**Classification Analysis Report for Dataset**

Contents

[The workflow 2](#_Toc59055078)

[The models adopted 2](#_Toc59055079)

[*KNN model (K-Nearest-Neighbors Classification)* 2](#_Toc59055080)

[*Naïve Bayes model* 3](#_Toc59055081)

[*Perceptron model* 5](#_Toc59055082)

[*Decision Tree model* 7](#_Toc59055083)

[*SVM model* 8](#_Toc59055084)

[Experimental results of different models 8](#_Toc59055085)

[*KNN model (K-Nearest-Neighbors Classification)* 9](#_Toc59055086)

[*Naïve Bayes model* 10](#_Toc59055087)

[*Perceptron model* 11](#_Toc59055088)

[*Decision Tree model* 12](#_Toc59055089)

[*SVM model* 13](#_Toc59055090)

[Result analysis 14](#_Toc59055091)

# The workflow

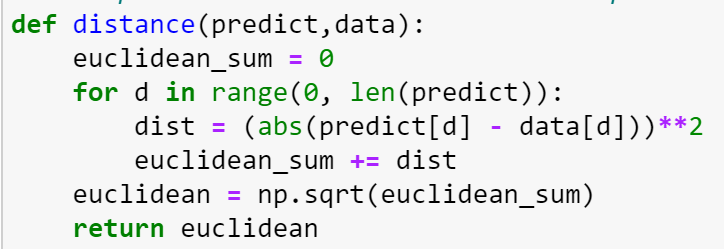
In this report, three models (KNN, Naïve Bayes and Perceptron) implemented by own will be written down with more detail and the rest of two models are implemented with python library ***scikit-learn***. All the models are shown based on ***Dataset.xlsx*** file which contains nine different attributes with three classes. Similarly, the first procedure for building a classification model is to import and preprocess data. To offer a good performance for each model, training data are split into two parts for training and testing respectively. In the following content, these five models are presented one by one. Since all models have the same procedure for importing and splitting data, hence the detail of this procedure is ignore. Then it is necessary to show experimental result. And eventually we do result analysis to figure out which model has the best performance and find the reason for those models that are not good enough.

# **The models adopted**

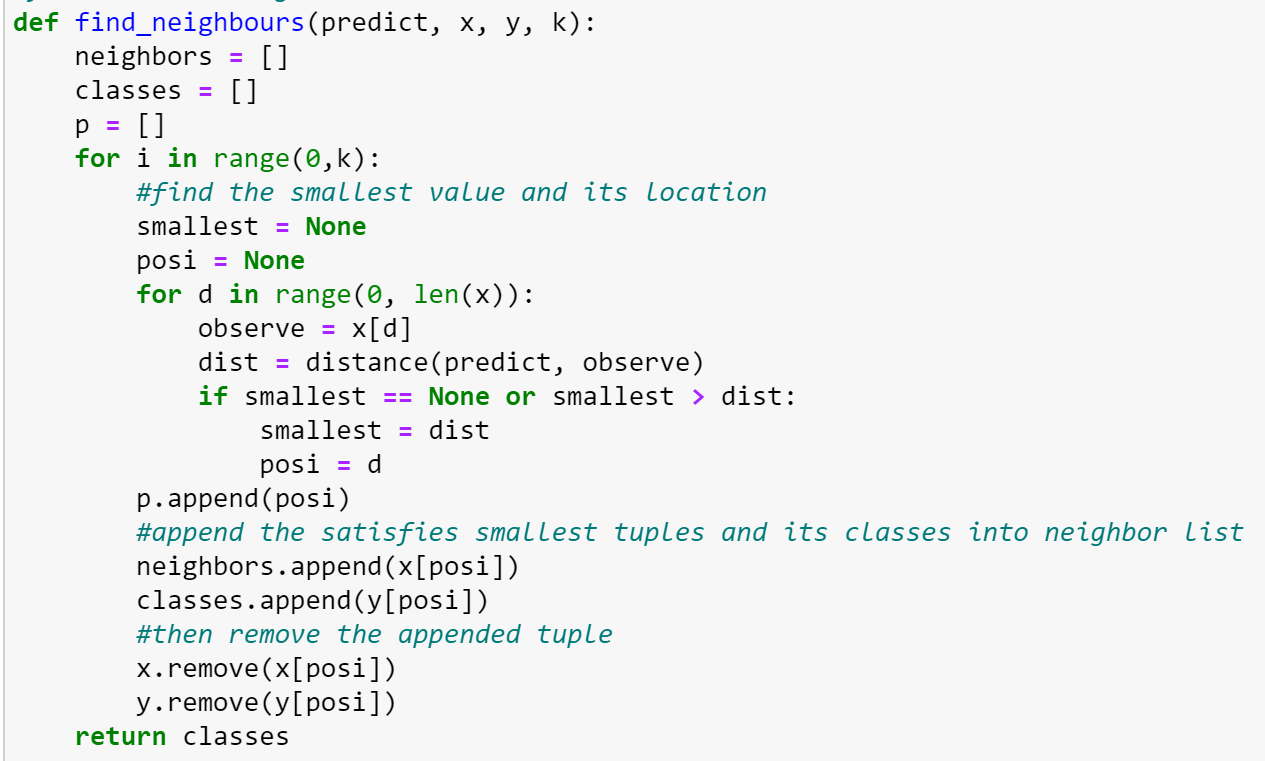
## *KNN model (K-Nearest-Neighbors Classification)*

In this model, we mainly divide it into two sections: calculate Euclidean distance, find neighbors the majority classes.

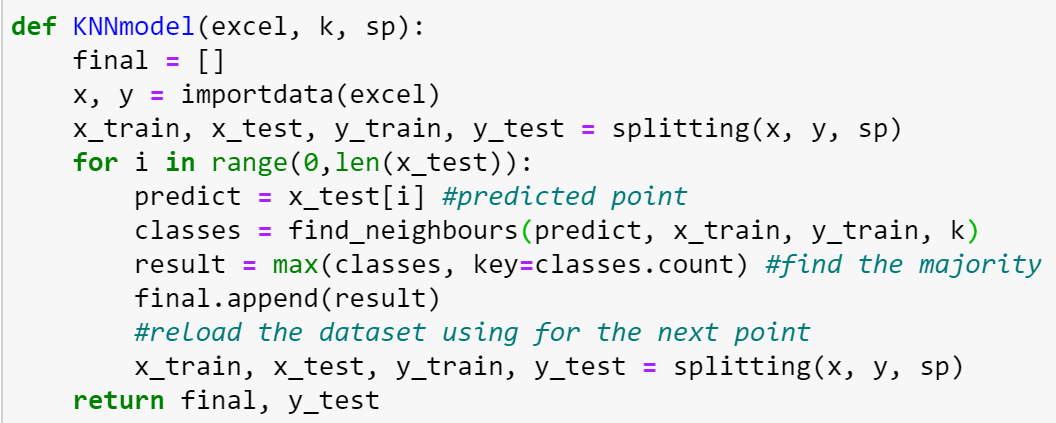
***Distance(predict, data)*** calculates the Euclidean distance between predicted tuples and each observed data tuple.



Then through comparing to each distance, function ***find\_neighbours(predict, x, y, k)*** find a list of the smallest element based on k value. Input ***predict*** represents each test tuple and ***x, y*** stand for the training set. And value ***k*** determines how many smallest elements should be included. In the function, we do looping for iterating every k value, when we traverse each ***k*** value, the smallest element is appended to ***neighbors*** list and removed from the original dataset for further searching.



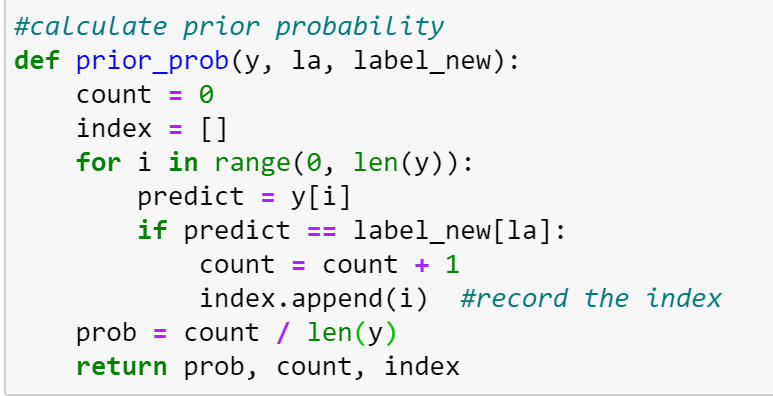
For every tuple in test set(***x\_test***), we find its neighbors based on k value and then figure out the majority class as the result of tuple. After looping one test tuple, we need to recover the original training dataset for searching the following tuples. In the following function ***KNNmodel(excel, k, sp)***, the input ***k*** and ***sp*** stand for k value and splitting cut respectively.



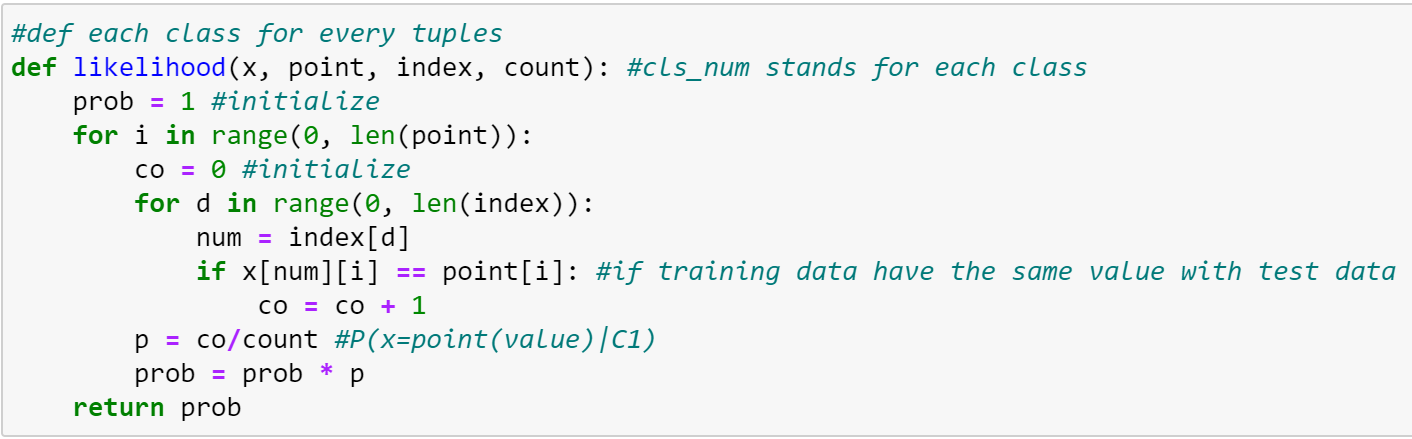
## *Naïve Bayes model*

In this model, we assume that the attributes are conditionally independent, hence Naïve bayes approach are implemented. The model are mainly divided into three sections: calculating the prior probability, obtaining and finding maximized .

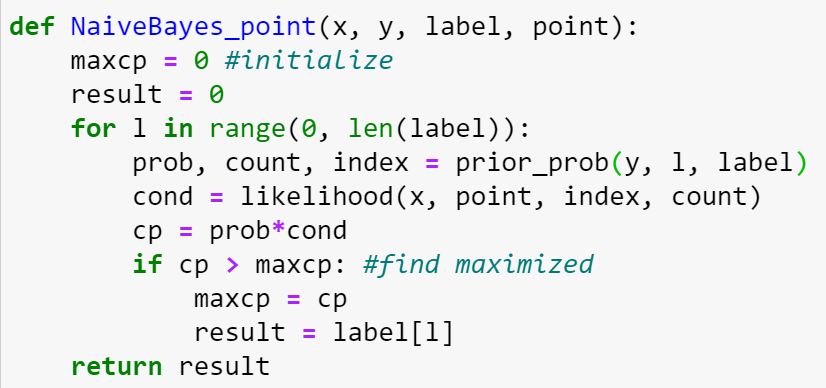
To calculate prior probability, ***prior\_prob(y, la, label\_new)*** is shown as below. There are three parameters as input where the input ***y*** is training data classes for classifying the classes respectively. As for the second and last one, ***la*** is an index in list ***label\_new***where ***label\_new*** contains three kinds of classes . Then we can compute respectively, which is the first output ***prob*** in the function. The second and third output ***count***, ***index*** are preparing for the following function to compute .



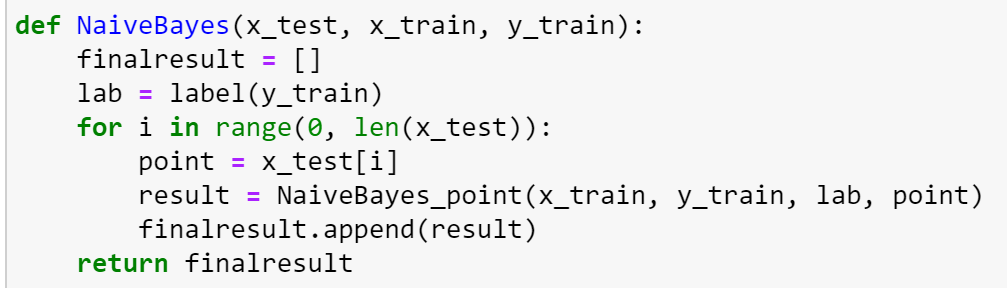
Next, ***likelihood(x,point,index,count)*** is to calculate respectively for every test tuple. Firstly do initialization for probabilities ***prob*** and count ***co***. Then for each attribute, count the number if tuple’s value is same as the value in the selected training set where the selected one is recorded by ***index***. After computing each , we calculate the sum of its product to be our returned output.



Eventually, ***NaiveBayes\_point(x, y, label, point)*** calculate , which compute the sum of product of previous two functions. Through finding maximized value, returning ***result*** which is the class that maximized value belonging to.



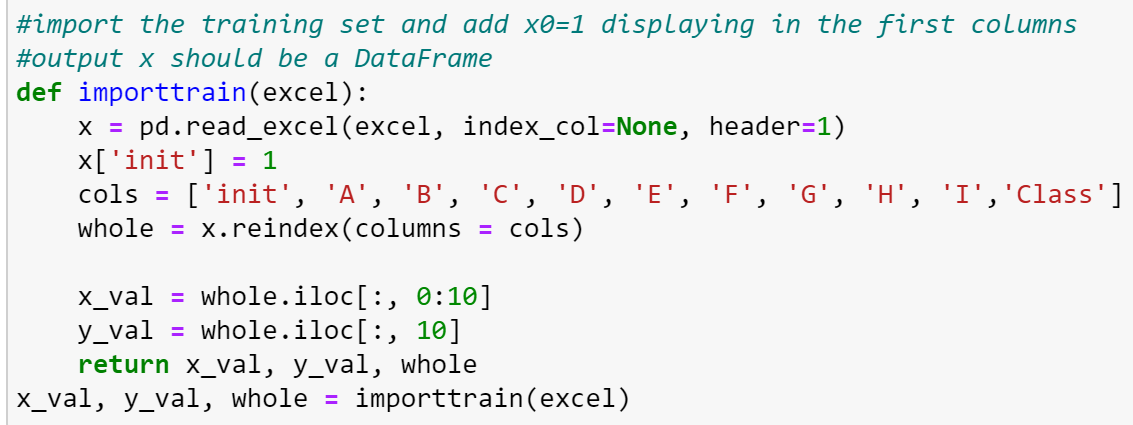
Then we traverse every tuple in test set to form final function ***NaiveBayes(x\_test, x\_train, y\_train)***.



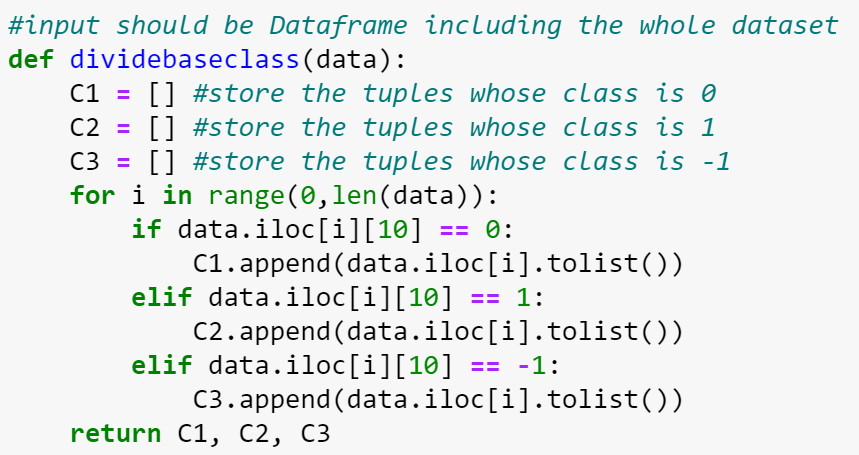
## *Perceptron model*

In the perceptron model, since we have three classes in training dataset, then we need to make three hyperplanes for the model. Therefore, our strategy is to split the training dataset at the proportion 2:1 and to list all the candidate elements into a new list. And figure out the predicted value based on their frequencies.

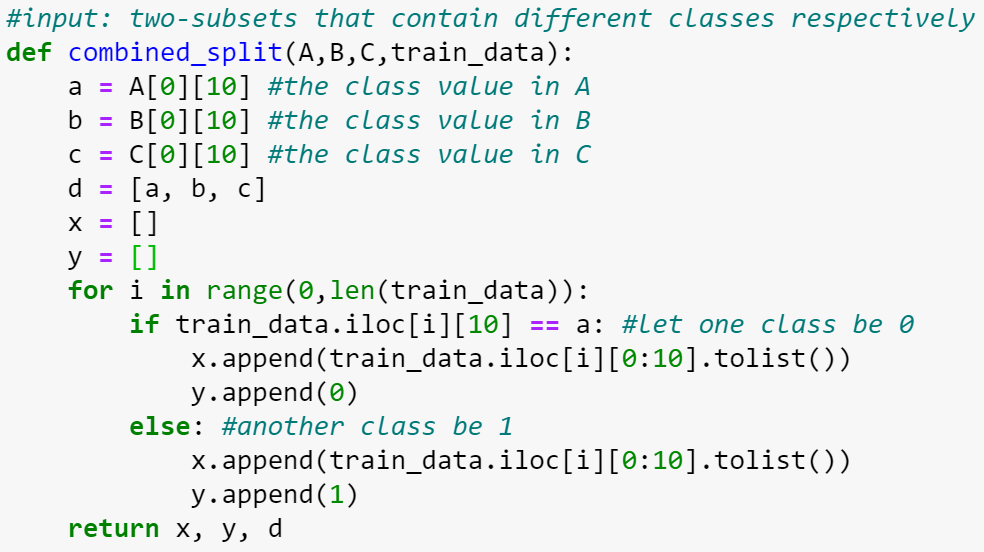
Firstly, we need to preprocess the data. When dataset is imported, initial weight is added into the dataframe, namely, add one more columns ***init*** that equal to 1.



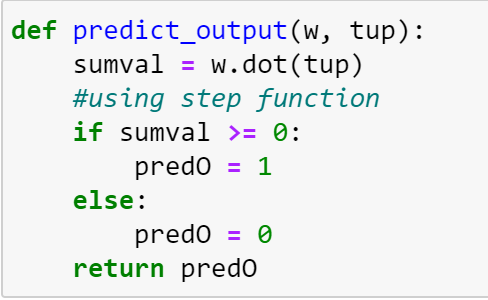
Then divide data based on different classes. In the function ***dividebaseclass(data)***, the input ***data*** should be DataFrame including the whole training dataset. Then finally output three lists that contain different classes.



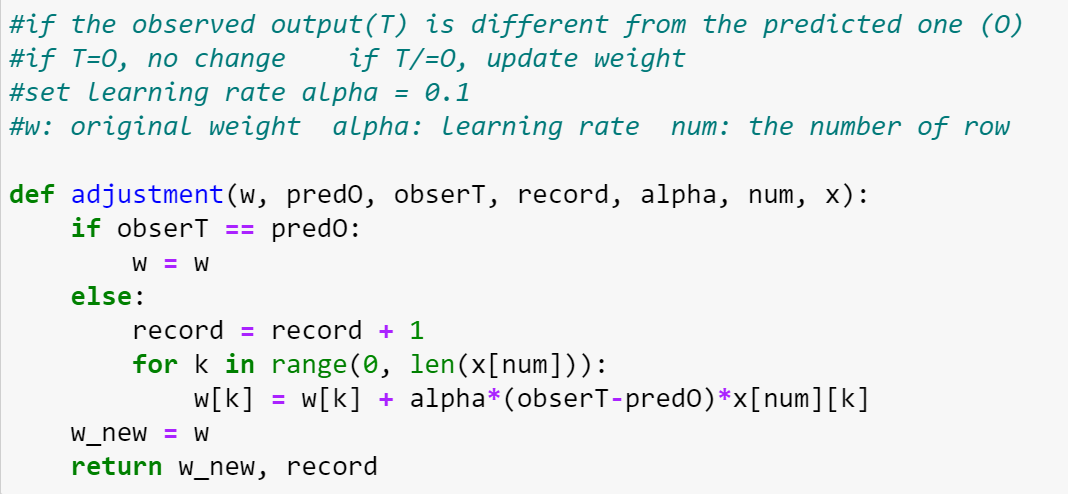
After that, we combine the above divided lists forming two new subsets. There are four parameters in function ***combined\_split(A, B, C, train\_data)***. The input ***A*** will be a single training dataset where its observe values become 0 while the input ***B, C*** will be combined together to be another dataset where observe values become 1.



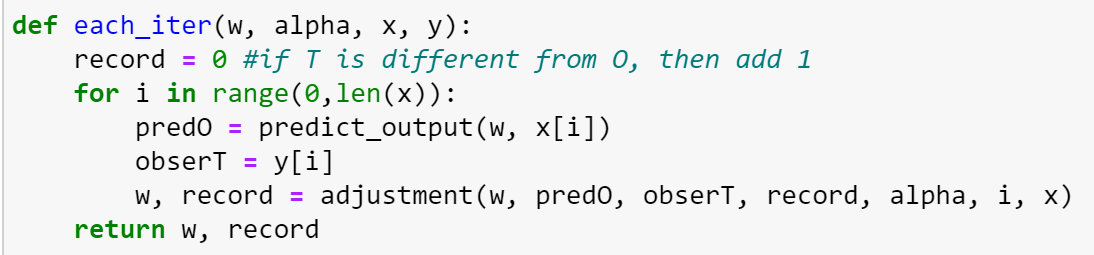
In the function ***predict\_output(w, tup)***, we calculate the dot product ***sumval*** of current weight and training tuple. By the step function, we determine the exact predict value ***predO*** for further comparison.



To adjust the weight, we have function ***adjustment(w, predO, obserT, record, alpha, num, x)***. The input ***predO*** and ***obserT*** stand for the predicted value and observed value respectively. The input ***num*** represent the index in the training dataset. If observed value is equal to predicted value, then there is no change on weight. However, if they contain different values, updating weight is needed. After this function, a new weight ***w\_new*** and ***record*** are the outputs.

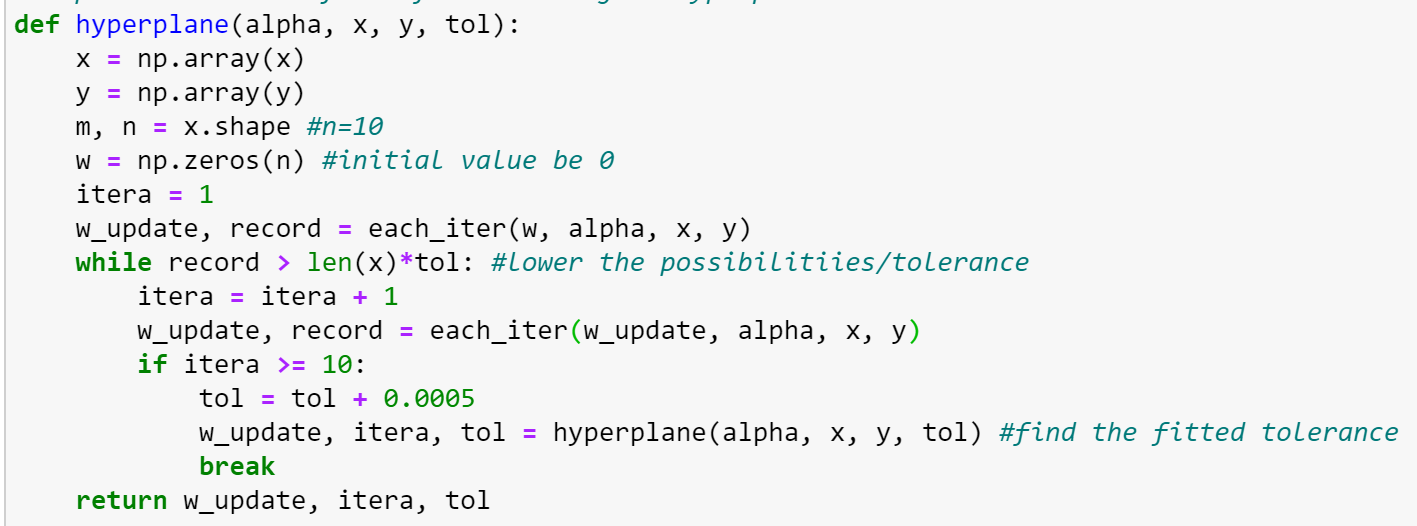


In the function ***each\_iter(w, alpha, x, y)***, we traverse every tuple in the training dataset and compare predicted value ***predO*** with the observed value ***obserT***. Then do adjustment in the function previously defined.

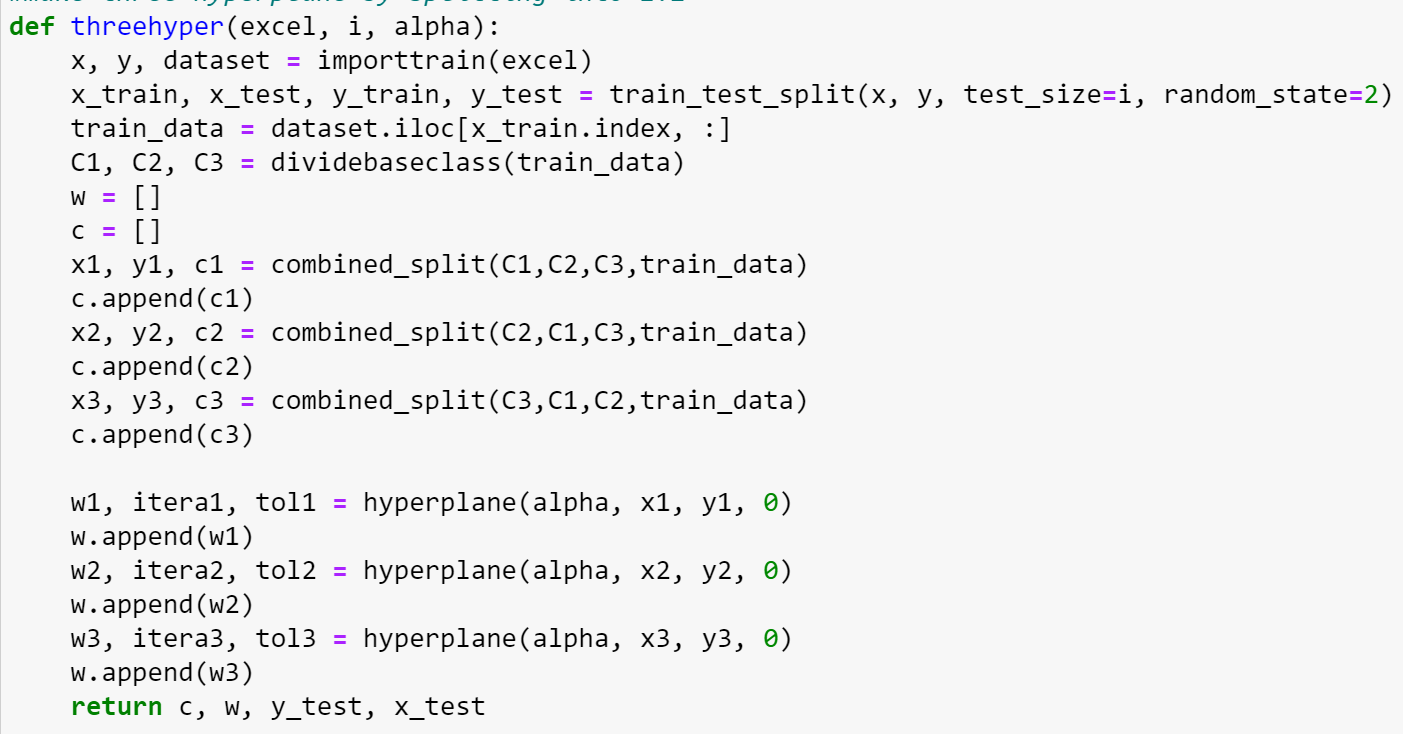


Finally, to find a specific hyperplane, we have a function ***hyperplane(alpha, x, y, tol)*** where the input ***alpha*** stands for learning rate. The input ***x*** and ***y*** are two lists which is the outcome of ***combine\_split(A, B, C, train\_data)***. Since it is hard to find a stable hyperplane that makes sure all data could be linearly separable, then we have to set some tolerances ***tol*** to lower the constraints for finding the hyperplane.

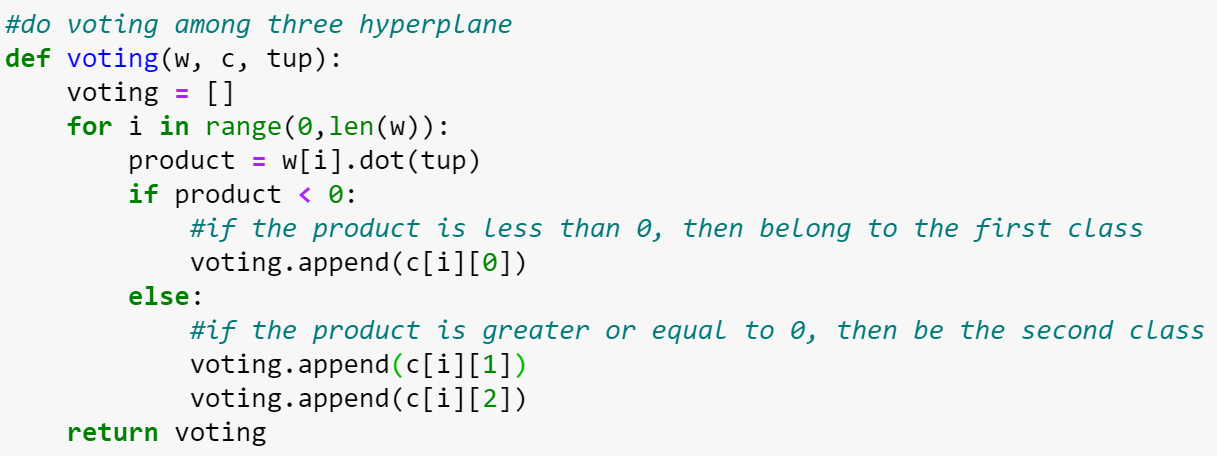
Initially, we need to initialize a weight ***w*** to be 0, which has the same length as each training tuple. After that we need to do iteration based on function ***each\_iter(w, alpha, x, y)*** to find the updated weight and to record the number ***record*** when observed and predicted are different. If the recorded number is greater than 0, then iteration are needed again and again until find a hyperplane that satisfy ***record = 0***. However, through running data for several times, it is hardly to find suitable hyperplane without any constraints. Therefore, we assume that if the iteration time is greater than 10, it is highly impossible for separating linearly, then we lower our constraints. At the speed of relaxing ***0.005*** tolerance ***(tol = tol+0.0005)*** every time, we do recursion to find the hyperplane until data have no chance for getting into while loop.



Combining the above function, we get a function ***threehyper(excel, i, alpha)*** to figure out three hyperplanes. The output ***c*** is a list containing a classes list at a splitting sequence. ***w*** is a list containing the final weight that each hyperplane get.

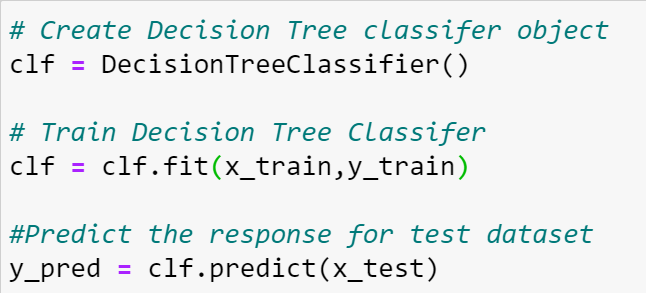


For each tuple in test set, we need to do voting with function **voting(w, c, tup)**. Since one tuple need to enter three different hyperplanes, the outcome from each hyperplane will probably various. Therefore, we store all the candidate outcomes into a new list ***voting***, then figure out the element that occurs frequently most.

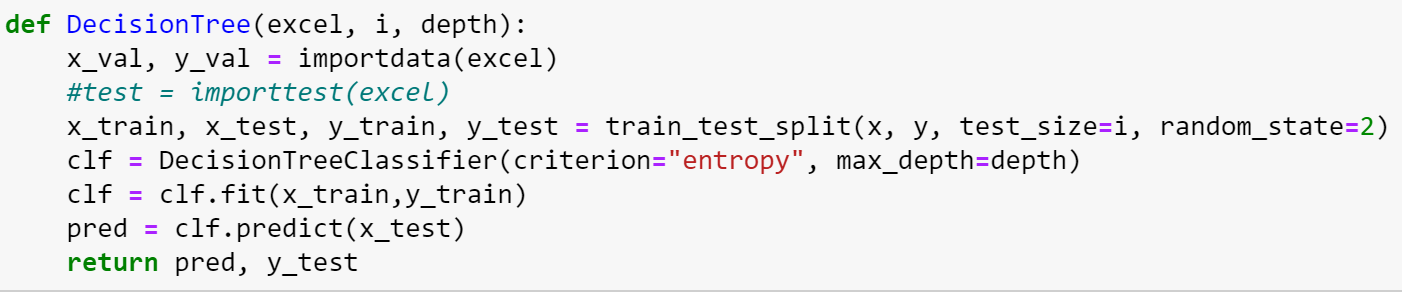


## *Decision Tree model*

Using library ***scikit-learn***, we directly use ***DecisionTreeClassifier()*** to train model.

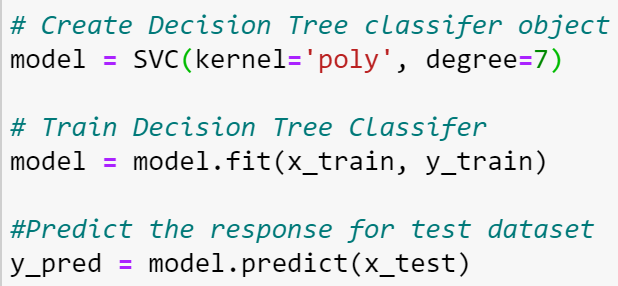


Then we combine all process into a function ***DecisionTree(excel, i, depth)***

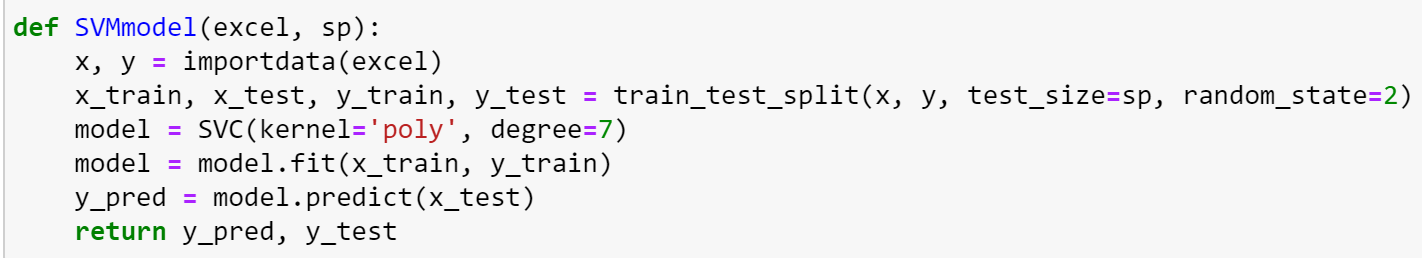


## *SVM model*

In SVM model, we use ***SVC()*** imported from ***sklearn.svm***. By testing the accuracy of the model for several times, when the kernel type is polynomial with degree of 7, SVM model can generate a good performance.

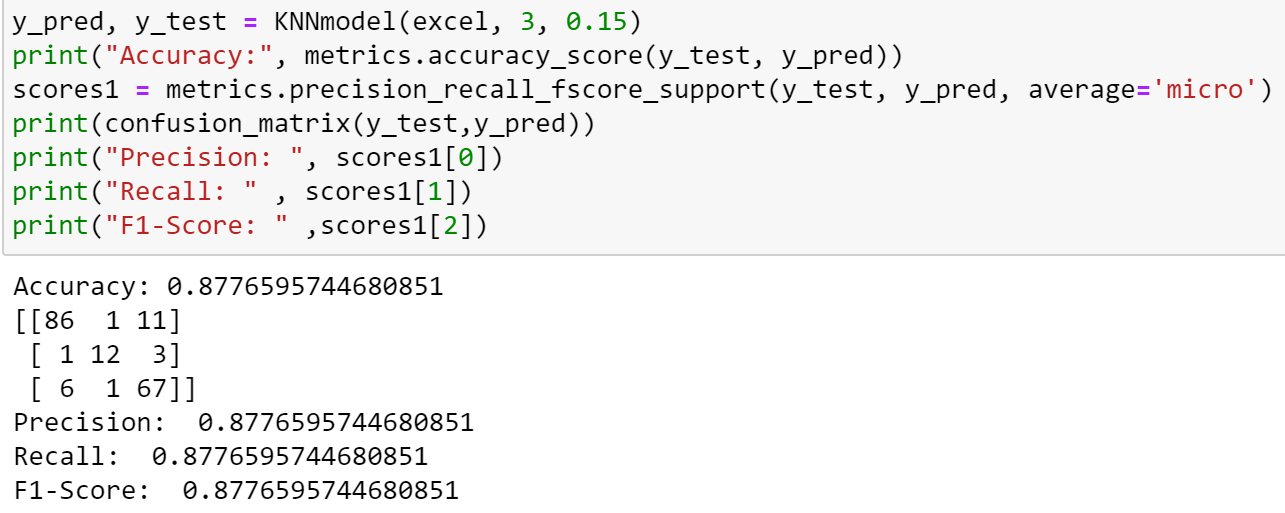


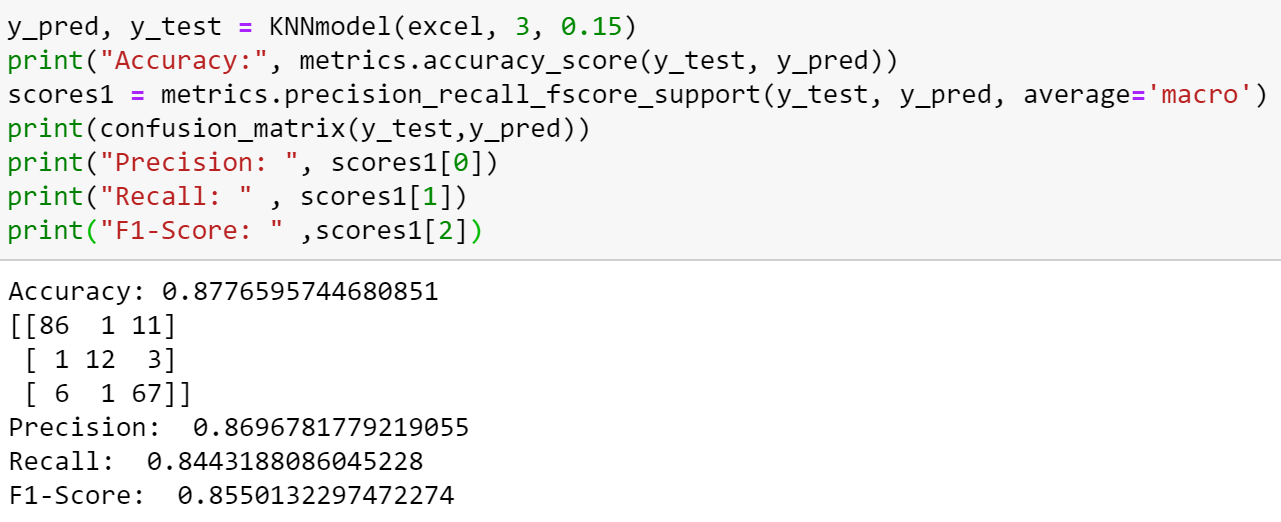
Then we combine all process into a function ***SVMmodel(excel, sp)*** where ***sp*** represents the splitting cut.



# **Experimental results of different models**

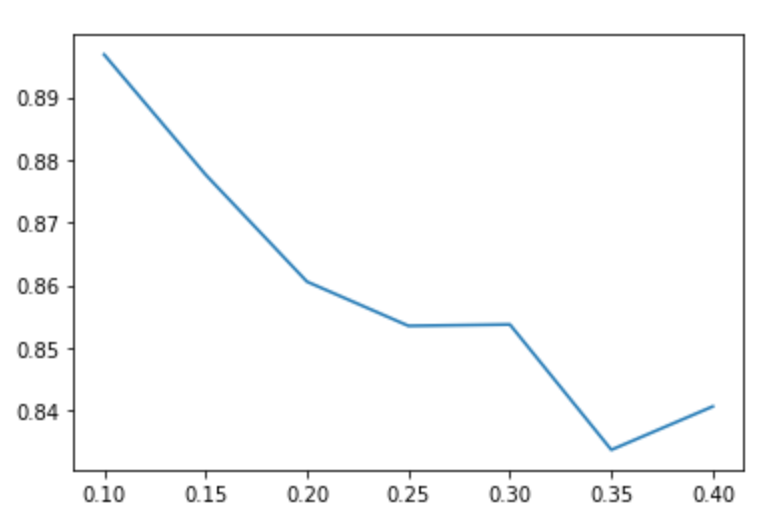
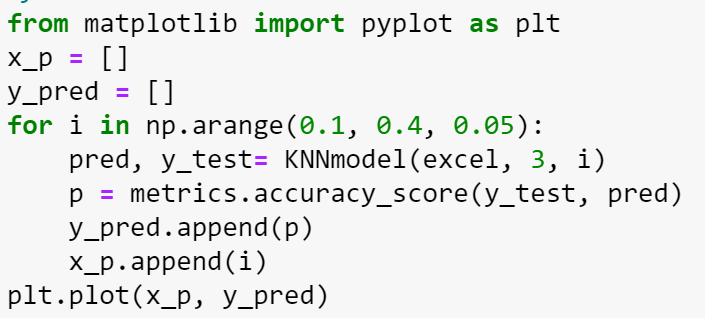
In this section, we had experiments on each model for testing their performance. Each model, we did a loop for testing the best splitting cut. Besides, we had to test the optimal k value for KNN model, learning rate for perceptron model and maximum depth for decision tree model. Additionally, to assessing how good or how accurate the classifier is at predicting the class label of tuples, precision (a measure of exactness), recall (a measure of completeness) and F-score (combined precision and recall) needed to be considered based on Macro-averaging and Micro averaging using ***metrics*** imported from ***scikit-learn*** library as following.



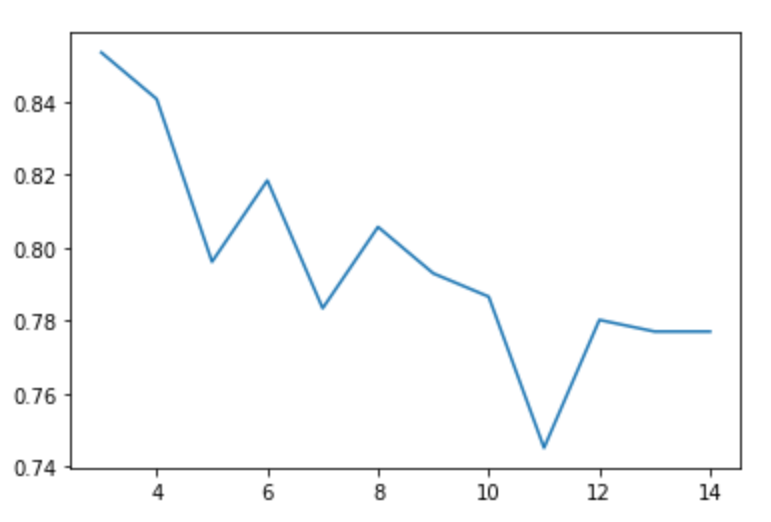
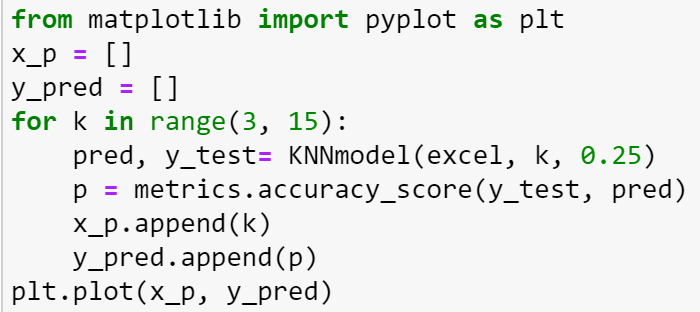


## *KNN model (K-Nearest-Neighbors Classification)*

To offer the best performance and make sure that the accuracy is well enough, we have to consider which splitting cut and k value are the best. We consider the splitting point in the first place. From the graph displayed, it is obvious that when splitting cut is below 0.15, the model has relatively good performance.



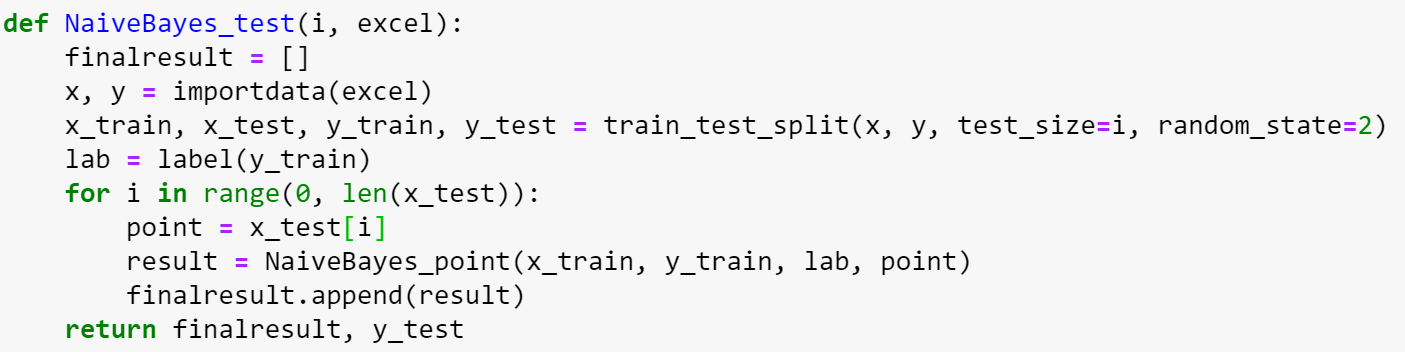
Secondly, we have to choose the optimal k value with the range. Apparently, the model has relatively good performance when



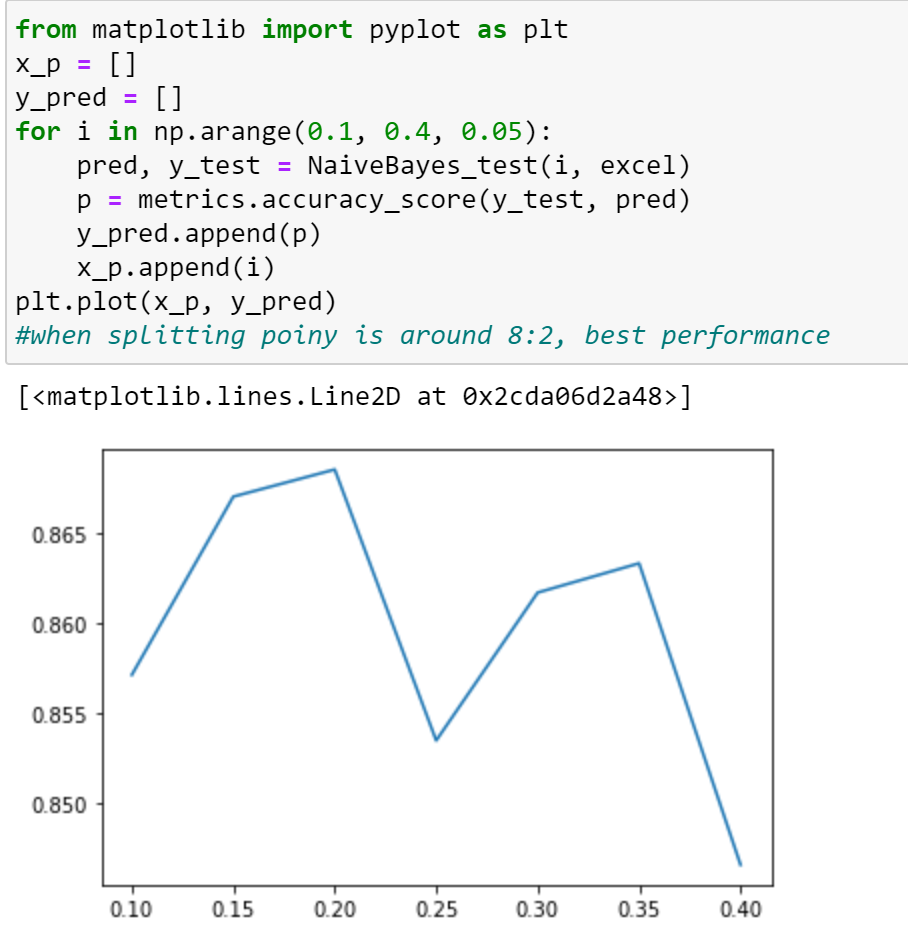
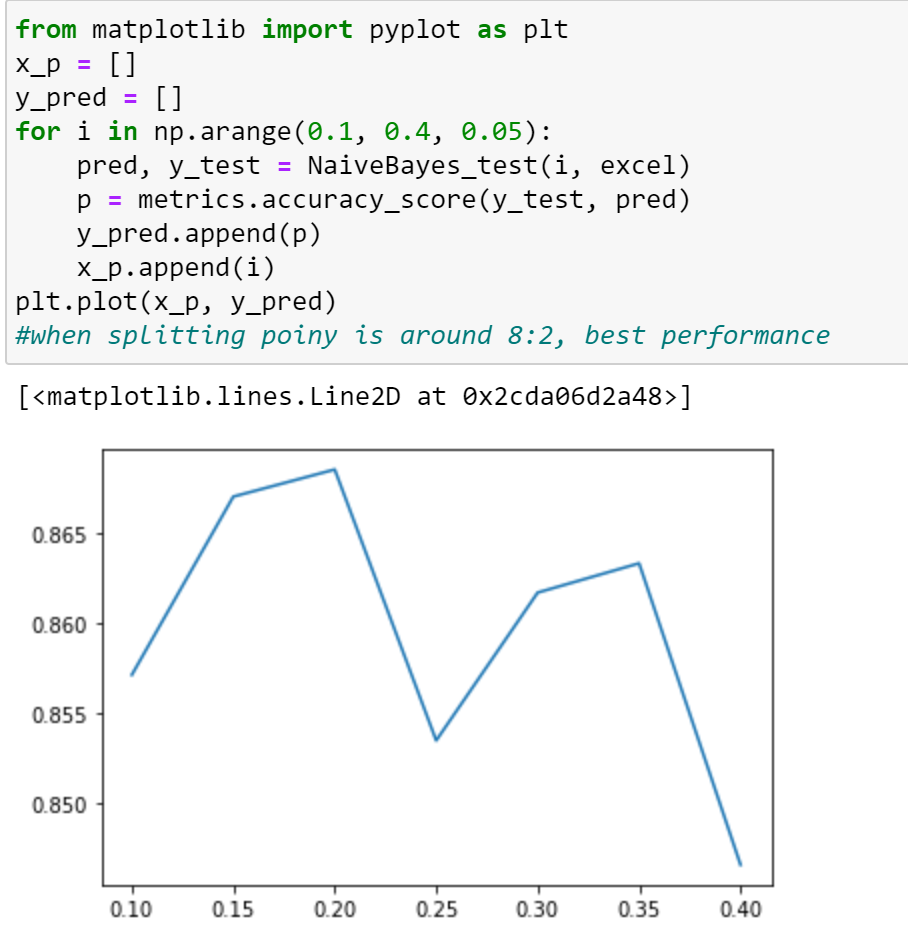
Therefore, we consider that when and , KNN model has the best performance.

## *Naïve Bayes model*

To provide a good performance and make model’s accuracy at the same time, we need to find the optimal splitting point for Naïve Bayes model. In the following function, we simply adjust the previous function forming ***NaiveBayes\_test(i,excel)*** where i is an input parameter for splitting the dataset. The output here ***finalresult*** and ***y\_test*** represent the predicted value and actual value respectively.



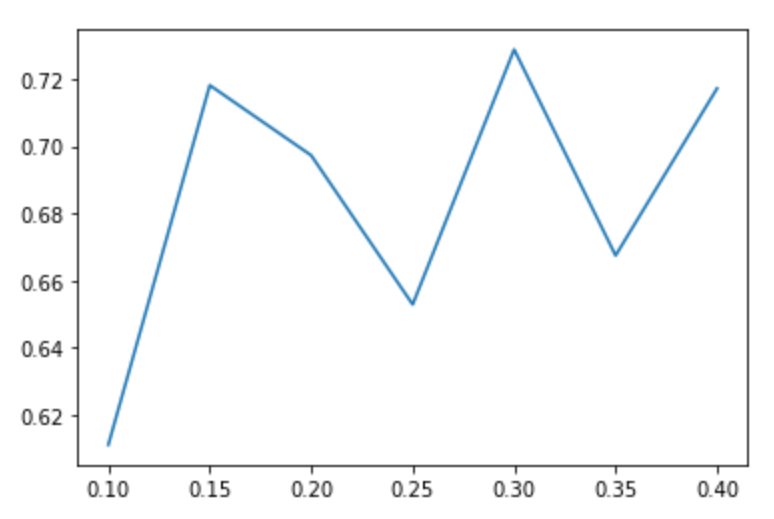
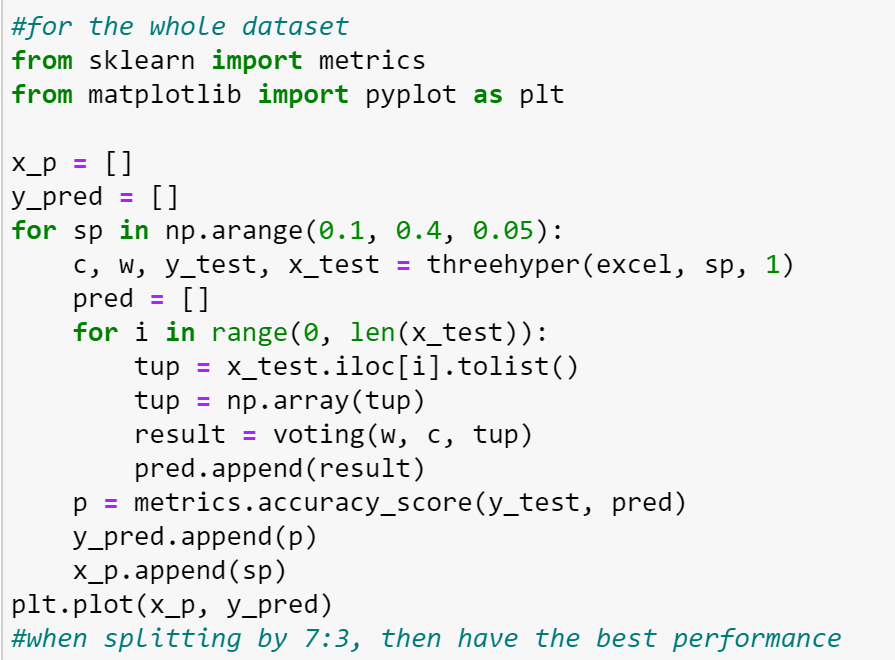
In the following, we do a looping within a range that is suitable for splitting. Then we calculate the accuracy for each time when splitting cut increase and plot a graph for visualization. Therefore, apparently when the dataset is split into around , the model can make the best performance.



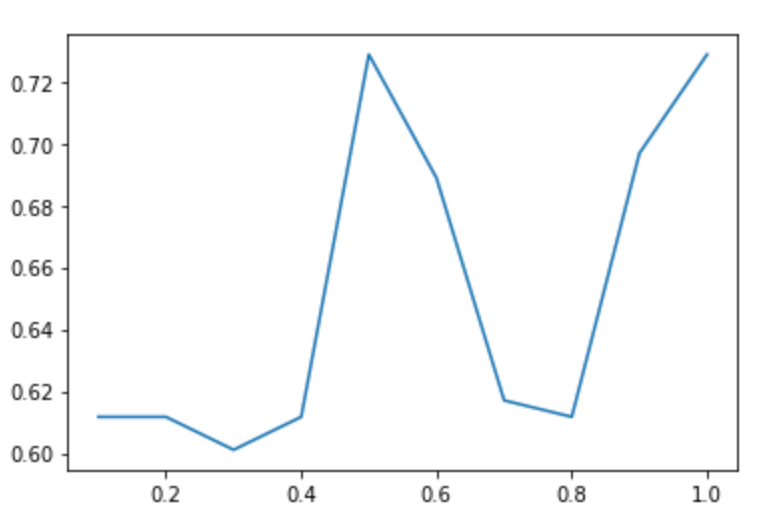
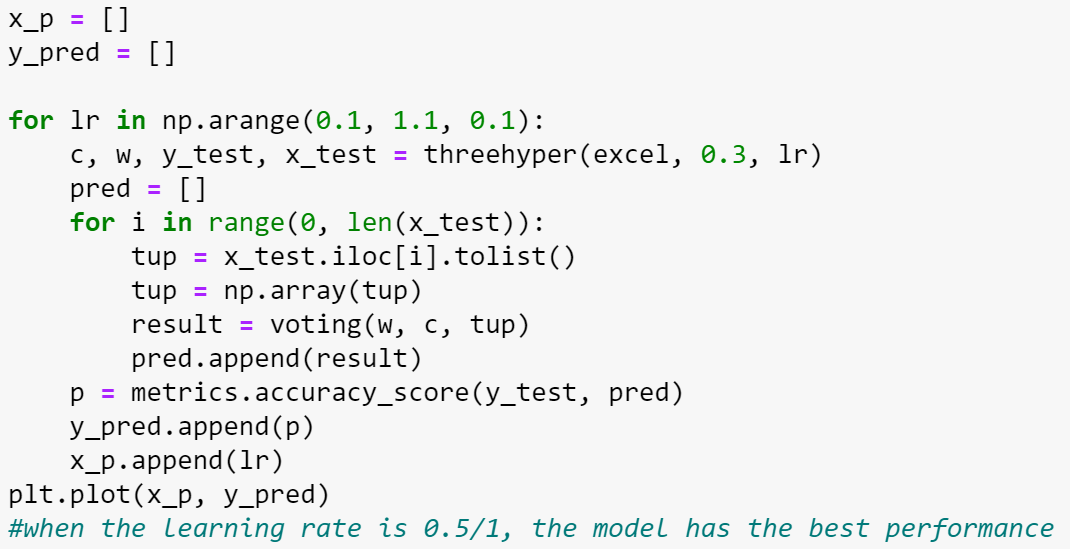
Therefore, we consider that when , Naïve Bayes model has the best performance.

## *Perceptron model*

To offer a good performance, we need to consider two variables which are learning rate and splitting cut. We do the similar operation with the previous model. From the graph, it plots the relationship between the splitting cut and accuracy. And when splitting cut is 0.3, the model have the best performance.



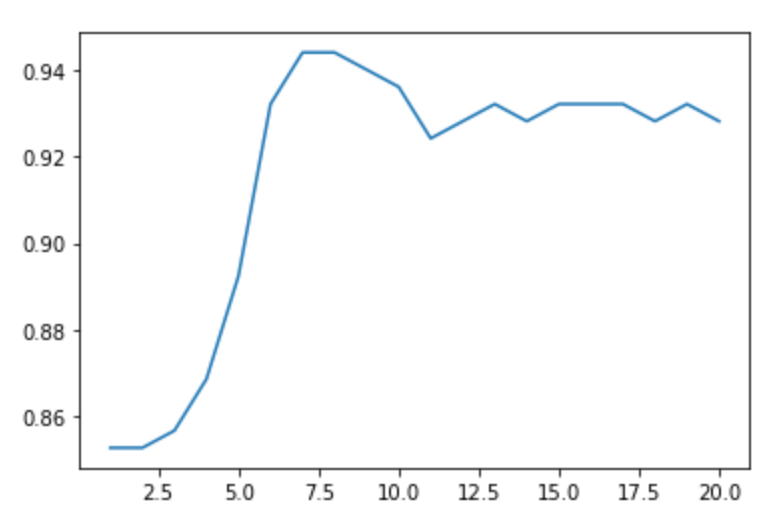
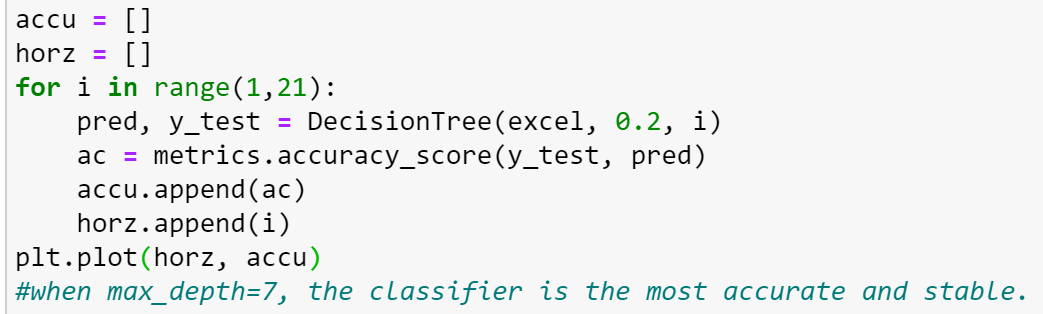
Then to figure out the optimal learning rate, we do similar operation with finding the optimal k value in KNN model. When learning rate is 0.5 or 1, the performance is the best.



Therefore, we consider that when and , perceptron model has the best performance.

## *Decision Tree model*

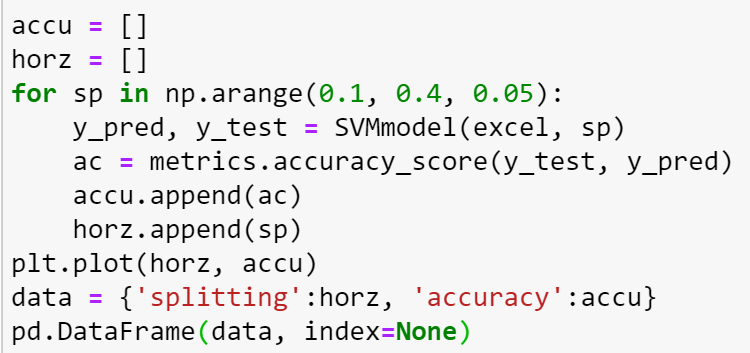
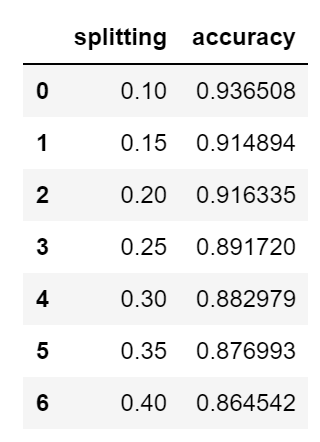
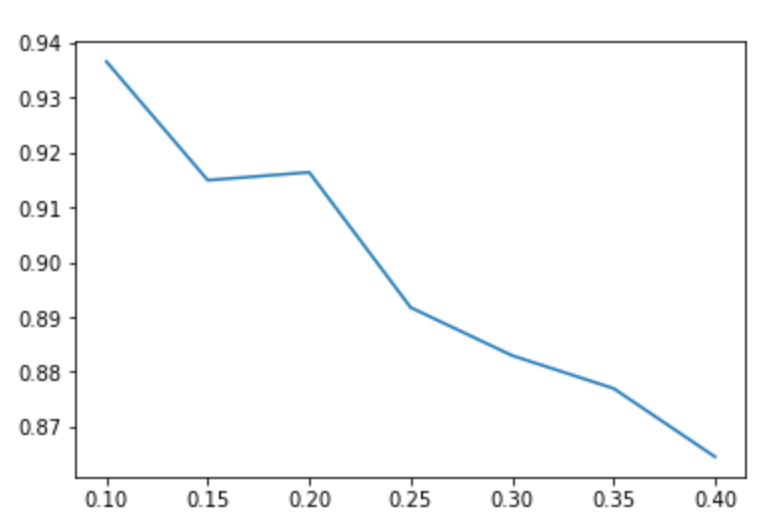
In decision model, ***max\_depth*** represent the maximum depth of tree, which can be used as a control variable for pre-pruning. Therefore it is necessary to figure out the best tree depth to ensure a good performance. From the plot below, it seems that when maximum depth is 7, the model has relatively good performance.



Therefore, we consider that when , decision tree model has the best performance.

## *SVM model*

Similarly, by doing a looping for the splitting cut, it is shown that when splitting cut is , the model is relatively good.

Therefore, we consider that when , SVM model has the best performance.

Overall, by using micro-averaging (shown in **Table 1**) and macro-averaging (shown in **Table 2**) respectively, we list a table as below for further analysis.

***Table 1***

*Accuracy using micro-averaging*

|  |  |  |  |
| --- | --- | --- | --- |
| *Model* | *Precision* | *Recall* | *F-Score* |
| KNN | 0.878 | 0.878 | 0.878 |
| Naïve Bayes | 0.868 | 0.868 | 0.868 |
| Perceptron | 0.729 | 0.729 | 0.729 |
| Decision Tree | 0.944 | 0.944 | 0.944 |
| SVM | 0.916 | 0.916 | 0.916 |

***Table 2***

*Accuracy using macro-averaging*

|  |  |  |  |
| --- | --- | --- | --- |
| *Model* | *Precision* | *Recall* | *F-Score* |
| KNN | 0.869 | 0.844 | 0.855 |
| Naïve Bayes | 0.692 | 0.641 | 0.631 |
| Perceptron | 0.610 | 0.631 | 0.610 |
| Decision Tree | 0.913 | 0.943 | 0.927 |
| SVM | 0.896 | 0.829 | 0.854 |

# Result analysis

From the above analysis, all models can be used with the accuracy greater than 70% after choosing the optimal control variables. Among these model, the best one is decision tree model, which has 94% accuracy. SVM models are also relatively good. Unfortunately, KNN, Naïve Bayes and perceptron model are not good enough. Here are several potential reason for explanation.

First, since there are three classes in dataset while when we train the model, we have to spilt classes into two categories, therefore in the above perceptron model, three hyperplane that we set are always biased. In order to become two categories, data with two different classes are always combined together, which leads to data from two sides seriously unbalance resulting in the predicted error.

Second, when train the perceptron model, it is difficult to train a hyperplane that is linearly separable. Therefore, tolerance are made for this model, which means the possibility of accepting error is greater.

Third, Although KNN and Naïve Bayes have relatively good accuracy, their precision and recall are not high enough. Recall value represents the ratio that successful predicted among actual data while precision value stands for the ratio that those actual have among all predicted data. Higher recall and precision the model have, better performance they possess. Obviously, these three models cannot guarantee from this perspective.