In terms of the implementation, the **table** below shows the function, classifier, parameter that I defined and the corresponding explanation.

Function, classifier, parameter	Explanation
get_macro_f1()	Calculate the macro_f1 score.
get_confusion_matrix()	Draw the confusion matrix.
reduce_label_num()	Reduce 5 labels to 3 labels.
save_result()	Save the prediction results to a file.
Naive Bayes classifier	- The training data should be imported in the constructor. - The number of labels need to set to 5 in the constructor.
Vocab	It includes all the words in the training data with no duplicate words.
_words_in_vocab()	- The input parameters of this function are the words in a whole sentence. - Record the subscript positions of these words in vocab and store them in a vector (list). - If there are duplicate words in the sentence, the corresponding value will be increased, and then return the list.

At the training stage, as it shown in the following equation, the final task is to compare **P(A|B)**. Since each item has the same **P(B)**, so we just need to compare **P(A)P(B|A)**. In my implementation, in order to prevent underflow in the process of multiplication, it is necessary to calculate the natural logarithm of the result, which means compare the **log(P(A)) + log(P(B|A))**.

$$P(A|B) = \frac{P(A)P(B|A)}{P(B)}$$

In the **train()** function, I construct the **word_matrix** first which record the word vector of each sentence in the training data. After the **P(A)** is calculated, the result is a list, and each value of the list records the probability of the class. Then **P(B|A)** is calculated. For each of these classes, a list of **word_distribution** is created to store the number of words belonging to the class, it initializes to 1 to in cause prevent the words that did not exist before. i.e., the probability is 0. Next, I define the **word_sum** which present the total number of words belongs to that class, and it initializes to 2 to avoid the probability of 0. For each word list in **word_matrix**, the word vector of the sentence is added to its corresponding **word_distribution**, and the number of words is summed up and added to **word_sum**. Finally, divide **word_distribution** with the data for each class in **word_sum** so that we can get **P(B|A),** then calculate its logarithm.

In the prediction stage, the word vector of test data is first obtained. With each sentence of each class to calculate the **log(P(A)) + log(P(B|A))**, store it in the result. And return the index of the maximum value in result, which means the result of classification.

Once the model is built, the prediction process begins. First, it predicts the data of 5 labels. The f1 score of the prediction result is 0.49. Then, reduce the number of labels to 3, and inherit the classifier **NB_5**. Set **Label_num** to 3 so that we can get the classifier with 3, and then it starte to predict again. The f1 score of the prediction result is 0.69, which can be recognized a

big improvement over the result of 5 labels.

Then, a new Naive Bayes classifier is designed. Before training the model, I preprocess the data. These include changing all words to lowercase and removing punctuation from words. After that, the **_words_in_vocab()** function no longer adds the number of repeating words. Instead, after setting it to 1 and modifying it, the new classifier start to classify, and the F1 score is 0.51. As it mentioned above, we can figure that the new classifier made some progresses according to the **LEFT** confusion matrix below. The prediction results are then saved in the TSV file.

Based on the new classifier I design before, there is a new classifier for the class 3 after I set the **label_num** to 3. And I use this classifier to predict the data of three classes, the F1 score is 0.70. We can figure that the classifier made some progresses according to the **RIGHT** confusion matrix below.

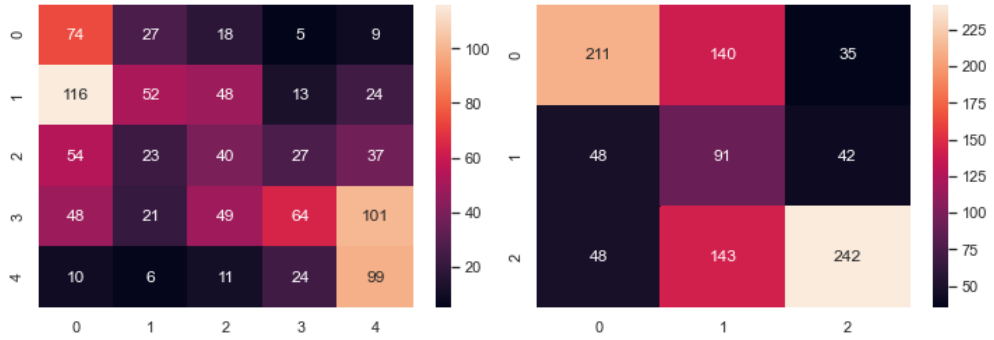

Table below shows the performance results under different configurations.

Classifier	F1 score
NB_5	0.49
NB_3	0.69
NB_5 with process	0.51
NB_3 with process	0.7

Finally, we can see that the latter model works better, so we use the latter model to predict test data and save the results to the file. By comparing these models, we can see that the efficiency of prediction can be improved by ignoring the punctuation of the sentences.

A comparison of these models shows that ignoring the punctuation of sentences can have some improvement on the efficiency of prediction. If the classes of classification are reduced there will be a big improvement on the classification results.