**FTEC 4003 Project Report**

Name: IP Shing On
SID: 1155109011

Name: WONG Shing
SID: 1155109027

**Task 1:**
**Background:**
We use Visual Studio with anaconda installed to do task 1. We import the corresponding libraries to evaluate each of the classification methods. We choose Bagging as the sample of Ensemble Method.

**Experimental evaluations:**
Performance of all the methods:

| | |
|---|---|
| Decision Tree | 0.3057297756989476 |
| k-Nearest Neighbor | 0.18764056977615937 |
| SVM | 0.07607090103397342 |
| Bayes | **0.42970297029702964** |
| Bagging | **0.4298139381261697** |

Therefore, the best 2 methods are Bayes and Bagging.

**Methods:**
**Decision Tree:**
Settings:
DecisionTreeClassifier(criterion= ‘gini’, splitter= ‘best’, max_depth=None, min_samples_split=2, min_samples_leaf=1, max_features= “sqrt”, random_state=50)

For random_state:

| max_depth | random_state |
|---|---|
| None | random_state(0) = 0.28996442815565376<br>random_state(1) = 0.290893489220208<br>random_state(2) = 0.2869397383658589<br>random_state(3) = 0.30073899539466636<br>random_state(4) = 0.28858191236779623<br>random_state(5) = 0.2835724994645534<br>random_state(6) = 0.28990472112193555 |

| | |
|---|---|
| | random_state(7) = 0.29394626092586595<br>random_state(8) = 0.2931498010538768<br>random_state(9) = 0.291363830696879<br>random_state(10) = 0.2973580062038721<br>**random_state(50) = 0.3057297756989476**<br>random_state(100) = 0.3007646559048428<br>random_state(150) = 0.2835757057313944<br>random_state(200) = 0.2835757057313944 |

We find that random_state=50 gives the highest performance.

Finding the best max_depth: (random_state=50)

| max_depth | performance |
|---|---|
| None | **0.3057297756989476** |
| < 8 | 0 |
| 8 | 0.0008710801393728222 |
| 9 | 0.0021743857360295715 |
| 10 | 0.005183585313174947 |
| 20 | 0.12378184305009403 |
| 50 | 0.29053177691309984 |
| 55 | 0.3000319863524896 |
| 56 | 0.30554964916011057 |
| 57 | **0.3057297756989476** |
| 58 | 0.3057297756989476 |
| 70 | 0.3057297756989476 |

We first find that the performance of max_depth<20 is very low, so we jump to test max_depth=50. Then, we find that when max_depth=57, the performance is the same as max_depth=None. We further increase the max_depth but the performance seems to be maximized. It means that the fully developed tree has a max_depth of 57. With random_state=50, we have a performance of 0.3057297756989476. That’s the best performance we can get after tuning the parameters.

**k-Nearest Neighbor:**
Settings:
KNeighborsClassifier(n_neighbors=1, weights= ‘distance’)

We set weights to be ‘distance’ instead of ‘uniform’ because we put the weights into consideration, which can provide a better performance.

To find the n_neightbors:

| n_neighbors | Performance | |
|---|---|---|
| | weights= ‘distance’ | weights= ‘uniform’ |
| 1 | **0.18764056977615937** | **0.18764056977615937** |
| 2 | 0.18475550254797787 | 0.06652126499454744 |
| 3 | 0.14000562271577172 | 0.13037166085946572 |
| 4 | 0.13406688866367455 | 0.045358447049778976 |
| 5 | 0.1042622950819672 | 0.09061488673139159 |
| 6 | 0.09632488672596073 | 0.030923694779116464 |
| 7 | 0.07237549265496238 | 0.05272013460459899 |

We can see that the performance decreases while n_neighbors increases. Finally, we set n_neightbors=1 and weights= ‘distance’. That means there are a total of 2 elements in each group.

**SVM**
Settings:
SVC(C=5, kernel=’linear’, random_state=0)

We choose ‘linear’ to be the kernel because the dataset can be linearly classified. Linear classification, which is the main topic of this course, can do classification fast and accurately.

Since running a large amount of data with SVM requires plenty of time, we sample the data from 300000th.

| C | Performance(linear) | Performance(rbf) |
|---|---|---|
| 1 | 0.07480096278466951 | 0.0021715526601520083 |
| 2 | 0.07507396449704141 | 0.010559662090813094 |

| | | |
|---|---|---|
| 3 | 0.0747871158830063 | 0.013033424427159974 |
| 4 | 0.07417022065640647 | 0.013445378151260505 |
| 5 | **0.07607090103397342** | 0.014276716355238295 |
| 6 | 0.07601476014760147 | 0.015107007973143096 |
| 7 | 0.07438934122871947 | 0.015107007973143096 |
| 8 | 0.07408779403593257 | 0.015107007973143096 |
| 9 | 0.07447202667654687 | 0.01510383889238515 |
| 10 | 0.07548566142460685 | 0.01510383889238515 |
| 20 | 0.07445823300611225 | 0.015100671140939596 |
| 50 | 0.07538802660753881 | 0.015100671140939596 |

Therefore, we find that c=5 with a linear kernel gives the best performance.

**Bayes:**
Settings:
CategoricalNB(alpha=3, fit_prior=False, class_prior=None)

There are five models of Naive Bayes including Gaussian Naive Bayes, Multinomial Naive Bayes, Complement Naive Bayes, Bernoulli Naive Bayes and Categorical Naive Bayes.

Referring to the formulas in the lecture notes $\Pr[A_i = X_i | Y = Yes] = \frac{N_{i,Yes}+1}{N_{Yes}+c}$ , we first eliminate Gaussian Naive Bayes and Complement Naive Bayes. Then, we eliminate Bernoulli Naive Bayes because the features are not necessarily binary-valued. For instance, age, region_code, Annual_Premium consists of a range of values instead of only binary-value. For Multinomial Naive Bayes, since it is mainly used for text classification where data are represented as word vector counts, it is eliminated. Finally, looking at Categorical Naive Bayes, it is good for classification with categorically distributed discrete features. In our train dataset, each feature like Driving_License with 0 and 1 as well as Vehcle_Damage with 'No, and 'Yes' has its own categorical distribution. Therefore, we finally choose to implement Categorical Naive Bayes with the formula $P(x_i = t \mid y = c \,;\, \alpha) = \frac{N_{tic} + \alpha}{N_c + \alpha n_i}$ , where 'alpha' (Additive Laplace smoothing parameter (0 for no smoothing)) is behaving like 'c' in the formula of the lecture note. When alpha becomes zero, the whole expression becomes zero when Ntic is zero.

For parameter fit_prior, we just keep the classes with a uniform prior, so we set it to False. For parameter class_prior, since the classes have uniform prior probabilities, we set the class_prior to None.

Finding a suitable alpha:

| alpha | Performance |
|---|---|
| 0 (without smoothing/without Laplace) | 0.38751625487646296 |
| 1 | 0.42792041791708346 |
| 2 | 0.42943904593639576 |
| **3** | **0.42970297029702964** |
| 4 | 0.42902588854760865 |
| 5 | 0.4294391244870041 |
| 6 | 0.42945321461735947 |
| 7 | 0.429226235948925 |
| 8 | 0.4292177705096757 |
| 9 | 0.42910081743869205 |
| 10 | 0.4291784702549575 |
| 15 | 0.42826878157423004 |
| 20 | 0.4284939236111111 |
| 30 | 0.42786716506853034 |
| 40 | 0.42776694823478456 |
| 60 | 0.4278213802435724 |
| 100 | 0.4276101584545257 |
| 200 | 0.4278460278460279 |
| 500 | 0.4221806429664909 |

The performance without Laplace is not good, with a performance of 0.3735070575461455. Once the Laplace is set, that is setting a value to alpha, the performance improves. Overall, the nearer the alpha to be 3, the better the performance of the Naive Bayes Algorithm. So, we finally set alpha to be 3.

To explain the alpha, we may first look at the two formulas: $\Pr[A_i = X_i | Y = Yes] = \frac{N_{i,Yes}+1}{N_{Yes}+c}$ -(1) and $P(x_i = t \mid y = c\,;\, \alpha) = \frac{N_{tic} + \alpha}{N_c + \alpha n_i}$ -(2). The former formula that is in the lecture notes has numerator+1 and denominator+c. The latter formula that is in the implementation has numerator+alpha and denominator+ni*alpha.

In the training dataset, we have only 2 classes to predict, so c=2.
ni is the number of available categories of feature i. For instance, in the training dataset, previously_insured has 2 categories, so ni for previously_insured is 2.

We have found two relations:

1. (1) will be greater than (2) when the raw probability increases, for any alpha with a fixed value on ni.
2. For a fixed raw probability, an increase of ni will decrease (2).

1. Let alpha = 3, ni=2, c=2

| Ni,Yes | NYes | lecture result | Implementation result |
|---|---|---|---|
| 1 | 10 | 0.167 | 0.25 |
| 2 | 10 | 0.25 | 0.3125 |
| 3 | 10 | 0.333 | 0.375 |
| 4 | 10 | 0.4167 | 0.4375 |
| 5 | 10 | 0.5 | 0.5 |
| 6 | 10 | 0.5833 | 0.5625 |
| 7 | 10 | 0.667 | 0.625 |

2. Let alpha = 3, c=2

| Ni,Yes | NYes | ni | lecture result | Implementation result |
|---|---|---|---|---|
| 1 | 10 | 2 | 0.167 | 0.25 |
| 1 | 10 | 3 | 0.167 | 0.211 |
| 1 | 10 | 4 | 0.167 | 0.182 |
| 1 | 10 | 5 | 0.167 | 0.16 |

For the training dataset, the attributes are having different numbers of categories, resulting in different numbers of ni. Alpha is tuned to balance the differentiation brought by ni and the raw probability. Alpha is tuned to make a better decision. For instance, alpha is tuned to make (2) equals to (1).

**Baggings:**
Settings:
bayes = CategoricalNB(alpha=3, fit_prior=False, class_prior=None)
BaggingClassifier(base_estimator=bayes, n_estimators=15, random_state=4, bootstrap=True)

We have used DecisionTree, KNN, Bayes and SVM as the base_estimator. We find that Bayes performs the best.

Random state is set as to eliminate the randomness of the performance. And we find that the performance slight improves when random_state is set to 4.

bootstrap=True means that samples are drawn with replacement.

Finding the suitable n_estimators (number of classifiers):

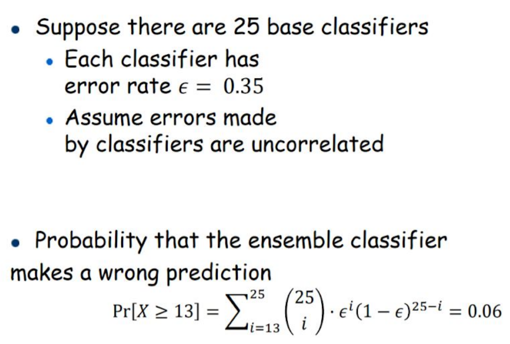
| | |
|---|---|
| n_estimators = 1 | 0.42861072902338376 |
| n_estimators = 2 | 0.4289800858180218 |
| n_estimators = 3 | 0.4295309403215151 |
| n_estimators = 4 | 0.42941273597886515 |
| n_estimators = 5 | 0.42910612065169523 |
| n_estimators = 6 | 0.4292081659604909 |
| n_estimators = 7 | 0.429137323943662 |
| n_estimators = 8 | 0.4291137151345107 |
| n_estimators = 9 | 0.4291293108189868 |
| n_estimators = 10 | 0.42945932566965506 |
| n_estimators = 11 | 0.42950657351889543 |
| n_estimators = 12 | 0.4297502475519858 |
| n_estimators = 13 | 0.42962473863761413 |
| n_estimators = 14 | 0.42962473863761413 |
| n_estimators = 15 | **0.4298139381261697** |
| n_estimators = 16 | 0.4295619634602685 |
| n_estimators = 17 | 0.42962473863761413 |
| n_estimators = 18 | 0.42952281358357647 |
| n_estimators = 19 | 0.42949105914718017 |

| | |
|---|---|
| n_estimators = 20 | 0.4295937465595068 |
| n_estimators = 21 | 0.4296011004126547 |
| n_estimators = 22 | 0.42951469131726644 |
| n_estimators = 23 | 0.4292473591549295 |
| n_estimators = 24 | 0.4292629262926293 |
| n_estimators = 25 | 0.4292629262926293 |
| n_estimators = 26 | 0.4293493207194324 |
| n_estimators = 27 | 0.4293020953638014 |
| n_estimators = 28 | 0.4293493207194324 |
| n_estimators = 29 | 0.4293020953638014 |
| n_estimators = 30 | 0.4293493207194324 |
| n_estimators = 40 | 0.4295693306198779 |
| n_estimators = 60 | 0.42948294829482947 |
| n_estimators = 100 | 0.42946742957746475 |
| n_estimators = 200 | 0.42949105914718017 |
| n_estimators = 500 | 0.42959295929592967 |

Though the performance keeps fluctuating, we find the best performance at n_estimators=15. Unlike Bayes, there isn’t a clear tendency of the performance while the parameter is increasing. n_estimators=15 means that original data is split into 15 subsets and each of them is classified under Naive Bayes. n_estimators=15 is concluded solely based on the experimental result.

**Comparison of the two best methods:**

Bagging has a slightly better performance than Bayes, with 0.4298139381261697 > 0.42970297029702964. Theoretically, this is because bagging repeats the classification a number of times and the probability of having a wrong decision is highly decreased. Mathematically, we can refer to the lecture notes.



- Suppose there are 25 base classifiers
  - Each classifier has error rate $\epsilon = 0.35$
  - Assume errors made by classifiers are uncorrelated

- Probability that the ensemble classifier makes a wrong prediction

$$\Pr[X \geq 13] = \sum_{i=13}^{25} \binom{25}{i} \cdot \epsilon^i (1-\epsilon)^{25-i} = 0.06$$

Among the methods, Bagging with Bayes as the base estimator gives the best performance. This is because Bayes itself is already performed well. Under Bagging, the performance will be further better off when applying the Bayes classification again and again .

**Task 2:**
**Background:**
We use Visual Studio with anaconda installed to do task 2. We import the corresponding libraries to evaluate each of the classification methods.

**Experimental evaluations:**
Performance of all the methods:

| | |
|---|---|
| Decision Tree | 0.611111111111111 |
| k-Nearest Neighbor | 0.1744421906693712 |
| Bayes | 0.5109374999999999 |
| SVM | 0.1602671118530885 |
| Bagging | **0.630071599045346** |

Therefore, the best method is Bagging with Decision Tree.

**Methods:**
**Decision Tree:**
Settings:
DecisionTreeClassifier(min_samples_split=257, random_state=0)

To tune the parameters, we first use GridSearchCV to find the approximate range of the best parameters. Then, we tune the parameters by ourselves.

First, set random state = 0 to eliminate the randomness of the performance.

Finding the min_samples_split:

| GridSearchCV range | Result (min_samples_split) | Performance |
|---|---|---|
| range(2,100,1) | 99 | 0.5846867749419954 |
| range(2,200,1) | **199** | **0.5875** |
| range(2,300,1) | **299** | **0.5898942420681551** |
| range(2,400,1) | 398 | 0.5707317073170731 |
| range(2,500,1) | 449 | 0.5707317073170731 |

We then search with a shorter range. Starting from the second group to the third group.

| GridSearchCV range | Result (min_samples_split) | Performance |
|---|---|---|
| range(190,220,1) | 218 | 0.5873417721518988 |
| range(221,251,1) | 221 | 0.5873417721518988 |
| range(252,282,1) | **281** | **0.5898942420681551** |
| range(283,300,1) | 299 | 0.5898942420681551 |

We can see that the third group gives the best performance. We then further fine tune min_samples_split by ourselves.

| min_samples_split | Performance |
|---|---|
| 252 - 255 | 0.5880861850443599 |
| 256 | 0.5880861850443599 |
| **257** | **0.6111111111111111** |
| 258 - 268 | 0.6111111111111111 |
| 269 - 282 | 0.5898942420681551 |

We find the best performance at min_samples_split=257, with performance of 0.611.

We then modify the random_state. However, we find that random_state does not affect the performance. We try random_state with values from 1 to 17, but the performance is still 0.611. On the other hand, for max_depth, since we do not set the max_depth, it will be None by default which means the tree is expected to be fully developed. We further test this by putting values to max_depth.

With max_split=257, random_state=0

| max_depth | performance |
|---|---|
| 1 | 0 |
| 2 | 0.5077844311377245 |
| 3 | 0.4131455399061033 |
| 4 | 0.5361366622864652 |
| 5 | 0.5944645006016848 |
| 6 | 0.5944645006016848 |
| 7 | 0.6060606060606062 |
| 8 | 0.6095017381228274 |
| 9 | 0.6095017381228274 |
| 10 | 0.6095017381228274 |
| 11 | 0.6095017381228274 |
| **12** | **0.611111111111111** |
| 13 | 0.611111111111111 |
| 14 | 0.611111111111111 |
| 15 | 0.611111111111111 |

We find the best performance at max_depth=12. Also, the performance keeps to be 0.6111 even max_depth increases. Thus, we can conclude that the fully developed decision tree has a max_depth of 12.

**k-Nearest Neighbor:**
Settings:
KNeighborsClassifier(n_neighbors=1, weights= ‘distance’)

We set weights to be ‘distance’ instead of ‘uniform’ because we put the weights into consideration, which can provide a better performance.

To find the n_neightbors:

| n_neighbors | weights= ‘distance’ | weights= ‘uniform’ |
|---|---|---|
| 1 | **0.1744421906693712** | **0.1744421906693712** |
| 2 | 0.1744421906693712 | 0.06039076376554174 |
| 3 | 0.14052697616060225 | 0.128686327077748 |
| 4 | 0.13559322033898305 | 0.07692307692307693 |
| 5 | 0.12446958981612447 | 0.12037037037037038 |
| 6 | 0.13037037037037036 | 0.06427221172022685 |
| 7 | 0.0947867298578199 | 0.09059233449477352 |

We find the n_neighbors by ourselves because GridSearchCV seems to be inaccurate for k-nearest neighbors using training dataset as sample. We find that n_neighbors=1 gives the best performance.

**SVM**
SVC(C=6, kernel=’linear’, random_state=0)
Finding the best C:

| C | Performance (linear) |
|---|---|
| 1 | 0.15177065767284992 |
| 2 | 0.15075376884422112 |
| 3 | 0.15126050420168066 |
| 4 | 0.15151515151515152 |
| 5 | 0.14310051107325383 |
| 6 | **0.1602671118530885** |
| 7 | 0.15228426395939085 |
| 8 | 0.15640599001663896 |
| 9 | 0.14839797639123103 |
| 10 | 0.14625850340136054 |
| 11 | 0.15488215488215487 |
| 15 | 0.14625850340136054 |
| 30 | 0.14864864864864866 |

We find that the best performance is at C=6.

**Bayes:**
Settings:
CategoricalNB(alpha=0, fit_prior=True, class_prior=None)

The reason for choosing CategoricalNB is mentioned in Task 1 Bayes.

Finding the best alpha: (set fit_prior=True, class_prior=None)

| alpha | Performance |
|---|---|
| **0** | **0.5109374999999999** |
| 1 | 0.507537688442211 |
| 2 | 0.4967658473479948 |
| 3 | 0.5006587615283268 |
| 4 | 0.4899057873485868 |
| 5 | 0.48707482993197276 |
| 6 | 0.484931506849315 |
| 7 | 0.4834254143646408 |
| 8 | 0.48543689320388345 |
| 9 | 0.4804469273743016 |
| 10 | 0.4797768479776847 |
| 15 | 0.4663805436337625 |
| 30 | 0.44052863436123346 |
| 50 | 0.39694656488549623 |
| 80 | 0.3653543307086614 |
| 100 | 0.33599999999999997 |

We find the alpha=0 gives the best performance.

Then, we look at the parameter ‘fit_prior’.

| | |
|---|---|
| fit_prior=False | 0.41870350690754515 |
| **fit_prior=True** | **0.5109374999999999** |

We find that alpha=0 and fit_prior=True gives the best performance with a value of 0.5109374999999999.

alpha=0 means that Laplace is not implemented. fit_prior=Ture means that the classes have prior probabilities rather than uniform prior. For class_prior, we would like to modify the class prior probabilities according to the data, so we set it to None. Therefore, we finally have the setting as CategoricalNB(alpha=0, fit_prior=True, class_prior=None).

**Bagging**
Settings:
D = tree.DecisionTreeClassifier(min_samples_split=39, random_state=0)
BaggingClassifier(base_estimator=D, n_estimators=10, random_state=4, bootstrap=True)

We have used DecisionTree, KNN, Bayes and SVM as the base_estimator. We find that Bayes performs the best.
Random state is set in order to eliminate the randomness of the performance. And we find that the performance is slightly improved when random_state is set to 4.
bootstrap=True means that the samples are drawn with replacement.

We have tried all the combinations between min_samples_split from 2 to 100 and n_estimators from 1 to 100. And we finally find the best performance at min_samples_split=39 and n_estimators=10, with a performance of 0.630071599045346.

**Additional Modification:**
Apart from the classifier tools, we do modification on the input data.

We first drop the columns that are unimportant.

```
    csv = csv.drop(columns=['Exited'])
    csv = csv.drop(columns=['RowNumber'])
    csv = csv.drop(columns=['CustomerId'])
    csv = csv.drop(columns=['Surname'])
```

Then, we group the data using ‘pd.DataFrame’.

```
    csv = pd.DataFrame(pd.get_dummies(csv))
```

All the features with non-numeric values will be changed to numeric, for example, for gender, male is changed to 0 while female is changed to 1. Besides, some features are separated out to form a new group. For instance, Geography with 3 choices (France, Germany, Spain) will be converted to three separate groups: ‘France’, ‘Germany’, ‘Spain’. Customers from ‘France’ will have value 1 in ‘France’, value 0 in ‘Germany’ and value 0 in ‘Spain’.

Lastly, we have observed some relationships and modified the ‘Balance’.

```
     for i in range(len(csv['Balance'])):
        if((csv['Balance'][i] == 0) and (csv['Tenure'][i] > 8)):
            csv['Balance'][i] = csv['Balance'].mean() +
csv['Balance'].std()*1
        if((csv['Balance'][i] == 0) and (csv['Tenure'][i] > 5)):
            csv['Balance'][i] = csv['Balance'].mean()

    csv['Balance'] = preprocessing.minmax_scale(
        csv[['Balance']], feature_range=(0, 1))
```

When ‘Balance’=0 and ‘Tenure’>8, ‘Balance’ is changed to the mean+1sd of ‘Balance’.
When ‘Balance’=0 and ‘Tenure’>5, ‘Balance’ is changed to the mean of ‘Balance’.

We find the relationships by looking at Balance, Tenure and estimated_salary. In normal sense, Balance should not be 0 when Tenure is relatively long, which means the customers are expected to have a higher Balance. Thus, we try to increase the Balance. We have found that the longer the Tenure, the higher the estimated salaries. Therefore, we modify Balance to mean+1sd for ‘Tenure>8’ and mean for ‘8>Tenure>5’ respectively. For the conditions, we have tried different values and finally come out with 8 and 5.