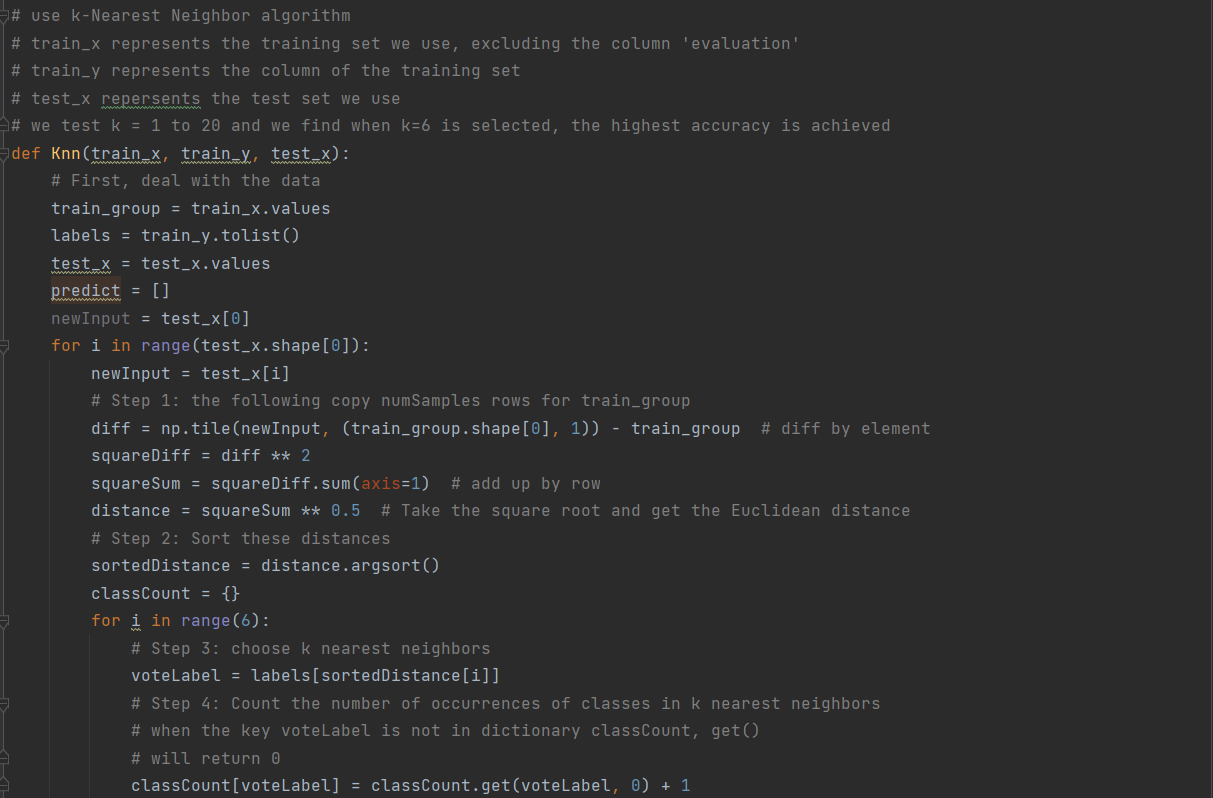
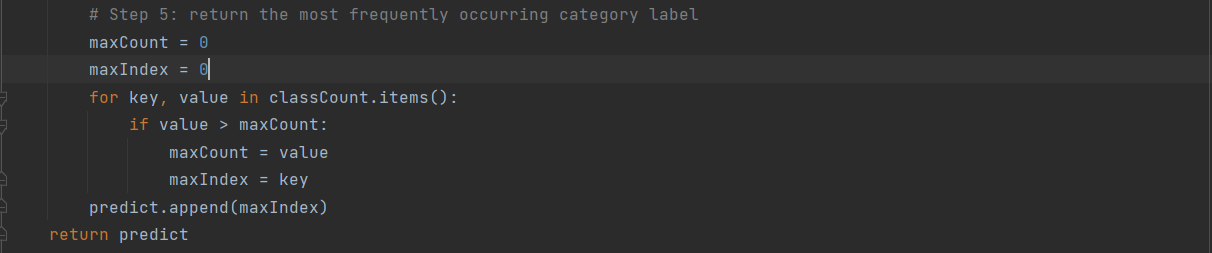
Step 3 and 4, choose k nearest neighbors and count the number of occurrences of classes in k nearest neighbors

for i in range(6):  
 # Step 3: choose k nearest neighbors  
 voteLabel = labels[sortedDistance[i]]  
 # Step 4: Count the number of occurrences of classes in k nearest neighbors  
 # when the key voteLabel is not in dictionary classCount, get()  
 # will return 0  
 classCount[voteLabel] = classCount.get(voteLabel, 0) + 1

Step 5, return the most frequently occurring category label

# Step 5: return the most frequently occurring category label  
 maxCount = 0  
 maxIndex = 0  
 for key, value in classCount.items():  
 if value > maxCount:  
 maxCount = value  
 maxIndex = key  
 predict.append(maxIndex)  
return predict

The complete code:



Output:

predict: a list of the predict value of the test set

## Naïve Bayes algorithm

Input:

train\_x: the training set we use, excluding the column 'evaluation'

train\_y: the column ‘evaluation’ of the training set

test\_x: the test set we use

Step 1: compute the prior probability for each class

# first compute the prior probability for each class  
P\_acc = train\_y.tolist().count(1) / train\_y.shape[0]  
P\_unacc = train\_y.tolist().count(-1) / train\_y.shape[0]

Step 2: create two matrices to store , which can greatly reduce the time complexity of the algorithm

P1 = np.zeros((6, 6))  
P2 = np.zeros((6, 6))

compute every

for attribute in range(6):  
 for index in range(train\_x.shape[0]):  
 ans = train\_x.iloc[index, attribute]  
 if train\_y[index] == 1:  
 P1[attribute][ans] = P1[attribute][ans] + 1  
 else:  
 P2[attribute][ans] = P2[attribute][ans] + 1

apply the Laplace correction

P1 = (P1 + 1) / (train\_y.tolist().count(1) + 2)  
P2 = (P2 + 1) / (train\_y.tolist().count(-1) + 2)

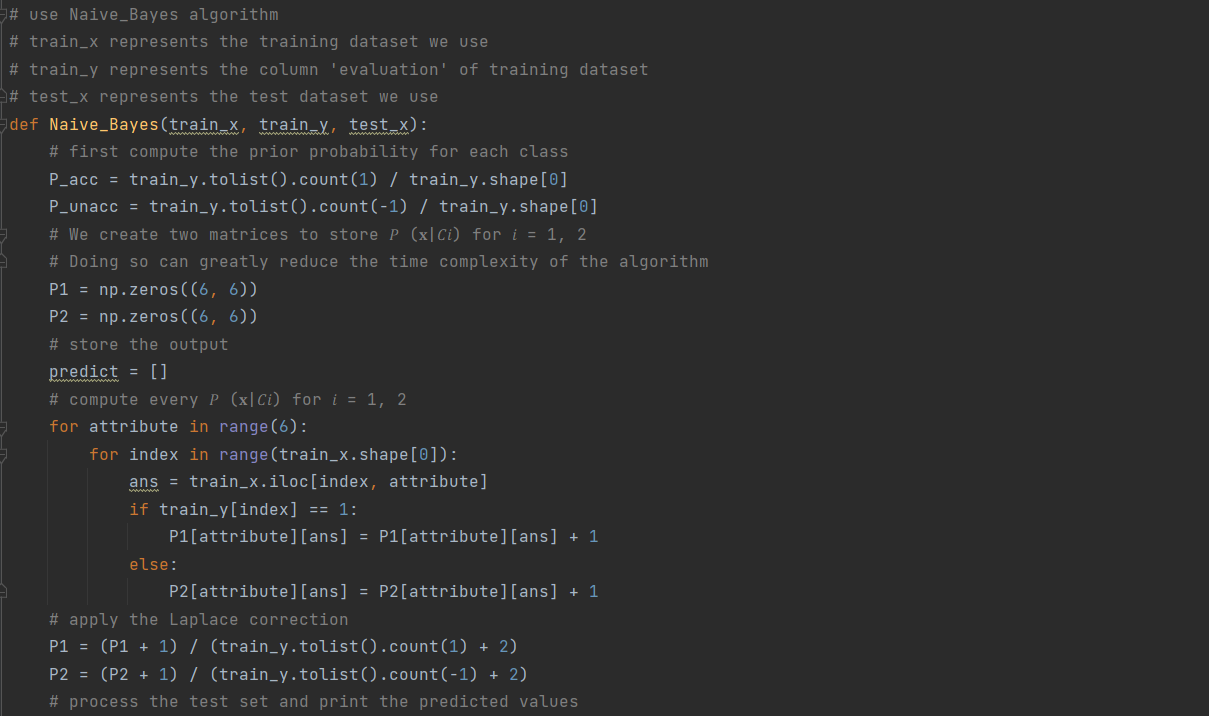
Now we have two matrices, no matter what the properties of the test set are, we can find the from the matrix directly.

process the test set and output the predict values

for i in range(test\_x.shape[0]):  
 Px\_1 = P\_acc  
 Px\_2 = P\_unacc  
 newInput = test\_x.iloc[i, :]  
 for i in range(6):  
 value = newInput[i]  
 Px\_1 = Px\_1 \* P1[i][value]  
 Px\_2 = Px\_2 \* P2[i][value]  
 if Px\_1 > Px\_2:  
 predict.append(1)  
 else:  
 predict.append(-1)

return predict

The complete code:



Output:

predict: a list of the predict value of the test set

## Perceptron algorithm

Input:

train\_x: the training set we use, excluding the column 'evaluation'

train\_y: the column ‘evaluation’ of the training set

test\_x: the test set we use

Step 1, we deal with the data and convert it to a matrix

(Add a column ‘I0’ with all numbers 1)

train\_x.insert(0, 'I0', 1)  
train\_x = np.matrix(train\_x)  
test\_x.insert(0, 'I0', 1)  
test\_x = np.matrix(test\_x)

Step 2, write a function which return 1 when the answer >=0 and return -1 when the answer < 0

def score(x, y):  
 if x \* y.T < 0:  
 return -1  
 else:  
 return 1

Step 3, create a new matrix to store the weight

weight = [0 for item in range(7)]  
# convert it to matrix  
weight = np.matrix(weight)

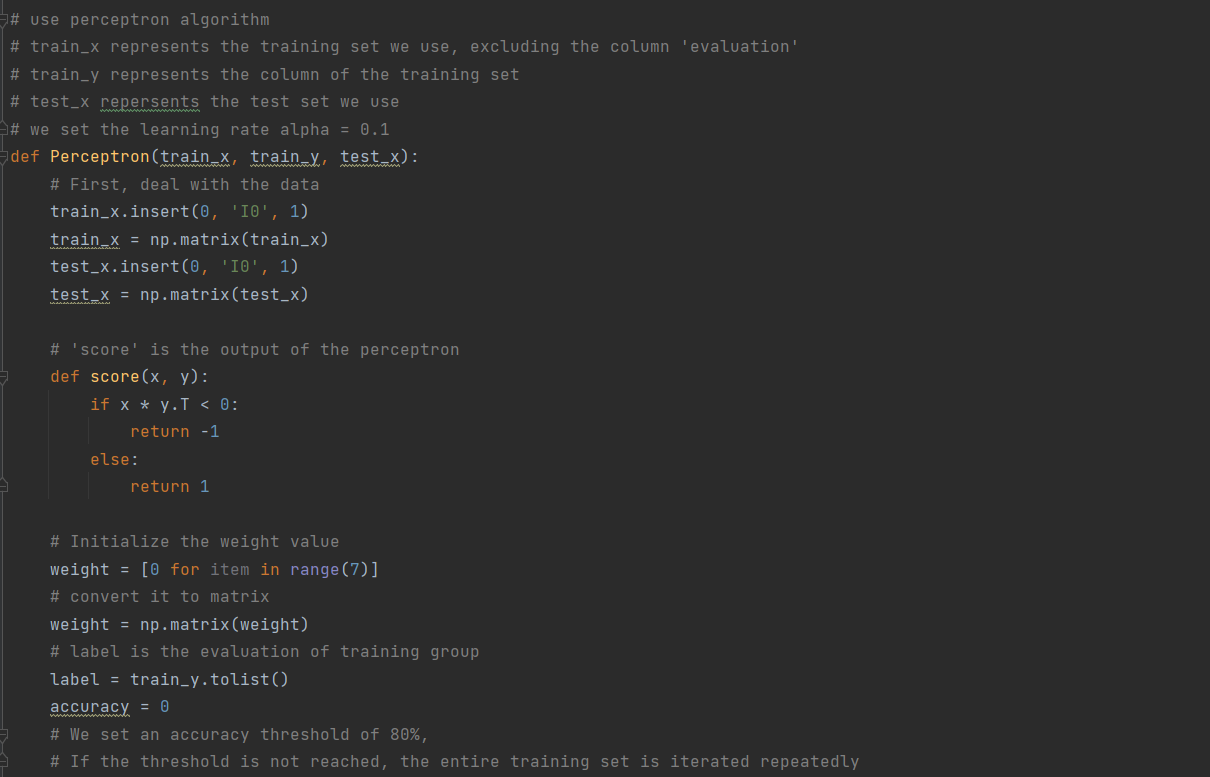
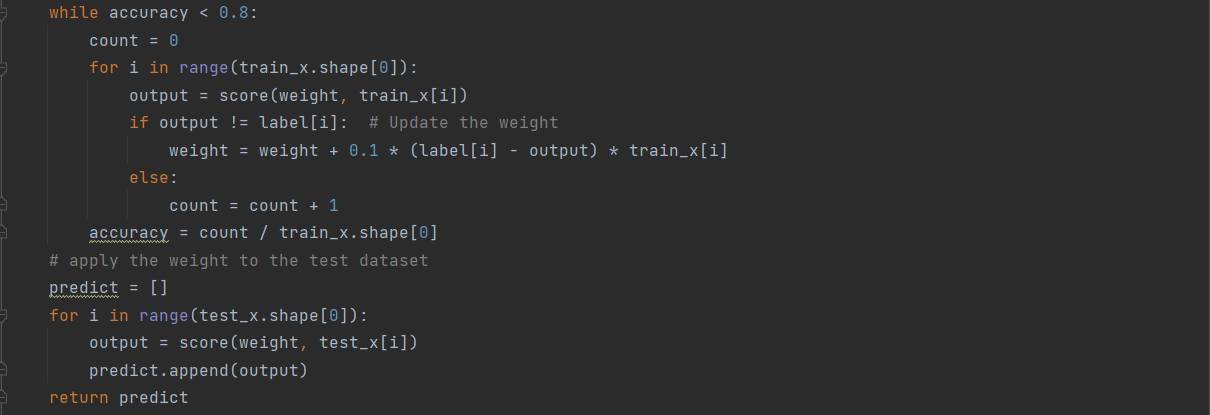
Step 4, set an accuracy threshold of 80%, If the threshold is not reached, the entire training set is iterated repeatedly. Finally we get the proper weight

while accuracy < 0.8:  
 count = 0  
 for i in range(train\_x.shape[0]):  
 output = score(weight, train\_x[i])  
 if output != label[i]: # Update the weight  
 weight = weight + 0.1 \* (label[i] - output) \* train\_x[i]  
 else:  
 count = count + 1  
 accuracy = count / train\_x.shape[0]

Step 5, apply the weights to the test set and return the predicted results

predict = []  
for i in range(test\_x.shape[0]):  
 output = score(weight, test\_x[i])  
 predict.append(output)  
return predict

The complete code:

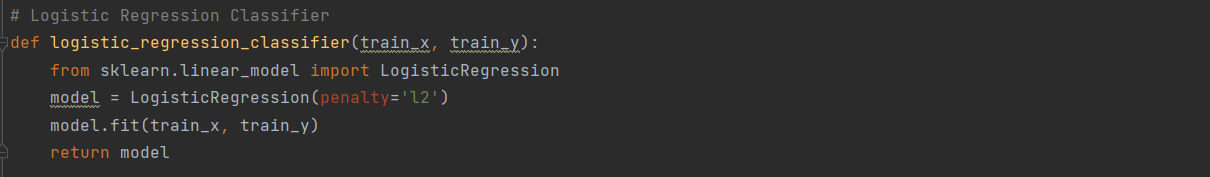


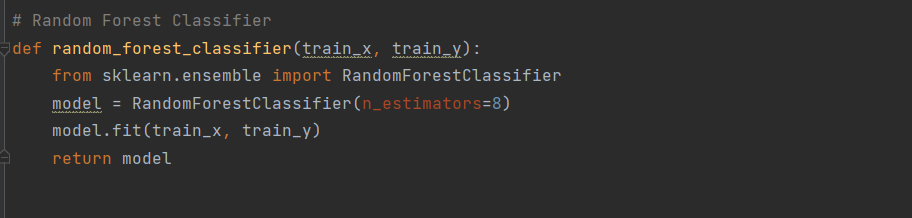
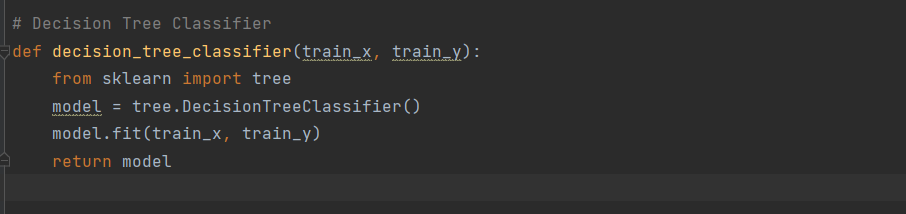
Output:

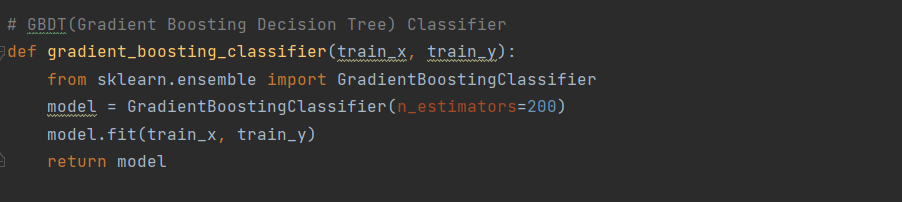
predict: a list of the predict value of the test set

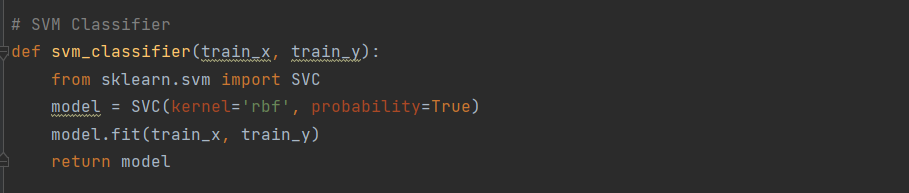
## Other models with SKLearn Library

1. Logistic Regression Classifier

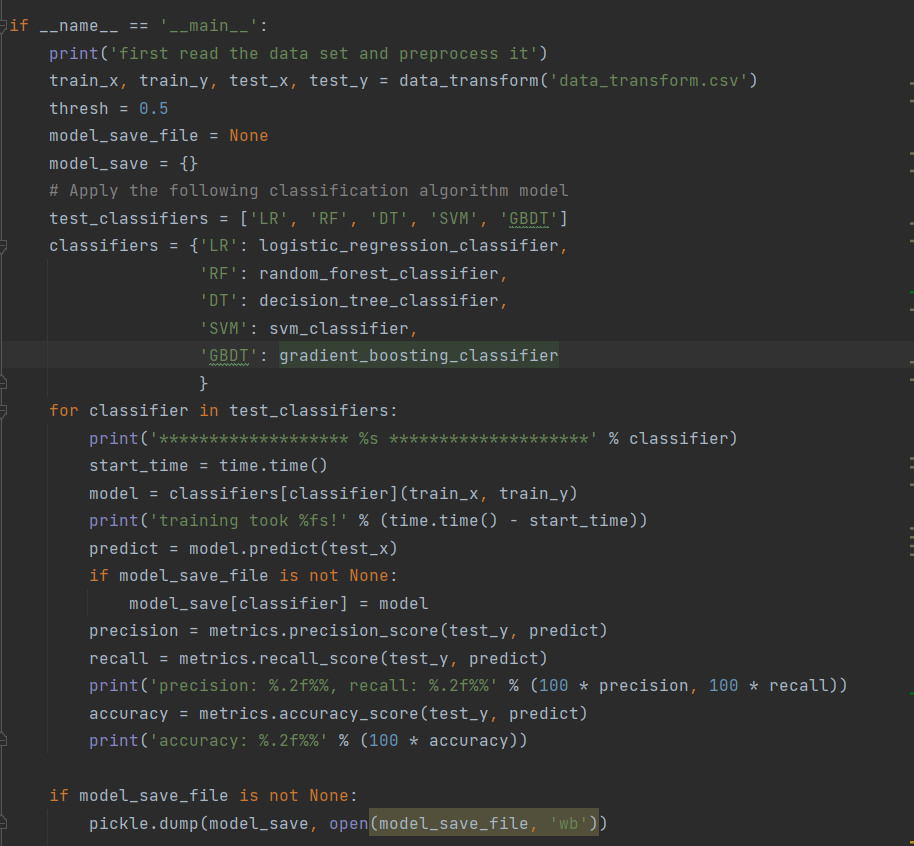


1. Random Forest Classifier
2. Decision Tree Classifier
3. GBDT (Gradient Boosting Decision Tree) Classifier



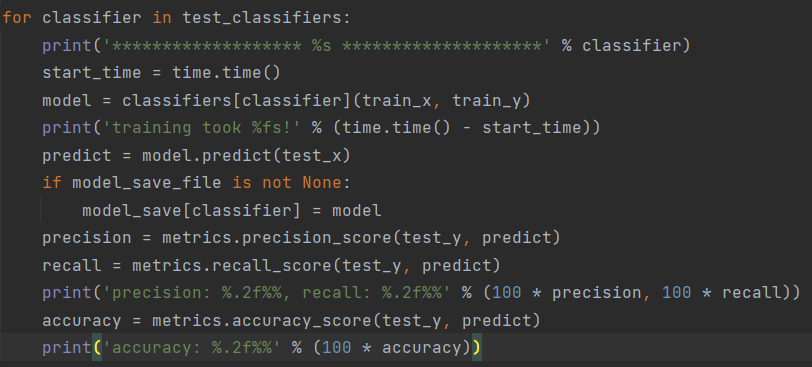
1. SVM Classifier

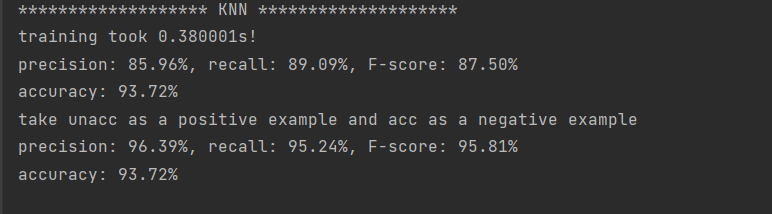
The main function:

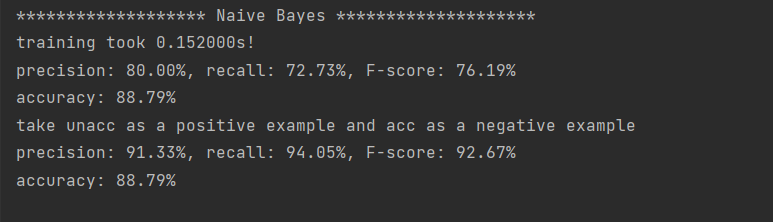


# Experimental results

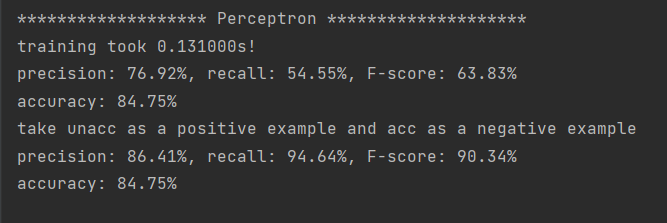
Use metrics to compute accuracy/precision/recall/F-score



1.  KNN algorithm
2. Naïve Bayes algorithm



1. Perceptron algorithm



# Result analysis

1. Which model achieves the BEST performance on this dataset? Why?

KNN algorithm is the BEST performance on this dataset, whose precision/recall/F-score/accuracy is:

96.39%/ 95.24%/ 95.81%/ 93.72%

Because the KNN algorithm has the highest complexity, it needs to work out the Euclidean distance between the data and all the data of the training set, so the accuracy is higher.

b) Conduct error analysis for the models that do not perform well.

The perceptron algorithm has poor performance, precision: 76.92%, recall: 54.55%, F-Score: 63.83%, Accuracy: 84.75%, because we use the simplest perceptron algorithm without penalty function, the only thing to do is to set the accuracy threshold (if the threshold is not reached, the whole training set will be repeated).