Individual Project Report

Abstract (workflow):

For analyzing and predicting people's acceptance of cars, this report has taken the following steps for collecting various factors (data):

Data processing:

1. **Preprocessing of data and visualization of basic information, observing the general distribution of the data.**

Build model:

2. **For the construction of 3 models of Naive Bayes, KNN and Perceptron without any For Naive Bayes, KNN and Perceptron, 3 models are built without existing machine learning or integrated learning libraries.**

Improve model:

3. **Improve KNN and Perceptron model, and check if there is a higher score/accuracy.**
4. **According to the knowledge in the lecture, implement a decision tree to predict the result without open-source libraries. In addition, complete the result generated by decision tree in $skelen$ library, find out the shortcomings of own implementation one.**
5. **Derivation of perceptron back propagation and use it to improve existing perceptron without open-source libraries.**

Integrated model:

6. **Call existing machine learning integrated models $catboost$.**

Analysis of the effects (scoring, accuracy, etc.) of each model

7. **Analyze the result produced by each model, analyze the advantages of the model with the highest results and point out the shortcomings of models with relatively poor results.**

Data Processing

Firstly, import the package that invokes the associated data processing.

In [3]:
```
import numpy as np
import pandas as pd
```

Then we can import the test data and training data as $pd.dataframe$ and check.

In [10]:
```
filename="./dataset/test.csv"

# import the test data
from pandas import read_csv
f=open(filename,encoding='UTF-8')
test=read_csv(f)
print(test)
```

```
    buying  maint  doors persons lug_boot safety
0      low  vhigh      3    more      med    med
1    vhigh   high      2    more    small   high
2    vhigh    med  5more       4      med   high
3      med    med  5more       4    small    med
4     high    med      4       4    small   high
..     ...    ...    ...     ...      ...    ...
474    low  vhigh      4       4      big   high
475    med  vhigh      3    more      big   high
476  vhigh    low      3       4      big   high
477    med  vhigh      2       2      med    med
478    med  vhigh      2       4      med    low

[479 rows x 6 columns]
```

In [11]:
```
filename="./dataset/training.csv"
f=open(filename,encoding='UTF-8')
training=read_csv(f)
print(training)
```

```
     buying  maint  doors persons lug_boot safety evaluation
0      high    low  5more       2      med    med      unacc
1       med    low  5more       4    small    low      unacc
2       med    med      4    more      med    low      unacc
3       low    med      3    more      med    low      unacc
4       low  vhigh      3    more      big    low      unacc
...     ...    ...    ...     ...      ...    ...        ...
1110    med  vhigh  5more    more    small    low      unacc
1111  vhigh   high      4       4      big    med      unacc
1112  vhigh   high      3    more      med    low      unacc
1113    med    med      4       2      big    med      unacc
1114    low    low  5more       4    small    low      unacc

[1115 rows x 7 columns]
```

Check whether there are some data is missing or empty.

In [6]:
```
# check whether the test data has NaN number
test.isnull().any()
```

Out[6]:
```
buying      False
maint       False
doors       False
persons     False
lug_boot    False
safety      False
dtype: bool
```

In [10]:
```
# check whether the test data has NaN number
training.isnull().any()
```

Out[10]:
```
buying        False
maint         False
doors         False
persons       False
lug_boot      False
safety        False
evaluation    False
dtype: bool
```

In some models, we need to transform the classified variable to the one-hot code to improve the speed at which the code runs or the reliability of the results.

```
train_input = pd.get_dummies(train_attr).values
```

Quantify and normalize data:

```
# methond 2: All labels are digitized and forward indexed 正向化
# solve categorical variable
train_tep = training.copy()
train_tep.buying.replace(('vhigh','high','med','low'),(1,2,3,4), inplace=True)
train_tep.maint.replace(('vhigh','high','med','low'),(1,2,3,4), inplace=True)
train_tep.doors.replace(('2','3','4','5more'),(1,2,3,4), inplace=True)
train_tep.persons.replace(('2','4','more'),(1,2,3), inplace=True)
train_tep.lug_boot.replace(('small','med','big'),(1,2,3), inplace=True)
train_tep.safety.replace(('low','med','high'),(1,2,3), inplace=True)
train_tep.evaluation.replace(('unacc','acc'),(1,2), inplace=True)

test_tmp = test.copy()
test_tmp.buying.replace(('vhigh','high','med','low'),(1,2,3,4), inplace=True)
test_tmp.maint.replace(('vhigh','high','med','low'),(1,2,3,4), inplace=True)
test_tmp.doors.replace(('2','3','4','5more'),(1,2,3,4), inplace=True)
test_tmp.persons.replace(('2','4','more'),(1,2,3), inplace=True)
test_tmp.lug_boot.replace(('small','med','big'),(1,2,3), inplace=True)
test_tmp.safety.replace(('low','med','high'),(1,2,3), inplace=True)

train_tep, test_tmp
```

Model establishment and solution

1. Naïve Bayes:

Algorithm: Initial and set up a list query mechanism. Based on the training set data calculate the prior probabilities and posterior probabilities of different cases, then store in the nested list. For each tuple of test, find out the case in the nested list. Then the result of the probabilities of each attribute is multiplied by two types of the prior probability, and the predicted result is the maximum of the two.

It is worth noting that: Because the number of samples in the training set is not very large, there may be situations that have not appeared in the training set in the test set, for example, it not exists a case that people_2 is acc in the training set, but there is a case in the test set. So, when it needs to calculate the probability, it should add the Laplacian operator.

Result:

In [66]:
```
def score():
    train_res = pd.DataFrame([bayes_res_train['pred'], training['evaluation']]).T
    train_res.columns = ['pred', 'actl']
    train_pred_unacc = train_res[train_res['pred']=='unacc']
    train_FP = train_pred_unacc[train_pred_unacc['actl']=='acc'].shape[0]
    train_TP = train_pred_unacc[train_pred_unacc['actl']=='unacc'].shape[0]
    # print('train_FP' + ': ' + str(train_FP), 'train_TP' + ': ' + str(train_TP))

    train_pred_acc = train_res[train_res['pred']=='acc']
    train_FN = train_pred_acc[train_pred_acc['actl']=='acc'].shape[0]
    train_TN = train_pred_acc[train_pred_acc['actl']=='unacc'].shape[0]
    # print('train_FN' + ': ' + str(train_FN), 'train_TN' + ': ' + str(train_TN))
    precision = train_TP / (train_TP+train_FP)
    recall = train_TP / (train_TP+train_FN)
    f_score = 2*precision*recall/(precision+recall)
    print("unacc is positive and acc is negative")
    print('precision: ' + str(precision) + ', recall: ' + str(recall) + ', f_score: ' + str(f_score))
    print("==========================================")

    train_pred_unacc = train_res[train_res['pred']=='acc']
    train_FP = train_pred_unacc[train_pred_unacc['actl']=='unacc'].shape[0]
    train_TP = train_pred_unacc[train_pred_unacc['actl']=='acc'].shape[0]
    # print('train_FP' + ': ' + str(train_FP), 'train_TP' + ': ' + str(train_TP))

    train_pred_acc = train_res[train_res['pred']=='unacc']
    train_FN = train_pred_acc[train_pred_acc['actl']=='acc'].shape[0]
    train_TN = train_pred_acc[train_pred_acc['actl']=='unacc'].shape[0]
    # print('train_FN' + ': ' + str(train_FN), 'train_TN' + ': ' + str(train_TN))
    precision = train_TP / (train_TP+train_FP)
    recall = train_TP / (train_TP+train_FN)
    f_score = 2*precision*recall/(precision+recall)
    print("acc is positive and unacc is negative")
    print('precision: ' + str(precision) + ', recall: ' + str(recall) + ', f_score: ' + str(f_score))
score()
```

```
unacc is positive and acc is negative
precision: 0.9163841807909604, recall: 0.8061630218687873, f_score: 0.8577472236911686
==========================================
acc is positive and unacc is negative
precision: 0.8478260869565217, recall: 0.724907063197026, f_score: 0.7815631262525051
```

Unacc as positive and acc as negative:
$precision$: 0.9163841807909604
$recall$: 0.8061630218687873
$f1\_score$: 0.857747223691168

acc as positive and unacc as negative:
$precision$: 0.8478260869565217
$recall$: 0.724907063197026
f_score: 0.7815631262525051

2. KNN:

Algorithm: Assume that the selected distance (k) is 3 by default. Pass the test set and training set into the knn function at the same time, use two for loop statements to compare how many elements of each tuple of the test set are inconsistent with each tuple of the training set. The result of the comparison is the distance between the two. Sort the results of the same row of the test set and take the first k evaluation results, the more one is used as the prediction of the test data.

Shortcoming: Different from other algorithms, knn model needs to recalculate the results of each tuple of the test set and each tuple of the training set. So, it may take more time cost than others. When you execute the code, you may wait a few minutes to see the result.
In addition, the choice of k value will have a great influence on the results of the model.

Improvement: Using cross-checking, another range of k values is taken and traversed, and the k that best fits this data set is selected (the effect is the best), The use visualization tools to show the effect of k on the final score.
Result:

```
# score with acc is negative and unacc is positive
TP = train_res_des['unacc']['unacc']
FN = train_res_des['acc']['unacc']
FP = train_res_des['unacc']['acc']
TN =  train_res_des['acc']['acc']
precision = TP/(TP+FP)
recall = TP/(TP+FN)
f_score = 2*pre*recall/(pre+recall)
accuracy = train_res[train_res['pred']==train_res['actl']].shape[0]/train_res.shape[0]
print("acc is positive and unacc is negative")
print('precision: ' + str(precision) + ', recall: ' + str(recall) + ', f_score: ' + str(f_score))

# acc is positive and unacc is negative
TP = train_res_des['acc']['acc']
FN = train_res_des['unacc']['acc']
FP = train_res_des['acc']['unacc']
TN =  train_res_des['unacc']['unacc']
precision = TP/(TP+FP)
recall = TP/(TP+FN)
f_score = 2*pre*recall/(pre+recall)
accuracy = train_res[train_res['pred']==train_res['actl']].shape[0]/train_res.shape[0]
print("acc is positive and unacc is negative")
print('precision: ' + str(precision) + ', recall: ' + str(recall) + ', f_score: ' + str(f_score))
```

```
unacc    862
acc      253
dtype: int64
0.7730941704035874 0.22690582959641256
==========================================
acc is positive and unacc is negative
precision: 0.9593967517401392, recall: 0.9775413711583925, f_score: 0.950493007035941
acc is positive and unacc is negative
precision: 0.924901185770751, recall: 0.8698884758364313, f_score: 0.896551724137931
```

unacc as positive and acc as negative:

$precision$: 0.9593967517401392

$recall$: 0.9775413711583925

f_score: 0.950493007035941

acc as positive and unacc as negative:

$precision$: 0.924901185770751

$recall$: 0.8698884758364313

f_score: 0.896551724137931

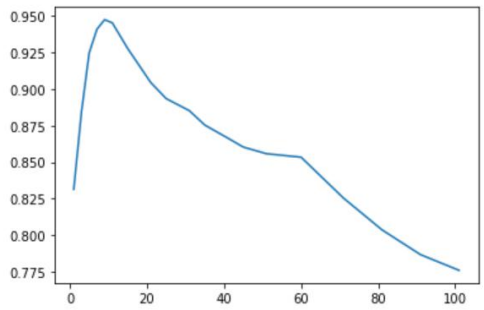
Cross-checking & K value choice:

After improvement, the score can reach 0.9474291140957808 and we can plot the graph that can show the effect of result k values can help us to choose the best k in this model for dataset. Of course, the parameter of k- fold crossing will also effect the score, so we just take the max one to fit the data.

```
In [112]: max_score = []
          # print(scores_2d)
          for i in range(len(scores_2d)):
              max_score.append(max(scores_2d[i]))
          plt.plot(kn,max_score)
          print(kn[np.argmax(np.array(scores))])
          max(max_score)

          9

Out[112]: 0.9474291140957808
```



Perceptron:

Algorithm:

a. The activation function is the step function: $step(x) = \begin{cases} 0 & if\ x < 0 \\ 1 & if\ x \geq 0 \end{cases}$

b. The bias is 1 by default, the learning rate defaults to one.

c. And the weight are (0,0,0) at the beginning and every time back to a new one.

d. $weight_{new} = weight_{old} + (rate) * (Actual - Predict) * Feature_{data}$

e. $Predict = sum(weight(vector)) * each\ attribute) - bias)$,

e. The lasting weight is used to predicted the result.

Shortcoming:

a. The convergence of the activation function is very poor and sometimes it even cannot converge.

b. Both the learning rate and the error will affect the accuracy and convergence of the model results, the default value should be adjusted.

c. There is only one activation function and only one hidden layer of the perceptron, which may not be able to accurately fit the data.

Improvement:

a. Change the activation function to sigmoid function which can converge better.

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

b. Introduce the loss function for back propagation, and redistribute the contribution rate of the function.

c. Use loops to select the best learning rate, learning period, judgment boundary and error. (Adjustment parameters)

Result:

```
In [38]: neuro1.append(neuro(train_input, train_input, train_target, 1, 1))

         unacc is positive and acc is negative
         precision: 0.7876712328767124, recall: 0.8550185873605948, f_score: 0.8199643493761141
         acc is positive and unacc is negative
         precision: 0.9526123936816525, recall: 0.9267139479905437, f_score: 0.939484721390054
```

unacc as positive and acc as negative:

$precision$: 0.9526123936816525

$recall$: 0.9267139479905437

f_score: 0.939484721390054

acc as positive and unacc as negative:

$precision$: 0.7876712328767124

$recall$: 0.8550185873605948

f_score: 0.8199643493761141

After improvement:

The score of the case of the acc as positive and unacc as negative attach 0.8628250892401836. And it just takes the best learning rate and the bias. In addition, the defined boundary threshold to decide what its category is very important in this case, here just take the final result divided by the possibility of the $unacc$ and acc. You can sort them and take the $prossibility * count$ as the ordered, select this value as your boundary.

```
a = np.arange(0.01, 1, 0.3)
neuro1 = []
for i in a:
    for j in np.arange(0.1, 5, 0.1):
        neuro1.append(neuro(train_input, test, train_target, j, 1, i))
x = np.array(neuro1).max()
x
```

```
0.8572944297082228
0.8535655960805661
0.8560727661851257
0.8517906336088154
0.8559185859667917
0.8557640750670241
0.8556092324208266
0.8554540569586244
0.8535655960805663
0.8563801388147356
0.8509695290858724
0.8534059945504087
0.8589407446250655
0.8565333333333334
0.8556092324208266
0.8503060656649972
0.8498045784477946
```

Out[114]: 0.8628250892401836

Decision tree

Perceptron:

Algorithm:

a. Build a tree structure, where the aspect records the connection between the parent node and the child node. Each node has its own identification and stored data.

b. Every time a new element is explored, the id is added for query.

c. Use the Gini coefficient to find the most confusing, that means select the most capable of classifying attributes Preferentially.

d. Use recursion to find the deepest node until all attributes are added to the tree structure.

e. Pruning operation: Cutting out the child nodes of the attribute with too large Gini coefficient means that it is difficult to separate the results. Therefore, a threshold must be established to judge.

Shortcoming:

a. For each branch, the Gini coefficient has a different degree of influence. So, you can't prune branches blindly.

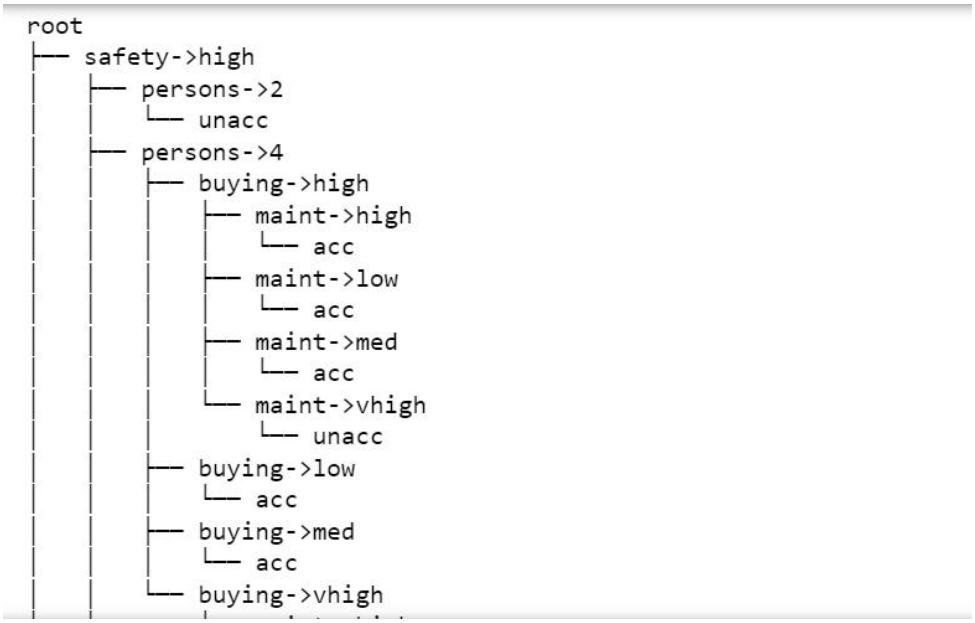
b. It is easy to overfit, the branches may be too detailed.

c. Difficult to adjust parameters.

It can be used random forest model to improve: Generate decision trees for many situations, and assign the contribution rate to each decision tree until the decision tree is found.

Result:

The part of the decision tree graph part:




```
In [217]: print(Score_unacc(pred_list,test_label))

{'accuracy': 0.9032258064516129, 'precise': 0.9945945945945946, 'recall': 0.8761904761904762, 'f_measure': 0.9316455696202532}
```

```
In [218]: # acc is positive acse
def Score_acc(pred_result, true_value):
    mistakes = 0
    acc_count = 0
    for w in range(len(pred_result)):
        if pred_result[w] != true_value[w]:
            mistakes += 1
        elif pred_result[w] == 'acc':
            acc_count += 1
    accuracy = 1 - (mistakes / len(true_value))
    precise = acc_count / pred_result.count('acc')
    recall = acc_count / true_value.count('acc')
    f_measure = 2 * precise * recall / (precise + recall)
    score_dic = {'accuracy':accuracy, 'precise': precise, 'recall': recall, 'f_measure': f_measure}
    return score_dic
print(Score_acc(pred_list, test_label))

{'accuracy': 0.9032258064516129, 'precise': 0.723404255319149, 'recall': 0.9855072463768116, 'f_measure': 0.834355828220859}
```

unacc as positive and acc as negative:

$$precision: 0.9945945945945946$$
$$recall: 0.8761904761904762$$
$$f\_score: 0.9316455696202532$$

acc as positive and *un*acc as negative:

$$precision: 0.723404255319149$$
$$recall: 0.9855072463768116$$
$$f\_score: 0.834355828220859$$

After improvement:

The part of the graph it plots likes this and you can find that the Gini generated by the ensemble learning package are all less than 0.5, and you can define its depth, the depth is no the bigger the better. As the depth increased, he divided the tree into more detail and maintained very high accuracy.

```
acc is positive and unacc is negative
precision: 0.9905882352941177, recall: 0.9952718676122931, f_score: 0.9929245283018868
acc is positive and unacc is negative
precision: 0.9849056603773585, recall: 0.9702602230483272, f_score: 0.9775280898876405
```

unacc as positive and acc as negative:

$\boldsymbol{precision}$: $\mathbf{0.9905882352941177}$

$\boldsymbol{recall}$: $\mathbf{0.9952718676122931}$

$\boldsymbol{f\_score}$: $\mathbf{0.9929245283018868}$

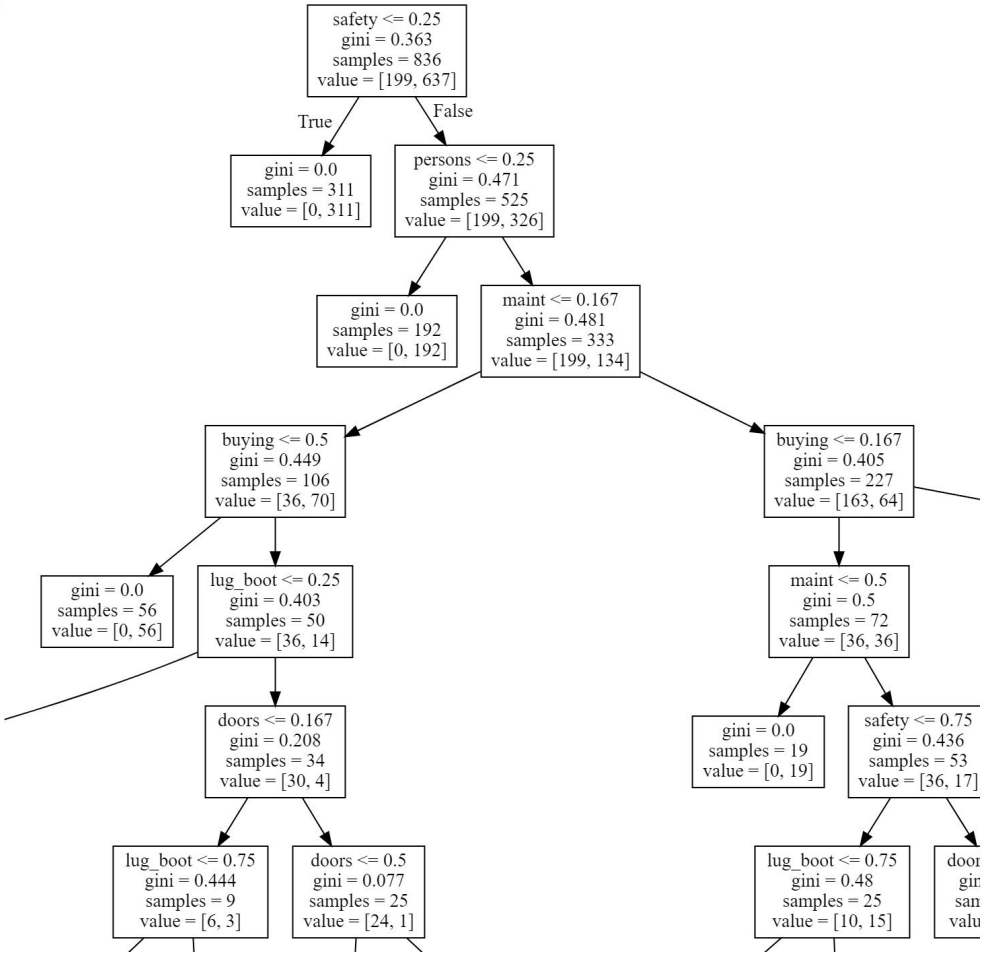
acc as positive and unacc as negative:

$\boldsymbol{precision}$: $\mathbf{0.9849056603773585}$

$\boldsymbol{recall}$: $\mathbf{0.9702602230483272}$

$\boldsymbol{f\_score}$: $\mathbf{0.9775280898876405}$

Out[50]:



Integrated learning：(The best one):

$\boldsymbol{catboost}$ **uses combined category features, which can use the relationship between features. In order to select the best tree structure, the algorithm enumerates different divisions, uses these divisions to build a tree, calculates the values in the obtained leaf nodes, and then the**

obtained tree calculates the score, and finally selects the best segmentation. And it reduces the need for tuning many hyperparameters and reduces the chance of overfitting.

```
acc is positive and unacc is negative
precision: 0.990521327014218, recall: 0.9881796690307328, f_score: 0.9893491124260355
acc is positive and unacc is negative
precision: 0.9630996309963099, recall: 0.9702602230483272, f_score: 0.9666666666666667
```

Through the loss function, it can be seen that its convergence speed is very fast. It only takes less than 20 times of learning and training to reduce the error to less than 0.2.

Result Analysis

The best one is *catboost* which is an integrated learning model, not only does it converge quickly, but the custom loss function allows us to find the function that best fits the model.

Possible reasons why other models are not good enough:

1. **Bayesian results are good but not high enough, because there are too few adjustable parameters, and for small samples, it is difficult to find a suitable reliance on the test set.**
2. **The disadvantages of KNN are also obvious. It has high requirements on k value selection and is easy to produce errors**
3. **The perceptron whose step is the activation function is difficult to converge, especially when the amount of data is small, it is difficult to obtain learning results, and the possibility of error in fitting a function is very high. Even if back propagation is used, since the number of layers is only 1, it is difficult to improve.**
4. **The Gini coefficient is an important indicator, and a self-written decision tree gives it too much weight. Therefore, if there is no good way to solve the over-fitting or under-fitting situation caused by the selected Gini coefficient boundary is too high or too low.**