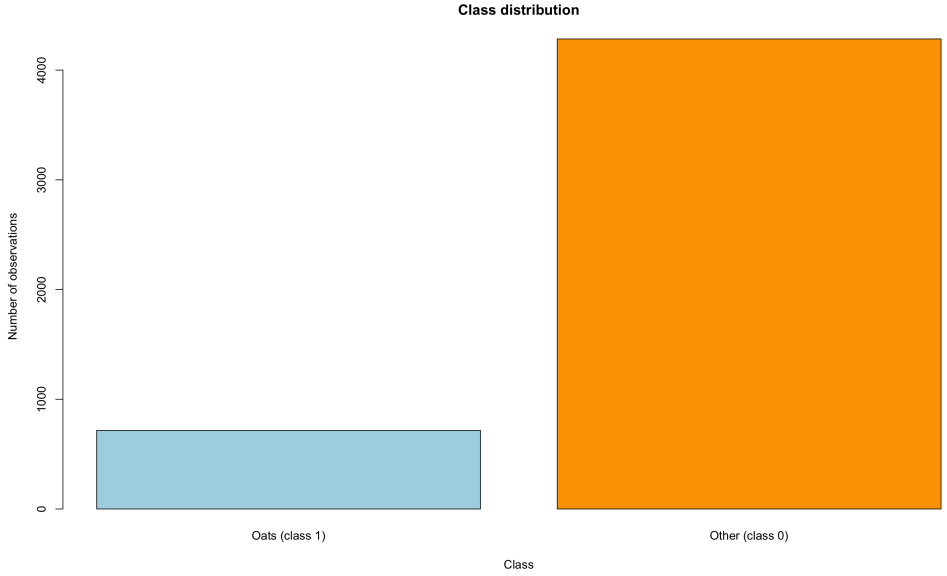
Assignment 2

[redacted]

Question 1.

The proportion of "Oats" (`class = 1`) to "Other" (`class = 0`) is 0.1671335, suggesting that the dataset is very imbalanced. The majority class is "Other", and the minority class is "Oats".



The `str(WD)` function shows that 20 out of 20 independent variables have the `num` data type. Investigating further, the summary of the real-valued attributes is produced as shown below.

	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.	SD.
A01	0	0	0	0.1058	0.197	0.575	0.1243979
A02	-25.81	-20.135	-16.288	-15.974	-12.588	0.431	4.98322929
A03	0.048	0.103	0.2145	0.3663	0.604	1.289	0.3211352
A04	0.006	0.015	0.077	0.2739	0.522	1.183	0.32097538
A05	-22.423	-16.214	0	-7.677	0	0	8.06438021
A10	-16.346	-8.681	0	-4.258	0	0	4.54562987
A11	0	0	0.006	0.01042	0.01	0.155	0.01816715
A13	0	0	0.074	0.1065	0.196	0.568	0.1211823
A14	0.007	0.035	0.149	0.3134	0.572	1.169	0.32980328
A18	0.079	0.829	0.943	1.077	1.324	1.944	0.34323471
A20	0	0	0.53	0.5829	1.061	2.197	0.52912605
A21	1.083	3.577	6.3	7.192	9.888	107.165	4.95167931
A22	0.154	0.412	0.557	0.6689	0.917	1.664	0.34254542
A23	0	0	0	0.2534	0.519	0.728	0.27185773

	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.	SD.
A24	1.015	3.123	4.632	5.843	8.191	15.109	3.32295926
A25	0.041	0.493	0.632	0.583	0.679	0.836	0.1288325
A27	0.001	0.031	0.082	0.2901	0.542	1.072	0.32758946
A28	0	0	0	0.3101	0.6813	1.131	0.34524868
A29	-4.042	0.076	0.195	0.3191	0.5823	1.224	0.36426318
A30	0	0	0.018	0.0264	0.044	0.131	0.03151993

Figure 1.1. Summary of real-valued independent variables using `summary(WD)` and `sd()`

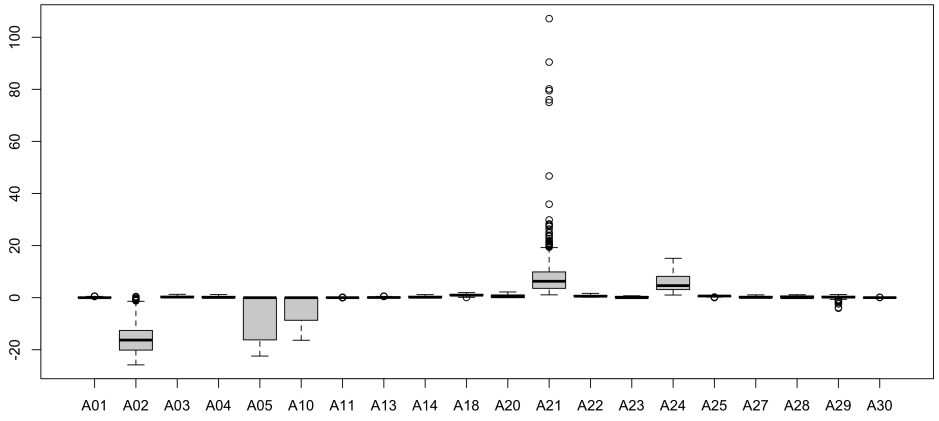


Figure 1.2. Distribution of Features

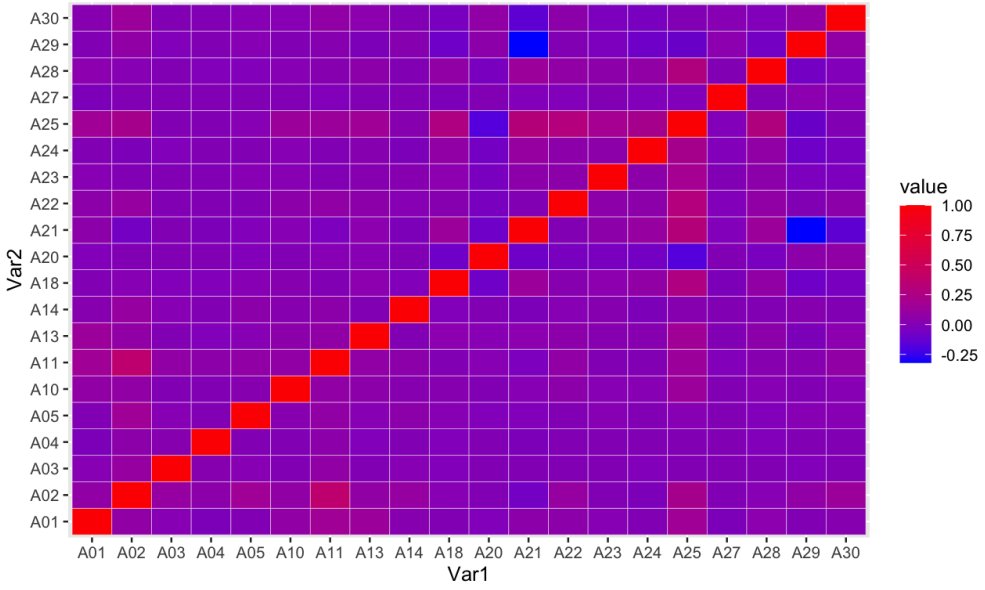


Figure 1.3. Feature correlation heat map

Looking at the table and the boxplot above, there are several worth noting observations:

- Many variables, such as `A01`, `A11`, and `A13`, have 0 for `Min.`, `1st Qu.`, and even `Median`, indicating many zero values. These are potential candidates for feature omissions as they might not carry much information, hurting the performances of models.
- Some of the features, namely `A02`, `A05`, `A10`, and `A29`, have negative values.
- Two variables `A21` and `A24` have exceptionally large `Max.` values, suggesting potential outliers.
- A few variables, such as `A30` and `A11`, have very low `SD.`. They are also candidates for omissions, as such features potentially carry little information.

Besides, the heat map shows some of the useful points:

- The variables in the dataset are mostly weakly correlated.
- Although some of them are correlated moderately (shown as brighter red), no features have too strong correlations.
- Some of the features have negligible negative correlations.

Question 2.

To make the dataset suitable for the model fitting, the following steps were performed:

- Convert the data type of the label column `Class` from `int` to `factor`.

No other steps were needed:

- No missing values (as stated on the UCI Machine Learning Archive's website and confirmed using `any(is.na(WD))`.
- No need for feature scaling (as all five modelling techniques do not depend on distance-based calculations and are able to deal with raw feature magnitudes).

Question 3.

The dataset was split into two: training and test sets.

Question 4.

1. Decision Tree

A classification model using the Decision Tree technique was implemented using `tree()` from the package `tree`.

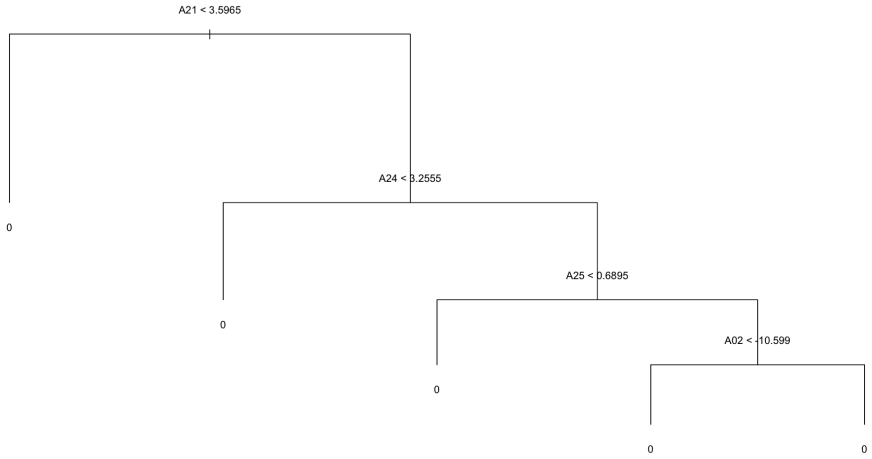


Figure 4.1. Decision Tree visualisation

2. Naive Bayes

A classification model using the Naive Bayes technique was implemented using `naiveBayes()` from the package `e1071`.

3. Bagging

A classification model using the Bagging technique was implemented using `bagging()` from the package `adabag`. As the parameter `mfinal = 100` by default, a comprehensive plot cannot be displayed. Therefore, the visualisation for Bagging was not included.

4. Boosting

A classification model using the Boosting technique was implemented using `boosting()` from the package `adabag`. As the parameter `mfinal = 100` by default, a comprehensive plot cannot be displayed. Therefore, the visualisation for Boosting was not included.

5. Random Forest

A classification model using the Random Forest technique was implemented using `randomForest()` from the package `randomForest`. As the parameter `ntree = 500` by default, a comprehensive plot cannot be displayed. Therefore, the visualisation for Random Forest was not included.

Question 5.

1. Decision Tree

- Confusion matrix

	Predicted: Other (0)	**Predicted: Oats (1)**
Actual: Other (0)	1270	0
Actual: Oats (1)	230	0

Figure 5.1.1. Decision Tree's confusion matrix

- Performance metrics

Precision	**Recall**	**Accuracy**	**F1-score**
N/A	0	0.8466667	0

Figure 5.1.2. Decision Tree's performance metrics

	Predicted: Other (0)	**Predicted: Oats (1)**
Actual: Other (0)	1141	129
Actual: Oats (1)	178	52

Figure 5.2.1. Naive Bayes's confusion matrix

- Performance metrics

Precision	**Recall**	**Accuracy**	**F1-score**
0.28729282	0.22608696	0.79533333	0.25304136

Figure 5.2.2. Naive Bayes's performance metrics

3. Bagging

- Confusion matrix

	Predicted: Other (0)	**Predicted: Oats (1)**
Actual: Other (0)	1270	0
Actual: Oats (1)	229	1

Figure 5.3.1. Bagging's confusion matrix

- Performance metrics

Precision	**Recall**	**Accuracy**	**F1-score**
1	0.00434783	0.84733333	0.00865801

Figure 5.3.2. Bagging's performance metrics

4. Boosting

- Confusion matrix

	Predicted: Other (0)	Predicted: Oats (1)
Actual: Other (0)	1218	52
Actual: Oats (1)	185	45

Figure 5.4.1. Boosting's confusion matrix

- Performance metrics

Precision	Recall	Accuracy	F1-score
0.46391753	0.19565217	0.842	0.27522936

Figure 5.4.2. Boosting's performance metrics

5. Random Forest

- Confusion matrix

	Predicted: Other (0)	Predicted: Oats (1)
Actual: Other (0)	1265	5
Actual: Oats (1)	223	7

Figure 5.5.1. Random Forest's confusion matrix

- Performance metrics

Precision	Recall	Accuracy	F1-score
0.58333333	0.03043478	0.848	0.05785124

Figure 5.5.2. Random Forest's performance metrics

Question 6.

- The confidence of predicting "Oats" for each case was calculated using the code attached in the Appendix.
- ROC (Receiver Operating Characteristic)

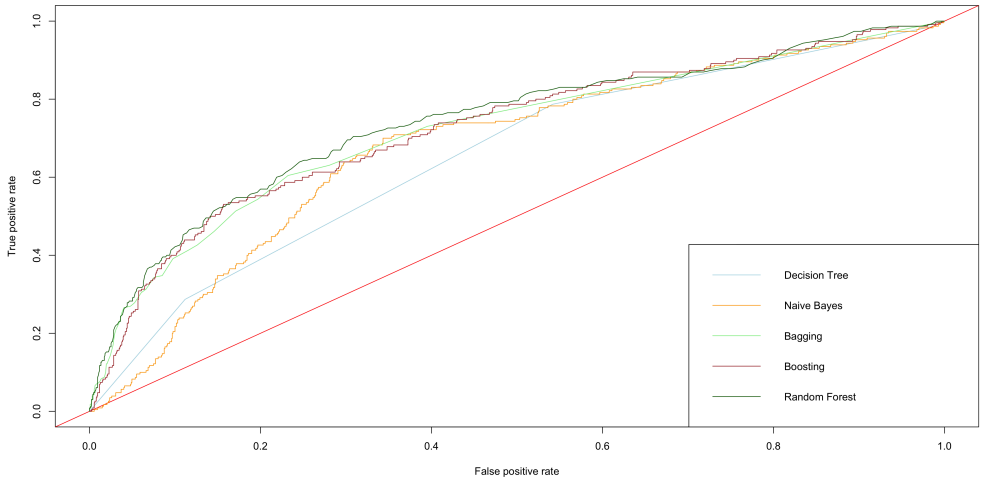


Figure 6.1. Receiver Operating Characteristic (ROC) for models

- AUC (Area under ROC curve)

Decision Tree	Naive Bayes	Bagging	Boosting	Random Forest
0.6543958	0.6778637	0.7213745	0.7245395	0.7403612

Figure 6.1. AUC for models

Question 7.

Metrics	Decision Tree	Naive Bayes	Bagging	Boosting	Random Forest
Precision	N/A	0.28729282	1	0.46391753	0.58333333
Recall	0	0.22608696	0.00434783	0.19565217	0.03043478
Accuracy	0.8466667	0.79533333	0.84733333	0.842	0.848
F1-score	0	0.25304136	0.00865801	0.27522936	0.05785124
AUC	0.6543958	0.6778637	0.7213745	0.7245395	0.7403612

Figure 7.1. Performance metrics of all classifiers

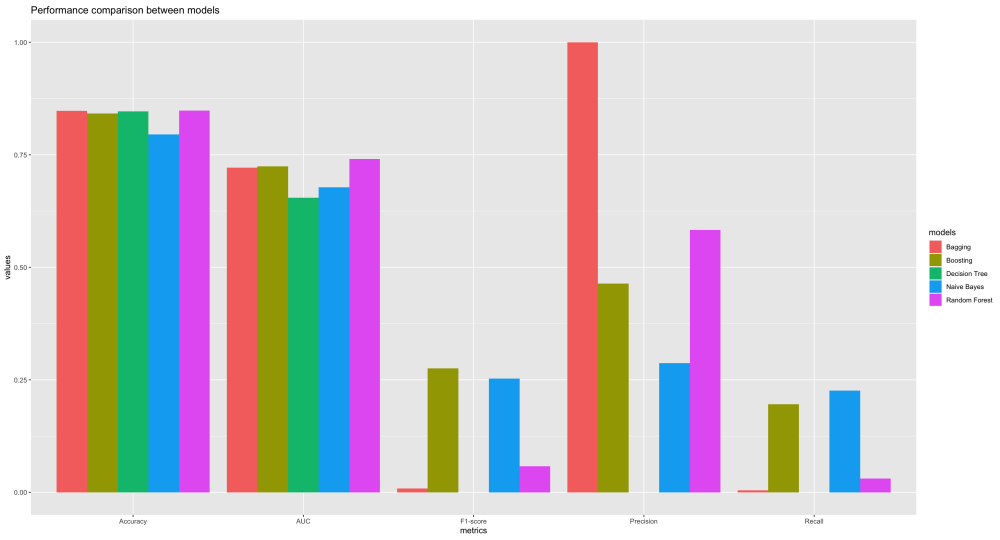


Figure 7.2. Performance metrics of all classifiers

As inferred from the table, no model outperforms the others across all metrics. For example, Random Forest has the highest accuracy, but it has the second-lowest recall at the same time. Or, Bagging has the best precision of 1 and the second-highest accuracy, but it has the lowest recall of 0.0043. Therefore, it can be concluded that there is no single best classifier, and different models perform better under different evaluation criteria.

For example:

- On the one hand, a model having higher precision, suggesting that it makes fewer false positive errors, is more valuable when false positives are costly, such as marking important emails as spam in spam filters.
- On the other hand, a model having higher recall, suggesting that it detects more actual positives, is more valuable when missing positives is serious, such as fraud detections.

Hence, model performance heavily depends on the context of the problems: the specific goals and constraints of the task.

Question 8.

1. Important attributes in the prediction

Decision Tree

To determine the importance of attributes, the features used for splitting at the nodes were investigated using the function `summary(WD.tree)`. The output showed that variables used in the tree construction were `A21`, `A24`, `A25`, and `A02`. These are the most critical attributes in the prediction.

Naive Bayes

For this classifier, it does not produce variable importance scores. It assumes feature independence and uses all attributes equally.

Bagging

The importance scores of attributes used in the Bagging classifier were extracted from `WD.bag$importance` as shown in the table below.

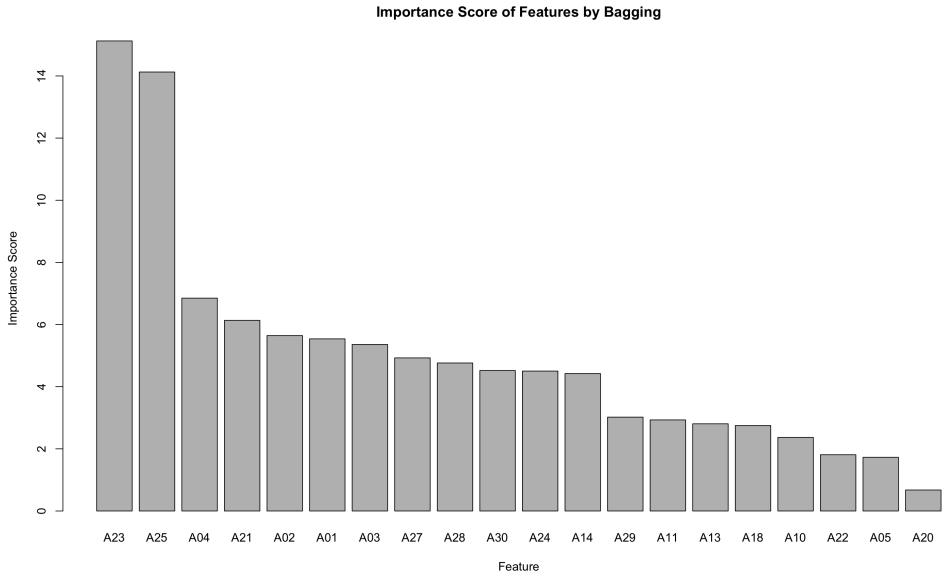


Figure 8.1. Importance Score of Features by Bagging

Attribute	Importance Score
A23	15.1266674
A25	14.1265014
A04	6.85017334
A21	6.13353526
A02	5.64475478
A01	5.53898158
A03	5.35820928
A27	4.92667533
A28	4.76281145
A30	4.51960248
A24	4.5026409
A14	4.4225182
A29	3.02027587
A11	2.93062756
A13	2.80575719
A18	2.74938724
A10	2.3676611
A22	1.81037902
A05	1.72738556
A20	0.67545511

Figure 8.2. Importance Score of Features by Bagging

From the table, the most critical variables in the Bagging classifier are `A23`, `A25`, `A04`, `A21`, and `A02`.

Boosting

The importance scores of attributes used in the Bagging classifier were extracted from `WD.boost$importance` as shown in the table below.

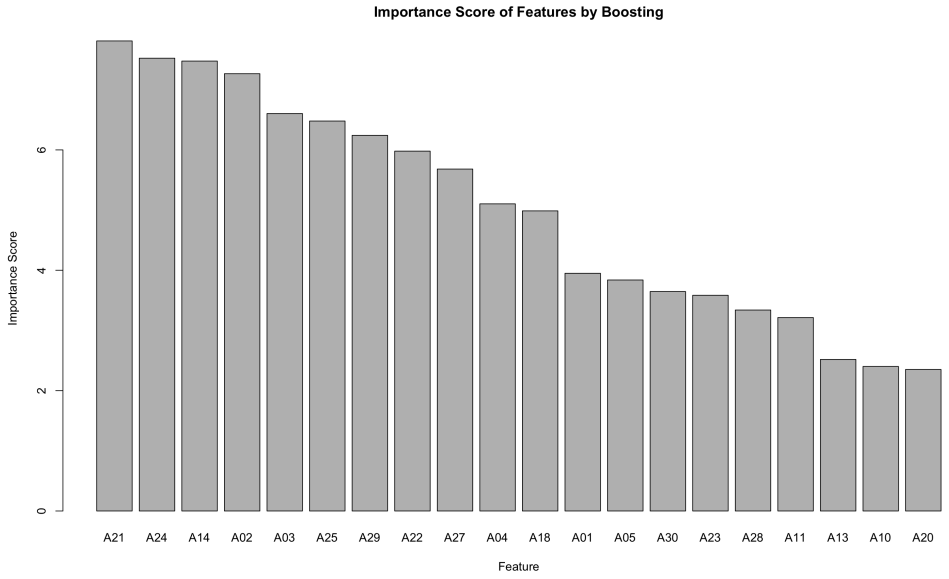


Figure 8.3. Importance Score of Features by Boosting

Attribute	Importance Score
A21	7.81087745
A24	7.52401397
A14	7.47631039
A02	7.26625847
A03	6.60449085
A25	6.47989813
A29	6.24189131
A22	5.9798735
A27	5.68184006
A04	5.10365557
A18	4.98683752
A01	3.94995364
A05	3.83853991
A30	3.64619248
A23	3.58446736
A28	3.33925877
A11	3.21429579
A13	2.51834622
A10	2.40137939
A20	2.35161924

Figure 8.4. Importance Score of Features by Boosting

From the table, the most critical variables in the Boosting classifier are `A21`, `A24`, `A14`, `A02`, and `A03`.

Random Forest

The importance scores of attributes used in the Random Forest classifier were extracted from `WD.rf$importance` as shown in the table below.

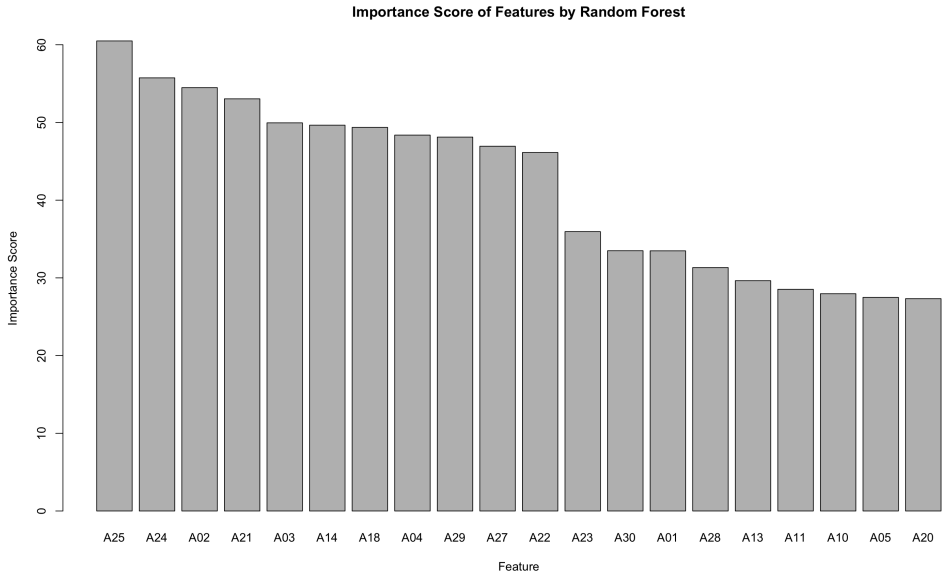


Figure 8.5. Importance Score of Features by Random Forest

Attribute	Importance Score
A25	60.4926945
A24	55.7395535
A02	54.4817953
A21	53.0411384
A03	49.9509423
A14	49.6520328
A18	49.3658151
A04	48.3750604
A29	48.1174917
A27	46.9431087
A22	46.1272374
A23	35.9492846
A30	33.4923657
A01	33.4804121
A28	31.3260671
A13	29.6345977
A11	28.5309978
A10	27.9582547
A05	27.4885427
A20	27.3259839

Figure 8.6. Importance Score of Features by Random Forest

From the table, the most critical variables in the Random Forest classifier are `A25`, `A24`, `A02`, `A21`, and `A03`.

2. Attributes omission

The results in the previous section show that `A20` consistently stands at the last place in the important variable lists for Bagging, Boosting, and Random Forest. Additionally, the attribute is not in the list of critical variable for Decision Tree. Therefore, it is safe to omit this attribute from the data.

Besides, among the top 10 of least important variables for Bagging, Boosting, and Random Forest, the attributes `A10`, `A05`, `A11`, and `A13` also appears in the lists of all three classifiers. Referring to the analysis in Question 1, these variables are stated as potential candidates for omissions due to their large numbers of 0. Hence, it can be concluded that `A10`, `A05`, `A11`, and `A13` can be safely omitted from the data.

Question 9.

Metrics	Decision Tree	Naive Bayes	Bagging	Boosting	Random Forest
Precision	N/A	0.28729282	1	0.46391753	0.58333333
Recall	0	0.22608696	0.00434783	0.19565217	0.03043478
Accuracy	0.8466667	0.79533333	0.84733333	0.842	0.848
F1-score	0	0.25304136	0.00865801	0.27522936	0.05785124
AUC	0.6543958	0.6778637	0.7213745	0.7245395	0.7403612

Figure 9.1. Performance metrics of all classifiers

From the table above, given that the dataset is imbalanced, there are two models that have relatively better performances compared to others: Naive Bayes and Boosting.

In terms of Naive Bayes, the model has the balance between precision and recall, approximately around 0.20-0.30 for both of the metrics. This results in its one of the highest F1-score, suggesting that the model shows its strengths when it comes to managing the trade-off between recognising correctly positives while limiting false alarms. However, its precision and recall are both low compared to typical standards, showing that its performance is not good under the imbalanced dataset. According to Kim & Lee (2023), Naive Bayes, similar to other traditional classifiers, sometimes has poor performance in predicting minority instances due to "its sensitivity to class distribution".

The other model is Boosting. This model has the highest F1-score, suggesting its balance between precision and recall. It is worth-noting that this model has a much higher precision and a slightly lower recall compared to Naive Bayes, showing that it is limiting false positives better but recognising positives less effectively. This model, similar to Naive Bayes, has better performance compared to the other four but is still inferior compared to typical standards. When the dataset's imbalance is not severe, Boosting (AdaBoost) might perform better (Shahri et al., 2021). The reason is that the algorithm has an intrinsic mechanism to deal with misclassified instances. It increases the weight of such instances (commonly from the minority class) automatically, allowing it to be "trained to give more attention to the underrepresented class" (Reed, 2024).

Other models, although having a very high accuracy, are considered to have poor performance in detecting correctly positive instances and have bias towards the majority class.

The reason why some models work better than others when it comes to dealing with imbalanced data is some models, such as tree-based, have a natural way of handling imbalance due to their structures (Olamendy, 2024). Besides, Naive Bayes also has superior performance after modifications. Complement Naive Bayes is an example (*Complement Naive Bayes (CNB) Algorithm*, 2020). Therefore, it is important to understand the working mechanisms, strengths, and weaknesses of each algorithm, as well as to explore the recommended modifications for handling imbalanced datasets to enhance model performance.

Question 10.

1. Implementation

In this question, I use the Decision Tree classifier in Question 4 to create a classifier that is simple enough for a person to classify "Oats" (1) and "Other" (1) by hand. Looking at the existing model, it is evident that the model classifies all cases as 0. The analysis performed in question 1 shows that the proportion of "Oats" to "Other" crops is 0.1671335, which means that the number of observations for class 0 is much higher than for class 1. This suggests that there is an imbalance in the dataset. In such a dataset, decision trees have a tendency to favor the majority class, leading to high misclassification rates for the minority class (GeeksforGeeks, 2024d). Therefore, to make the new model classify some cases as class 1, **only 2000 out of 4284 rows** with class 0 are included in the dataset for the new model, together with rows with class 1.

Besides, to further simplify the Decision Tree, **only the top 5 most important features** suggested by the Random Forest classifier are included in this model. The reason behind this decision is Random Forest is "extremely useful and efficient in selecting the important features" (Chen et al., 2020). It provides a built-in method that evaluates the importance of attributes, and the selection is reliable as this type of model can handle high dimensionality and capture complex relationships between variables (GeeksforGeeks, 2024c).

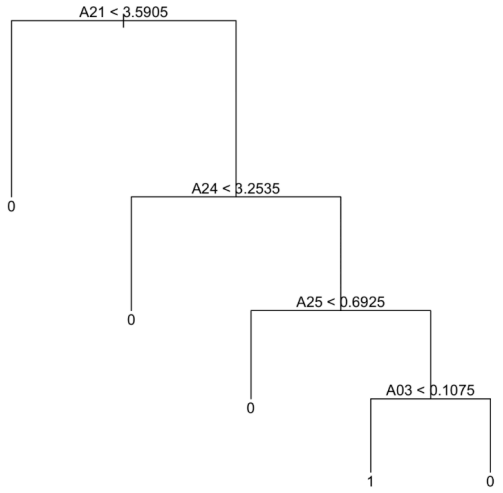


Figure 10.1.1. Decision Tree's visualisation

A person using this model to classify "Oats" and "Other" crops by hand can follow the steps below:

- If `A21` is less than 3.5905, then it is "Other".
- Otherwise:
  - If `A24` is less than 3.2535, then it is "Other".
  - Otherwise:
    - If `A25` is less than 0.6925, then it is "Other".
    - Otherwise:
      - If `A03` is less than 0.1075, then it is "Oats".
      - Otherwise, it is "Other".

2. Performance

With the test dataset in Question 3, I evaluate the model's performance.

- Confusion matrix

	Predicted: Other (0)	**Predicted: Oats (1)**
Actual: Other (0)	582	14
Actual: Oats (1)	183	22

Figure 10.2.1. Decision Tree's confusion matrix

- Performance metrics

Precision	**Recall**	**Accuracy**	**F1-score**
0.61111111	0.10731707	0.75405743	0.18257261

Figure 10.2.1. Decision Tree's performance metrics

3. Comparison

The summarise the performance of all models in Question 4 and the new Decision Tree is shown as below.

Metrics	Decision Tree	Naive Bayes	Bagging	Boosting	Random Forest	Decision Tree (Q10)
Precision	N/A	0.28729282	1	0.46391753	0.58333333	0.61111111
Recall	0	0.22608696	0.00434783	0.19565217	0.03043478	0.10731707
Accuracy	0.8466667	0.79533333	0.84733333	0.842	0.848	0.75405743
F1-score	0	0.25304136	0.00865801	0.27522936	0.05785124	0.18257261
AUC	0.6543958	0.6778637	0.7213745	0.7245395	0.7403612	0.6641103

Figure 10.3.1. Performance metrics of all classifiers

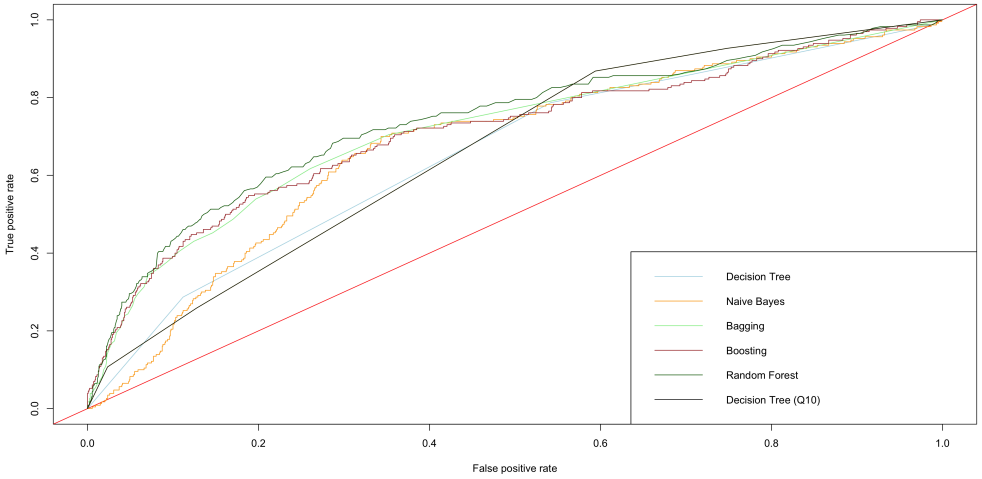


Figure 10.3.2. Receiver Operating Characteristic (ROC) for models

Comparing this model to the others in Question 4, it is clear that this model has the worst accuracy with a value of 0.75405743. This observation is expected due to the classifier's simplicity. In terms of AUC, the figure for this model is slightly higher than that of the Decision Tree in question 4, suggesting improvements in distinguishing between the two classes. However, it is still lower than other models, showing its inferior discriminative ability. When it comes to precision, the model achieves first place with 0.61111111. This means the probability that its predicted positives are correct is higher than that of other models. Regarding recall and F1-score, the Decision Tree has average values, with the rank of 3rd out of 5.

While the model has a strength when it comes to precision, its overall performance is still limited for the following reasons. The low AUC, accompanied with average recall and F1-score, indicates its underperformance compared to other complex classifiers that perform better overall. Therefore, it can be concluded that this model is not the most effective classifier compared to others.

Question 11.

1. Overview

One of the weaknesses of the models in previous questions is that they tend to be biased towards the majority class ("Other"), which is suggested by low recall but high precision, and hence a low F1-score. Therefore, I want to focus on improving F1-score and Recall, as well as reducing the model's bias towards the majority class.

Given the performance metrics for the five models above, I consider Random Forest and Boosting as the strongest candidates. After investigating further, I have come to the decision that Random Forest is the model I am choosing to work on in this question for the reasons below:

- Both Random Forest and Boosting provide approaches to achieve the desired improvements.
- However, Random Forest appears to be the safer classifier due to its inherent overfitting resistance from the bagging technique, simpler hyperparameter tuning, and less sensitivity to noisy data (GeeksforGeeks, 2024a).

As mentioned in Question 10, the dataset is highly imbalanced, leading to high misclassification rates for the minority class, which is "Oats" (1) in this case. According to Chen & Liaw (2004), there are two effective ways to handle this problem using Random Forest: Balanced Random Forest (BRF) and Weighted Random Forest (WRF). There is no clear evidence suggesting which model is superior. However, WRF might be more vulnerable to noise than BRF, as WRF allocates a weight to the minority class. Additionally, WRF is less computationally efficient with large datasets. Therefore, I chose to work on BRF in this question.

In the construction of the model, I decided to keep all available features in the dataset and not remove any of them. The rationale behind this is that BRF already performs downsampling, reducing a high number of observations in the training set. Removing features together with downsampling would reduce the amount of information used to train the model further, increasing the risk of underfitting.

According to Thölke et al. (2023), in an imbalanced dataset, the Accuracy metric shows "misleadingly high performances", and it is suggested that the Balanced Accuracy (BAcc) metric provides "more reliable performance evaluations". Additionally, a high F1-score demonstrates that the model can achieve high precision and high recall at the same time. Hence, I am mainly focusing on the BAcc and F1-score metrics.

2. Model construction

The idea behind BRF is that we sample the same number of observations from class 0 and class 1. This, potentially, improves the model's ability to learn from the minority class and classify them correctly (GeeksforGeeks, 2024b).

According to Fox et al. (2017), a BRF can be estimated using the `sampsize` argument from the `randomForest` package. Therefore, I am creating the model using this approach.

- Confusion matrix

	Predicted: Other (0)	**Predicted: Oats (1)**
Actual: Other (0)	1028	242
Actual: Oats (1)	95	135

Figure 11.2.1. Balanced Random Forest's confusion matrix

- Performance metrics

Precision	**Recall**	**Accuracy**	**F1-score**	**BAcc**
0.35809019	0.58695652	0.77533333	0.44481054	0.6982027

Figure 11.2.2. Balanced Random Forest's performance metrics

3. Comparison

This is the table summarising the performance metrics of all models so far.

Metrics	**Decision Tree**	**Naive Bayes**	**Bagging**	**Boosting**	**Random Forest**	**Balanced Random Forest**
Precision	N/A	0.28729282	1	0.46391753	0.58333333	0.35809019
Recall	0	0.22608696	0.00434783	0.19565217	0.03043478	0.58695652
Accuracy	0.8466667	0.79533333	0.84733333	0.842	0.848	0.77533333
F1-score	0	0.25304136	0.00865801	0.27522936	0.05785124	0.44481054
AUC	0.6543958	0.6778637	0.7213745	0.7245395	0.7403612	0.7413523
BAcc	0.5	0.5622561	0.50434783	0.5957378	0.51246149	0.6982027

Figure 11.3.1. Performance metrics of all classifiers

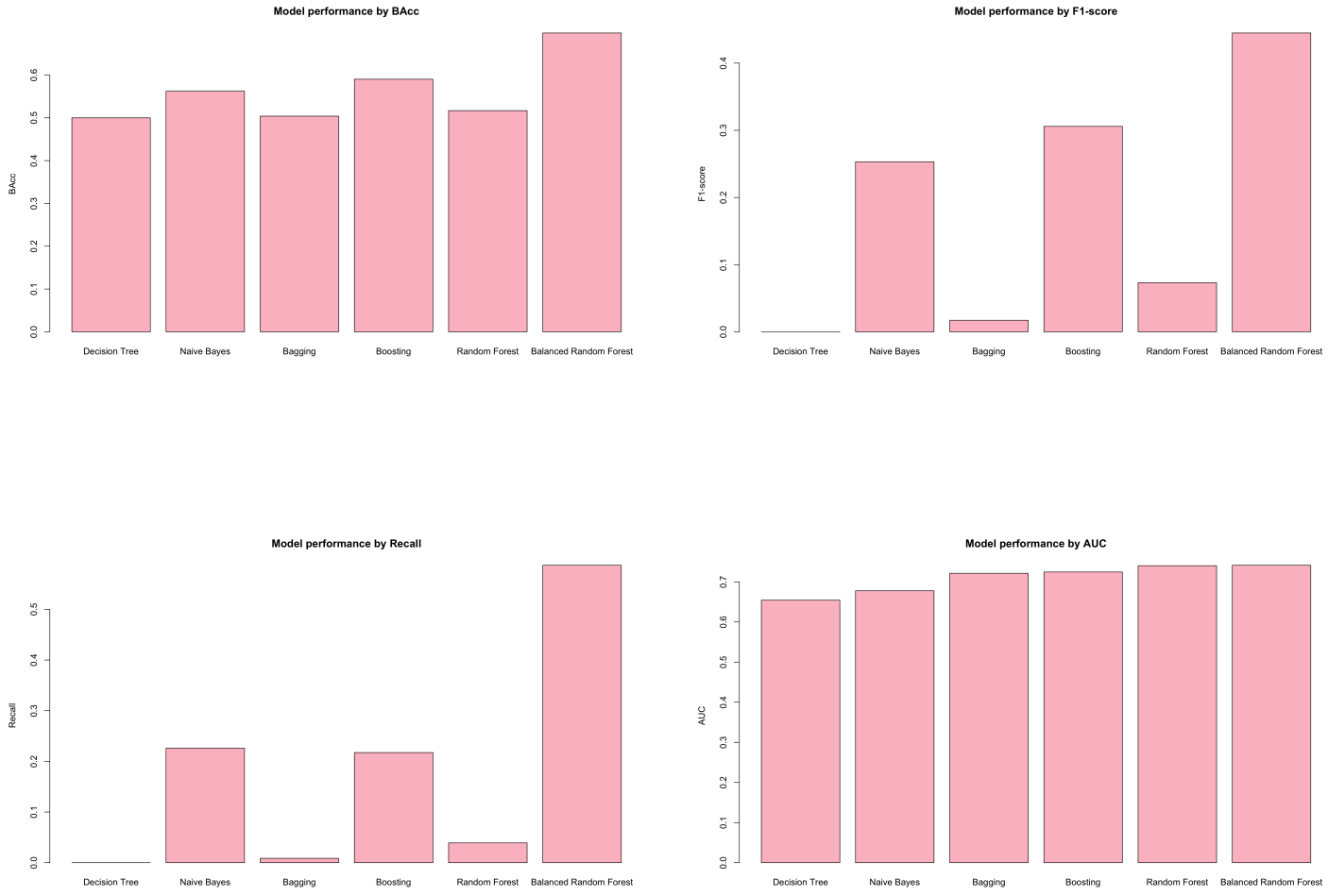


Figure 11.3.2. Performance metrics of all classifiers

Looking at four main metrics (BAcc, F1-score, Recall, AUC), it is evident that the Balanced Random Forest model has a superior performance compared to other classifiers in Question 4. On the one hand, in terms of BAcc, F1-score, and Recall, BRF ranks 1st with the figures of 0.6982027, 0.44481054, and 0.58695652, respectively, much higher than the rest of the models. Additionally, when it comes to AUC, this model also achieves the best result, approximately on par with the Random Forest model in Question 4. Precision and Accuracy for this model, on the other hand, are not as good as compared of others. However, as stated above, Accuracy in an imbalanced dataset is not a reliable metric and sometimes causes misleading evaluations. Besides, a lower Precision, accompanied by a higher Recall, is an expected observation and also the target of this model: to increase Recall and reduce bias towards the majority class. Therefore, it can be concluded that the Balanced Random Forest model has a higher performance than other models in Question 4.

Question 12.

1. Pre-processing

Setting up the Artificial Neural Network classifier involves:

- Recode the output `Class` as numeric.
- Normalise data (Min Max Scaling).

There is no missing value handling as there are no missing values in the dataset.

For the `class_weight` parameter, I set the value for class 0 as `1` and the value for class 1 as `6`.

Regarding feature selection for this model, I decide to compare two feature sets: the top 10 most important variables ranked by the Random Forest model in Question 4 and the full set of features. After training the model with both sets of features, I compare the performances and make the last decision on the list of attributes.

2. Implementation

Using the `keras` package, I build two ANN models.

The first model is the one trained using the top 10 most important variables ranked by the Random Forest model in Question 4.

- Confusion matrix

	Predicted: Other (0)	**Predicted: Oats (1)**
Actual: Other (0)	712	558
Actual: Oats (1)	62	168

Figure 12.2.1. ANN (1)'s confusion matrix

- Performance metrics

Precision	**Recall**	**Accuracy**	**F1-score**	**AUC**	**BAcc**
0.2314050	0.7304348	0.5866667	0.3514644	0.670404	0.6455323

Figure 12.2.2. ANN (2)'s performance metrics

The second model is the one trained using the full set of features.

- Confusion matrix

	Predicted: Other (0)	**Predicted: Oats (1)**
Actual: Other (0)	963	307
Actual: Oats (1)	125	105

Figure 12.2.3. ANN (2)'s confusion matrix

- Performance metrics

Precision	**Recall**	**Accuracy**	**F1-score**	**AUC**	**BAcc**
0.2548544	0.4565217	0.7120000	0.3271028	0.6482746	0.6073947

Figure 12.2.4. ANN (2)'s performance metrics

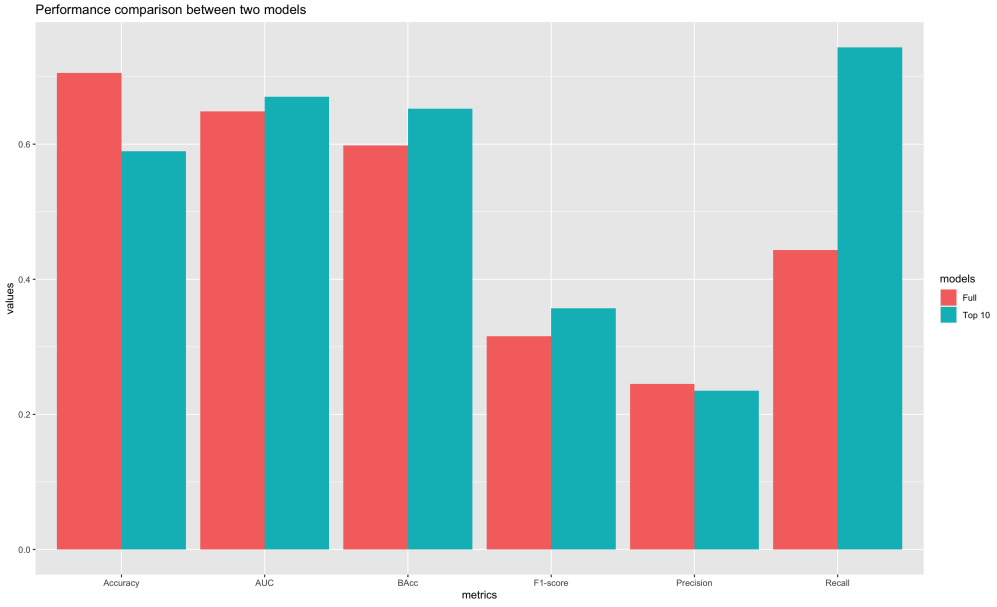


Figure 12.2.5. Performance metric comparison between ANN (1) and ANN (2)

As inferred from the tables and bar graph above, it is evident that Model 1 has higher AUC, BAcc, F1-score, and Recall, while Model 2 has higher Accuracy and Precision. As the dataset is imbalanced, metrics such as Recall, F1-score, and BAcc should be given greater emphasis, as they reflect the model's performance on classifying minority class cases better. Therefore, in spite of lower Accuracy and Precision, I consider Model 1 as the better model and would like to choose it as the final model.

3. Comparison

Metrics	Decision Tree	Naive Bayes	Bagging	Boosting	Random Forest	Balanced Random Forest	ANN
Precision	N/A	0.28729282	1	0.46391753	0.58333333	0.35809019	0.2314050
Recall	0	0.22608696	0.00434783	0.19565217	0.03043478	0.58695652	0.7304348
Accuracy	0.8466667	0.79533333	0.84733333	0.842	0.848	0.77533333	0.5866667
F1-score	0	0.25304136	0.00865801	0.27522936	0.05785124	0.44481054	0.3514644
AUC	0.6543958	0.6778637	0.7213745	0.7245395	0.7403612	0.7413523	0.6482746
BAcc	0.5	0.5622561	0.50434783	0.5957378	0.51246149	0.6982027	0.6073947

Figure 12.3.1. Performance metrics for all classifiers

Comparing this model to others, as inferred from the table above, it is readily apparent that the model has the highest Recall with 0.7304348, meaning the model is good at detecting minority class. In terms of Accuracy and Precision, it has one of the lowest figures, which is expected as my main objective is to train the model to classify the minority class better in an imbalanced dataset. Additionally, the low AUC suggests that the model still struggles with distinguishing the positives samples versus the negative samples, but it has the second highest BAcc, showing that the model is handling imbalance relatively well. In conclusion, depending on the context, if the priority is detecting as many positives as possible, I believe that this model has a good performance.

Question 13.

1. Introduction

In this question, I have selected XGBoost as the classifier. Extreme Gradient Boosting, or XGBoost, is an ensemble learning algorithm. Similar to AdaBoost, the idea behind XGBoost is that the algorithm uses decision trees (or other weak learners) as its base models to make predictions. It involves building a sequence of models, each of which focuses on correcting the errors from the previous one.

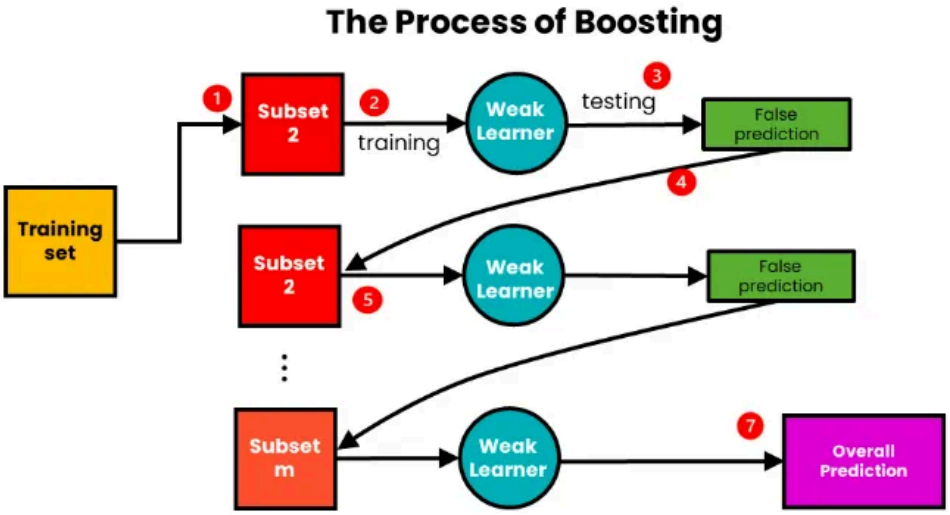


Figure 13.1.1. How Boosting works (Subha, 2024)

The main difference between AdaBoost (the model used in Question 4) and XGBoost is that AdaBoost focuses on re-weighting samples, whereas XGBoost uses gradient descent to iteratively fit new weak learners that correct the residual errors of the previous ones.

2. Implementation

XGBoost model is implemented by using the package `xgboost` (https://cran.r-project.org/web/packages/xgboost/index.html). Initially, the model is trained with the full set of features, as I want to investigate the importance ranking provided by the package later.

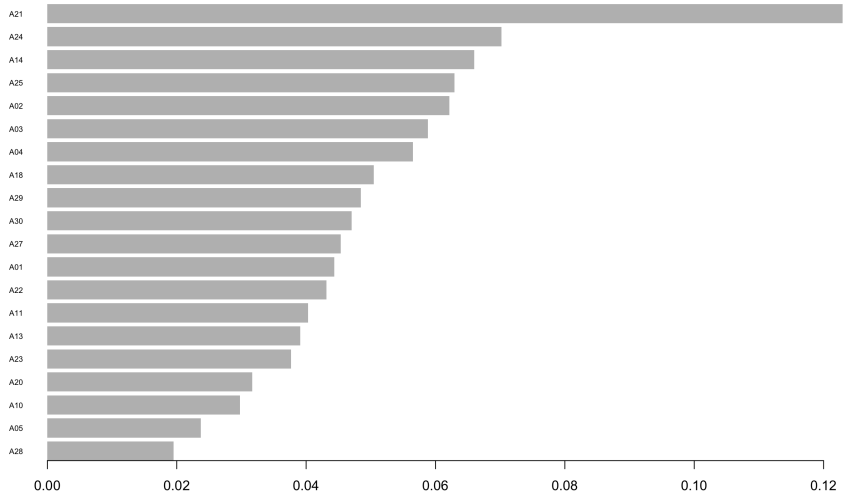


Figure 13.2.1. Importance ranked by the XGBoost model

To indicate which is the best set of features, I build 20 versions of XGBoost, each using an incremental set of top-ranked features, starting from the top 1, top 2, top 3, up to the top 20 features based on the importance ranking above.

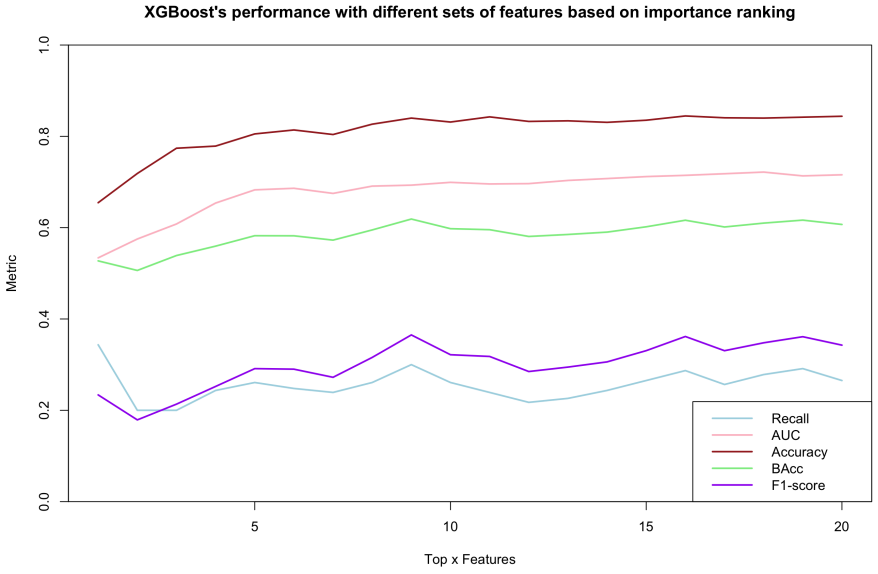


Figure 13.2.2. XGBoost's performance with different sets of features based on importance ranking

As inferred from the graph, it is evident that the model with the top 9 features has one of the best performances. It ranks 1st in terms of F1-score and BAcc, while 2nd regarding Accuracy and Recall.
Therefore, I chose this model as the final one.

- Confusion matrix

	Predicted: Other (0)	**Predicted: Oats (1)**
Actual: Other (0)	1191	79
Actual: Oats (1)	161	69

Figure 13.2.3. XGBoost's confusion matrix

- Performance metrics

Precision	Recall	Accuracy	F1-score	AUC	BAcc
0.46621622	0.3	0.84	0.36507937	0.6153372	0.61889764

Figure 13.2.4. XGBoost's performance metrics

3. Comparison

Metrics	Decision Tree	Naive Bayes	Bagging	Boosting	Random Forest	Balanced Random Forest	ANN	XGBoost
Precision	N/A	0.28729282	1	0.46391753	0.58333333	0.35809019	0.2314050	0.46621622
Recall	0	0.22608696	0.00434783	0.19565217	0.03043478	0.58695652	0.7304348	0.3
Accuracy	0.8466667	0.79533333	0.84733333	0.842	0.848	0.77533333	0.5866667	0.84
F1-score	0	0.25304136	0.00865801	0.27522936	0.05785124	0.44481054	0.3514644	0.36507937
AUC	0.6543958	0.6778637	0.7213745	0.7245395	0.7403612	0.7413523	0.6482746	0.6931085
BAcc	0.5	0.5622561	0.50434783	0.5957378	0.51246149	0.6982027	0.6073947	0.61889764

Figure 13.3.1. Performance metrics of all classifiers

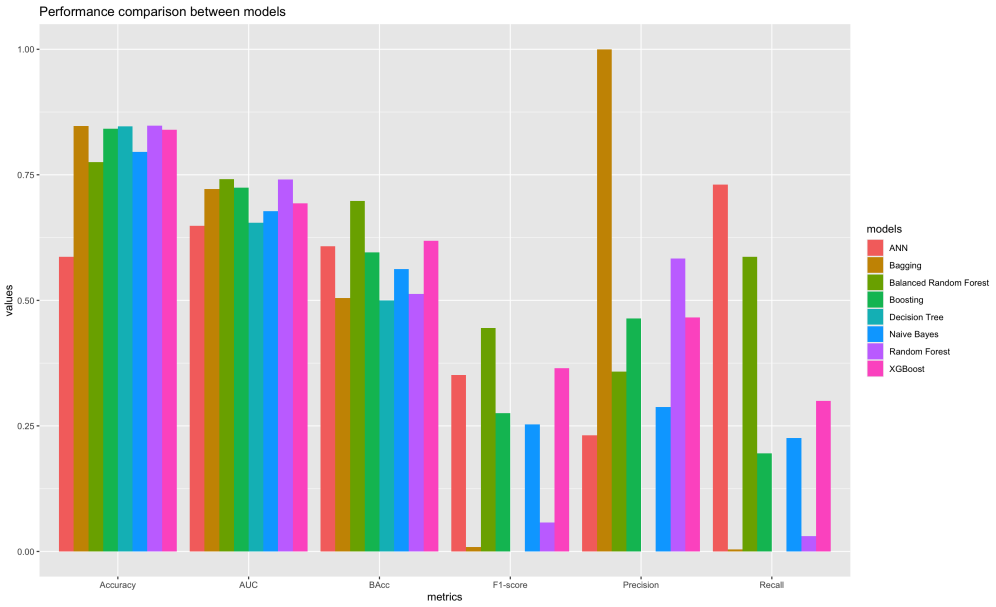


Figure 13.3.2. Performance metrics of all classifiers

As inferred from the table above, the XGBoost model has the 2nd highest precision, indicating its superior performance in limiting false positives. Regarding AUC and Recall, the metrics show XGBoost's weaknesses in catching positives and distinguishing between positives and negatives. However, its BAcc still ranks second-highest, showing its relatively good performance for both classes. Therefore, it can be concluded that while XGBoost does not seem to be the best choice for tasks that prioritise detecting positive cases, it is still a strong candidate when we aim to minimise false positives.

References.

Chen, C., & Liaw, A. (2004). _Using Random Forest to Learn Imbalanced Data_. https://statistics.berkeley.edu/sites/default/files/tech-reports/666.pdf

Chen, R.-C., Dewi, C., Huang, S.-W., & Caraka, R. E. (2020). Selecting critical features for data classification based on machine learning methods. _Journal of Big Data_, _7_(1). https://doi.org/10.1186/s40537-020-00327-4

Complement Naive Bayes (CNB) Algorithm. (2020, July 23). GeeksforGeeks. https://www.geeksforgeeks.org/complement-naive-bayes-cnb-algorithm/

Fox, E. W., Hill, R. A., Leibowitz, S. G., Olsen, A. R., Thornbrugh, D. J., & Weber, M. H. (2017). Assessing the accuracy and stability of variable selection methods for random forest modeling in ecology. _Environmental Monitoring and Assessment_, _189_(7).

https://doi.org/10.1007/s10661-017-6025-0

GeeksforGeeks. (2024a, March 6). _Gradient Boosting vs Random Forest_. GeeksforGeeks. https://www.geeksforgeeks.org/gradient-boosting-vs-random-forest/

GeeksforGeeks. (2024b, March 11). _Bagging and Random Forest for Imbalanced Classification_. GeeksforGeeks. https://www.geeksforgeeks.org/bagging-and-random-forest-for-imbalanced-classification/

GeeksforGeeks. (2024c, May 28). _Feature Selection Using Random Forest_. GeeksforGeeks. https://www.geeksforgeeks.org/feature-selection-using-random-forest/

GeeksforGeeks. (2024d, November 25). _Training a decision tree against unbalanced data_. GeeksforGeeks. https://www.geeksforgeeks.org/training-a-decision-tree-against-unbalanced-data/

Kim, T., & Lee, J.-S. (2023). Maximizing AUC to learn weighted naive Bayes for imbalanced data classification. _Expert Systems with Applications_, _217_, 119564. https://doi.org/10.1016/j.eswa.2023.119564

Olamendy, J. C. (2024, March 8). _Tackling the Challenge of Imbalanced Datasets: A Comprehensive Guide_. Medium. https://medium.com/@juanc.olamendy/tackling-the-challenge-of-imbalanced-datasets-a-comprehensive-guide-2feb11ca2fa0

Reed, V. (2024, October 8). _AdaBoost: Solving Class Imbalance In Datasets Effectively_. AICompetence. https://aicompetence.org/adaboost-solving-class-imbalance-in-datasets/

Shahri, N. H. N. B. M., Lai, S. B. S., Mohamad, M. B., Rahman, H. A. B. A., & Rambli, A. B. (2021). Comparing the Performance of AdaBoost, XGBoost, and Logistic Regression for Imbalanced Data. _Mathematics and Statistics_, _9_(3), 379–385. https://doi.org/10.13189/ms.2021.090320

Subha. (2024, March 29). _Boosting — Adaboost, Gradient Boost and XGBoost - Subha - Medium_. Medium. https://medium.com/@pingsubhak/boosting-adaboost-gradient-boost-and-xgboost-bdda87eed44e

Thölke, P., Mantilla-Ramos, Y.-J., Abdelhedi, H., Maschke, C., Dehgan, A., Harel, Y., Kemtur, A., Mekki Berrada, L., Sahraoui, M., Young, T., Bellemare Pépin, A., El Khantour, C., Landry, M., Pascarella, A., Hadid, V., Combrisson, E., O'Byrne, J., & Jerbi, K. (2023). Class imbalance should not throw you off balance: Choosing the right classifiers and performance metrics for brain decoding with imbalanced data. _NeuroImage_, _277_, 120253. https://doi.org/10.1016/j.neuroimage.2023.120253

Appendix.

Question 1.

1. Proportion of "Oats" to "Others"

```
oats_count = length(WD["Class"][WD["Class"] == 1])
other_count = length(WD["Class"][WD["Class"] == 0])
oats_to_other = oats_count / other_count
```

2. Summary table of real-valued independent variables

```
write.csv(summary(WD), "wd.csv")
```

As the data in the `csv` file is not in an appropriate format, manual formatting is performed. The table, eventually, is similar to the **Table 1** (without the SD column).

Question 2.

1. Converting to `factor`

```
WD$Class = as.factor(WD$Class)
```

Question 3.

1. Split dataset into a 70% training and a 30% test set

```
set.seed([redacted])
train.row = sample(1:nrow(WD), 0.7*nrow(WD))
WD.train = WD[train.row,]
WD.test = WD[-train.row,]
```

Question 4.

1. Decision Tree

- Implementation

```
library(tree)
WD.tree = tree(Class ~., data = WD.train)
```

- Plot the model

```
plot(WD.tree)
text(WD.tree, pretty = 0)
```

2. Naive Bayes

- Implementation

```
library(e1071)
WD.bayes = naiveBayes(Class ~., data = WD.train)
```

3. Bagging

- Implementation

```
library(adabag)
WD.bag = bagging(Class ~., data = WD.train)
```

4. Boosting

- Implementation

```
WD.boost = boosting(Class ~., data = WD.train)
```

5. Random Forest

- Implementation

```
library(randomForest)
WD.rf = randomForest(Class ~., data = WD.train)
```

Question 5.

1. Decision Tree

- Classify test cases using the test data

```
WD.predtree = predict(WD.tree, WD.test, type = "class")
```

- Confusion matrix

```
t1=table(Actual_Class = WD.test$Class, Predicted_Class = WD.predtree)
```

- Perfomance metrics

```
performance_metrics = function(TP, FN, FP, TN) {
  recall = TP/(TP+FN)
  precision = TP/(TP+FP)
  accuracy = (TP+TN)/(TP+TN+FP+FN)
  if (is.na(precision) | is.na(recall) | precision + recall == 0) {
    f1_score = 0
  }
  else {
    f1_score = (2*precision*recall)/(precision + recall)
  }
  print(paste("Recall: ", round(recall, 8)))
  print(paste("Precision: ", round(precision, 8)))
```

```
  print(paste("Accuracy: ", round(accuracy, 8)))
  print(paste("F1-score: ", round(f1_score, 8)))
}
performance_metrics(TP=t1["1","1"], FN=t1["1","0"], FP=t1["0","1"], TN=t1["0","0"])
```

2. Naive Bayes

- Classify test cases using the test data

```
WD.predbayes = predict(WD.bayes, WD.test)
```

- Confusion matrix

```
t2 = table(Actual_Class = WD.test$Class, Predicted_Class = WD.predbayes)
```

- Performance metrics

```
performance_metrics(TP=t2["1","1"], FN=t2["1","0"], FP=t2["0","1"], TN=t2["0","0"])
```

3. Bagging

- Classify test cases using the test data

```
WD.predbag = predict.bagging(WD.bag, WD.test)
```

- Confusion matrix

```
t3 = table(Actual_Class = WD.test$Class, Predicted_Class = WD.predbag$class)
```

- Performance metrics

```
performance_metrics(TP=t3["1","1"], FN=t3["1","0"], FP=t3["0","1"], TN=t3["0","0"])
```

4. Boosting

- Classify test cases using the test data

```
WD.predboost = predict.boosting(WD.boost, WD.test)
```

- Confusion matrix

```
t4 = table(Actual_Class = WD.test$Class, Predicted_Class = WD.predboost$class)
```

- Performance metrics

```
performance_metrics(TP=t4["1","1"], FN=t4["1","0"], FP=t4["0","1"], TN=t4["0","0"])
```

5. Random Forest

- Classify test cases using the test data

```
WD.predrf = predict(WD.rf, WD.test)
```

- Confusion matrix

```
t5 = table(Actual_Class = WD.test$Class, Predicted_Class = WD.predrf)
```

- Performance metrics

```
performance_metrics(TP=t5["1","1"], FN=t5["1","0"], FP=t5["0","1"], TN=t5["0","0"])
```

Question 6.

- ROC

```
library(ROCR)
# Decision Tree
WD.pred.tree = predict(WD.tree, WD.test, type="vector")
WDD.pred = prediction(WD.pred.tree[,2], WD.test$Class)
WDD.perf = performance(WDD.pred, "tpr", "fpr")
plot(WDD.perf, col = "lightblue")
abline(0,1, col="red")
# Naive Bayes
WD.pred.bayes = predict(WD.bayes, WD.test, type="raw")
WDN.pred = prediction(WD.pred.bayes[,2], WD.test$Class)
WDN.perf = performance(WDN.pred, "tpr", "fpr")
plot(WDN.perf, add=TRUE, col="orange")
# Bagging
WDBag.pred = prediction(WD.predbag$prob[,2], WD.test$Class)
WDBag.perf = performance(WDBag.pred, "tpr", "fpr")
plot(WDBag.perf, add=TRUE, col="lightgreen")
# Boosting
WDBoost.pred = prediction(WD.predboost$prob[,2], WD.test$Class)
WDBoost.perf = performance(WDBoost.pred, "tpr", "fpr")
plot(WDBoost.perf, add=TRUE, col="brown")
# Random Forest
WD.pred.rf = predict(WD.rf, WD.test, type="prob")
WDR.pred = prediction(WD.pred.rf[,2], WD.test$Class)
WDR.perf = performance(WDR.pred, "tpr", "fpr")
plot(WDR.perf, add=TRUE, col="darkgreen")
legend("bottomright",
       legend=c("Decision Tree", "Naive Bayes", "Bagging", "Boosting", "Random Forest"),
       col = c("lightblue", "orange", "lightgreen", "brown", "darkgreen"),
       lwd = 1)
```

- AUC

```
print(as.numeric(performance(WDD.pred, "auc")@y.values)) # Decision Tree
print(as.numeric(performance(WDN.pred, "auc")@y.values)) # Naive Bayes
```

```
print(as.numeric(performance(WDBag.pred, "auc")@y.values)) # Bagging
print(as.numeric(performance(WDBoost.pred, "auc")@y.values)) # Boosting
print(as.numeric(performance(WDR.pred, "auc")@y.values)) # Random Forest
```

Question 7.

- Plot

```
question7.metrics = read.csv("question7.csv")
metrics = c(rep("Recall", 5), rep("Precision", 5), rep("Accuracy", 5), rep("F1-score", 5), rep("AUC", 5))
models = rep(c("Decision Tree", "Naive Bayes", "Bagging", "Boosting", "Random Forest"), 5)
values = c()
for (metric in list("Recall", "Precision", "Accuracy", "F1-score", "AUC")) {
  for (i in 2:6) {
    values = c(values, question7.metrics[question7.metrics$Metrics == metric,i])
  }
}

data = data.frame(models,metrics,values)

ggplot(data, aes(fill=models, y=values, x=metrics)) +
  geom_bar(position="dodge", stat="identity") +
  labs(title = "Performance comparison between models")
```

Question 8.

```
print(summary(WD.tree))
print(WD.bayes)
write.csv(WD.bag$importance, "bagging_importance.csv")
write.csv(WD.boost$importance, "boosting_importance.csv")
write.csv(WD.rf$importance, "rf_importance.csv")
```

```
# Bagging
barplot(
  sort(WD.bag$importance, decreasing = TRUE),
  xlab = "Feature",
  ylab = "Importance Score",
  main = "Importance Score of Features by Bagging"
)
# Boosting
barplot(
  sort(WD.boost$importance, decreasing = TRUE),
  xlab = "Feature",
  ylab = "Importance Score",
  main = "Importance Score of Features by Boosting"
)
# Random Forest
barplot(
  sort(WD.rf.importance$MeanDecreaseGini, decreasing = TRUE),
  names.arg = WD.rf.importance$col,
  xlab = "Feature",
  ylab = "Importance Score",
  main = "Importance Score of Features by Random Forest"
)
```

Question 9.

No coding was performed in this task.

Question 10.

- Create a copy of the `WD` dataset and choose randomly 2000 rows of observations with class 0

```
WD.fs = WD[,]
WD.fs.zero = WD.fs[WD.fs$Class == 0,]
set.seed([redacted])
WD.fs = rbind(
  WD.fs.zero[sample(nrow(WD.fs.zero),2000,replace=FALSE),],
  WD.fs[WD.fs$Class == 1,]
)
```

- Split the dataset into a training set and a test set

```
set.seed([redacted])
train.row.hand = sample(1:nrow(WD.fs), 0.7*nrow(WD.fs))

WD.fs.train = WD.fs[train.row,]
WD.fs.test = WD.fs[-train.row,]
```

- Create the model with the chosen features, visualise it, and use the test dataset to calculate the performance metrics

```
# Create the model
WD.tree.hand = tree(Class ~ A25 + A24 + A02 + A21 + A03, data = WD.fs.train)
WD.tree.hand

# Visualise
plot(WD.tree.hand)
text(WD.tree.hand, pretty = 0)

# Predict using the test dataset
WD.predtree.hand = predict(WD.tree.hand, WD.fs.test, type = "class")

# Create confusion matrix
t6=table(Actual_Class = WD.fs.test$Class, Predicted_Class = WD.predtree.hand)
cat("\n#Decision Tree Confusion\n")
print(t6)
performance_metrics(TP=t6["1","1"], FN=t6["1","0"], FP=t6["0","1"], TN=t6["0","0"])

# Calculate AUC
WDD.hand.pred.tree = predict(WD.tree.hand, WD.fs.test, type="vector")
WDD.hand.pred = prediction(WDD.hand.pred.tree[,2], WD.fs.test$Class)
print(as.numeric(performance(WDD.hand.pred, "auc")@y.values))
```

- ROC

```
library(ROCR)
# Decision Tree
WD.pred.tree = predict(WD.tree, WD.test, type="vector")
WDD.pred = prediction(WD.pred.tree[,2], WD.test$Class)
```

```
WDD.perf = performance(WDD.pred, "tpr", "fpr")
plot(WDD.perf, col = "lightblue")
abline(0,1, col="red")
# Naive Bayes
WD.pred.bayes = predict(WD.bayes, WD.test, type="raw")
WDN.pred = prediction(WD.pred.bayes[,2], WD.test$Class)
WDN.perf = performance(WDN.pred, "tpr", "fpr")
plot(WDN.perf, add=TRUE, col="orange")
# Bagging
WDBag.pred = prediction(WD.predbag$prob[,2], WD.test$Class)
WDBag.perf = performance(WDBag.pred, "tpr", "fpr")
plot(WDBag.perf, add=TRUE, col="lightgreen")
# Boosting
WDBoost.pred = prediction(WD.predboost$prob[,2], WD.test$Class)
WDBoost.perf = performance(WDBoost.pred, "tpr", "fpr")
plot(WDBoost.perf, add=TRUE, col="brown")
# Random Forest
WD.pred.rf = predict(WD.rf, WD.test, type="prob")
WDR.pred = prediction(WD.pred.rf[,2], WD.test$Class)
WDR.perf = performance(WDR.pred, "tpr", "fpr")
plot(WDR.perf, add=TRUE, col="darkgreen")
# Decision Tree (Q10)
WDD.hand.perf = performance(WDD.hand.pred, "tpr", "fpr")
plot(WDD.hand.perf, add=TRUE, col="black")
legend("bottomright",
       legend=c("Decision Tree", "Naive Bayes", "Bagging", "Boosting", "Random Forest", "Decision Tree
(Q10)"),
       col = c("lightblue", "orange", "lightgreen", "brown", "darkgreen", "black"),
       lwd = 1)
```

Question 11.

- Implementation

```
set.seed([redacted])
WD.brf = randomForest(
  Class ~.,
  data = WD.train,
  sampsize = c(
    "0" = nrow(WD.train[WD.train$Class == 1,]),
    "1" = nrow(WD.train[WD.train$Class == 1,])
    )
)
```

- Classify test cases using the test data

```
WD.predbrf = predict(WD.brf, WD.test)
```

- Confusion matrix

```
t7 = table(Actual_Class = WD.test$Class, Predicted_Class = WD.predbrf)
```

- Performance metrics

```
performance_metrics(TP=t7["1","1"], FN=t7["1","0"], FP=t7["0","1"], TN=t7["0","0"])
```

- AUC

```
WD.pred.brf = predict(WD.brf, WD.test, type="prob")
WD.pred.brf = prediction(WD.pred.brf[,2], WD.test$Class)
print(as.numeric(performance(WD.pred.brf, "auc")@y.values))
```

- Barplot

```
# Collect metrics from all models
WD.metrics.bacc = c()
WD.metrics.f1_score = c()
WD.metrics.recall = c()
WD.metrics.auc = c(0.6543958, 0.6778637, 0.7213745, 0.7245395, 0.7403612, 0.7413523)

for (i in list(t1, t2, t3, t4, t5, t7)) {
  metrics = performance_metrics(TP=i["1","1"], FN=i["1","0"], FP=i["0","1"], TN=i["0","0"])
  WD.metrics.bacc = append(WD.metrics.bacc, metrics["bacc"])
  WD.metrics.f1_score = append(WD.metrics.f1_score, metrics["f1_score"])
  WD.metrics.recall = append(WD.metrics.recall, metrics["recall"])
}

# AUC
barplot(
  WD.metrics.auc,
  names.arg = c("Decision Tree", "Naive Bayes", "Bagging", "Boosting", "Random Forest", "Balanced Random
Forest"),
  col = "pink",
  main = "Model performance by AUC",
  ylab = "AUC"
  )

# F1-score
barplot(
  WD.metrics.f1_score,
  names.arg = c("Decision Tree", "Naive Bayes", "Bagging", "Boosting", "Random Forest", "Balanced Random
Forest"),
  col = "pink",
  main = "Model performance by F1-score",
  ylab = "F1-score"
  )

# Recall
barplot(
  WD.metrics.recall,
  names.arg = c("Decision Tree", "Naive Bayes", "Bagging", "Boosting", "Random Forest", "Balanced Random
Forest"),
  col = "pink",
  main = "Model performance by Recall",
  ylab = "Recall"
  )

# BAcc
barplot(
```

```
  WD.metrics.bacc,
  names.arg = c("Decision Tree", "Naive Bayes", "Bagging", "Boosting", "Random Forest", "Balanced Random
Forest"),
  col = "pink",
  main = "Model performance by BAcc",
  ylab = "BAcc"
  )
```

Question 12.

1. Model 1

- Pre-processing

```
# Scale data
min_max_scaling = function(col) {
  (col - min(col)) / (max(col) - min(col))
}

WD.ann.x.train = as.matrix(sapply(WD.train[,c("A25", "A24", "A02", "A21", "A03", "A14", "A18", "A04", "A29",
"A27")], min_max_scaling))
WD.ann.x.test = as.matrix(sapply(WD.test[,c("A25", "A24", "A02", "A21", "A03", "A14", "A18", "A04", "A29",
"A27")], min_max_scaling))

# Convert labels to numeric
WD.ann.y.train = ifelse(WD.train$Class == "0", 0, 1)
WD.ann.y.test = ifelse(WD.test$Class == "0", 0, 1)
```

- Model implementation

```
WD.ann = keras_model_sequential()
WD.ann %>%
  layer_dense(units = 256, activation = 'relu', input_shape = c(10)) %>%
  layer_dropout(rate = 0.4) %>%
  layer_dense(units = 128, activation = 'relu') %>%
  layer_dropout(rate = 0.3) %>%
  layer_dense(units = 1, activation = 'sigmoid')

WD.ann %>% compile(
  loss = 'binary_crossentropy',
  optimizer = "adam",
  metrics = c('accuracy')
)

WD.ann %>% fit(
  WD.ann.x.train, WD.ann.y.train,
  epochs = 100,
  batch_size = 16,
  validation_split = 0.3,
  verbose=1,
  class_weight = list("0" = 1, "1" = 6)
)
```

- Use test data to predict and evaluate performance

```
keraspred = WD.ann %>% predict(WD.ann.x.test)
keraspred
```

- Confusion matrix

```
WD.ann.table1 = table(Actual = WD.ann.y.test, Predicted = ifelse(keraspred >= 0.5, 1, 0))
WD.ann.table1
```

- Performance metrics

```
performance_metrics(TP=WD.ann.table1["1","1"], FN=WD.ann.table1["1","0"], FP=WD.ann.table1["0","1"],
TN=WD.ann.table1["0","0"])
```

2. Model 2

- Pre-processing

```
# Scale data
min_max_scaling = function(col) {
  (col - min(col)) / (max(col) - min(col))
}

WD.ann.x.train = as.matrix(sapply(WD.train[,1:20], min_max_scaling))
WD.ann.x.test = as.matrix(sapply(WD.test[,1:20], min_max_scaling))

# Convert labels to numeric
WD.ann.y.train = ifelse(WD.train$Class == "0", 0, 1)
WD.ann.y.test = ifelse(WD.test$Class == "0", 0, 1)
```

- Model implementation

```
WD.ann = keras_model_sequential()
WD.ann %>%
  layer_dense(units = 256, activation = 'relu', input_shape = c(20)) %>%
  layer_dropout(rate = 0.4) %>%
  layer_dense(units = 128, activation = 'relu') %>%
  layer_dropout(rate = 0.3) %>%
  layer_dense(units = 1, activation = 'sigmoid')

WD.ann %>% compile(
  loss = 'binary_crossentropy',
  optimizer = "adam",
  metrics = c('accuracy')
)

WD.ann %>% fit(
  WD.ann.x.train, WD.ann.y.train,
  epochs = 100,
  batch_size = 16,
  validation_split = 0.3,
  verbose=1,
```

```
  class_weight = list("0" = 1, "1" = 6)
)
```

- Use test data to predict and evaluate performance

```
keraspred = WD.ann %>% predict(WD.ann.x.test)
keraspred
```

- Confusion matrix

```
WD.ann.table2 = table(Actual = WD.ann.y.test, Predicted = ifelse(keraspred ⩾ 0.5, 1, 0))
WD.ann.table2
```

- Performance metrics

```
performance_metrics(TP=WD.ann.table2["1","1"], FN=WD.ann.table2["1","0"], FP=WD.ann.table2["0","1"],
TN=WD.ann.table2["0","0"])
```

3. Model comparison (Model 1 vs. Model 2)

- Plot

```
metrics = c(rep("Recall", 2), rep("Precision", 2), rep("Accuracy", 2), rep("F1-score", 2), rep("BAcc", 2),
rep("AUC", 2))
models = rep(c("Top 10", "Full"), 6)
values = c(WD.ann.metrics1["recall"],
           WD.ann.metrics2["recall"],
           WD.ann.metrics1["precision"],
           WD.ann.metrics2["precision"],
           WD.ann.metrics1["accuracy"],
           WD.ann.metrics2["accuracy"],
           WD.ann.metrics1["f1_score"],
           WD.ann.metrics2["f1_score"],
           WD.ann.metrics1["bacc"],
           WD.ann.metrics2["bacc"],
           WD.ann.auc1,
           WD.ann.auc2)
data = data.frame(models,metrics,values)

ggplot(data, aes(fill=models, y=values, x=metrics)) +
  geom_bar(position="dodge", stat="identity") +
  labs(title = "Performance comparison between two models")
```

Question 13.

- Pre-processing

```
# Create a new dataset
WD.xgboost = WD[,]
# Convert Class from factor to numeric
WD.xgboost$Class = ifelse(WD.xgboost$Class == "0", 0, 1)
```

```
# Split train and test sets
WD.xgboost.train = WD.xgboost[train.row,]
WD.xgboost.test = WD.xgboost[-train.row,]
```

- Implementation

```
WD.xgb = xgboost(data = data.matrix(WD.xgboost.train[,1:20]),
                 objective = "binary:logistic",
                 eval_metric = "auc",
                 nrounds = 25,
                 label = WD.xgboost.train$Class,
                 max_depth = 15,
                 scale_pos_weight = nrow(WD.xgboost.train[WD.xgboost.train$Class ==
0,])/nrow(WD.xgboost.train[WD.xgboost.train$Class == 1,]))
```

- Use the test set to predict

```
WD.xgb.pred = predict(WD.xgb, data.matrix(WD.xgboost.test[,1:20]))
WD.xgb.pred = ifelse(WD.xgb.pred ⩾ 0.5, 1, 0)
```

- Confusion matrix

```
t11 = table(Actual_Class = WD.xgboost.test$Class, Predicted_Class = WD.xgb.pred)
```

- Performance metrics

```
performance_metrics(TP=t11["1","1"], FN=t11["1","0"], FP=t11["0","1"], TN=t11["0","0"])
```

- Importance ranking plot

```
# Create an importance matrix
importance_matrix = xgb.importance(colnames(WD.xgboost.train[1:20]), model = WD.xgb)
# Plot the importance matrix
xgb.plot.importance(importance_matrix)
```

- Feature set selection

```
# Create vectors to store performance metrics
WD.xgb.recall = c()
WD.xgb.auc = c()
WD.xgb.accuracy = c()
WD.xgb.bacc = c()
WD.xgb.f1_score = c()

# Iteratively choose the top x features and train a model
for (i in 1:20) {
  WD.xgb = xgboost(data = data.matrix(WD.xgboost.train[,importance_matrix[1:i,]$Feature]),
                   objective = "binary:logistic",
                   eval_metric = "auc",
                   nrounds = 25,
```

```
                  label = WD.xgboost.train$Class,
                  max_depth = 15,
                  scale_pos_weight = nrow(WD.xgboost.train[WD.xgboost.train$Class ==
0,])/nrow(WD.xgboost.train[WD.xgboost.train$Class == 1,]))
  WD.xgb.pred = predict(WD.xgb, data.matrix(WD.xgboost.test[,importance_matrix[1:i,]$Feature]))
  t11 = table(Actual_Class = WD.xgboost.test$Class, Predicted_Class = ifelse(WD.xgb.pred >= 0.5, 1, 0))
  metrics = performance_metrics(TP=t11["1","1"], FN=t11["1","0"], FP=t11["0","1"], TN=t11["0","0"])
  xgb.pred = ROCR::prediction(WD.xgb.pred, WD.xgboost.test$Class)
  auc = as.numeric(performance(xgb.pred, "auc")@y.values)
  WD.xgb.recall = append(WD.xgb.recall, metrics["recall"])
  WD.xgb.auc = append(WD.xgb.auc, auc)
  WD.xgb.accuracy = append(WD.xgb.accuracy, metrics["accuracy"])
  WD.xgb.bacc = append(WD.xgb.bacc, metrics["bacc"])
  WD.xgb.f1_score = append(WD.xgb.f1_score, metrics["f1_score"])
}

# Plot the performance metrics for 20 models
plot(1:20, WD.xgb.recall, type = "l", col = "lightblue", ylim = c(0,1), ylab = "Metric", xlab = "Top x
Features", lwd = 2)
lines(1:20, WD.xgb.auc, col = "pink", lwd = 2)
lines(1:20, WD.xgb.accuracy, col = "brown", lwd = 2)
lines(1:20, WD.xgb.bacc, col = "lightgreen", lwd = 2)
lines(1:20, WD.xgb.f1_score, col = "purple", lwd = 2)
title(main = "XGBoost's performance with different sets of features based on importance ranking")
legend("bottomright",
       legend=c("Recall", "AUC", "Accuracy", "BAcc", "F1-score"),
       col = c("lightblue", "pink", "brown", "lightgreen", "purple"),
       lwd = 2)
```

- Final model implementation

```
# Model
WD.xgb = xgboost(data = data.matrix(WD.xgboost.train[,importance_matrix[1:9,]$Feature]),
                 objective = "binary:logistic",
                 eval_metric = "auc",
                 nrounds = 25,
                 label = WD.xgboost.train$Class,
                 max_depth = 15,
                 scale_pos_weight = nrow(WD.xgboost.train[WD.xgboost.train$Class ==
0,])/nrow(WD.xgboost.train[WD.xgboost.train$Class == 1,]))
# Use the test set to predict
WD.xgb.pred = predict(WD.xgb, data.matrix(WD.xgboost.test[,importance_matrix[1:9,]$Feature]))
# Confusion matrix
t11 = table(Actual_Class = WD.xgboost.test$Class, Predicted_Class = ifelse(WD.xgb.pred >= 0.5, 1, 0))
t11
# Performance metrics
metrics = performance_metrics(TP=t11["1","1"], FN=t11["1","0"], FP=t11["0","1"], TN=t11["0","0"])
```

- Plot

```
question13.metrics = read.csv("question13.csv")
metrics = c(rep("Recall", 8), rep("Precision", 8), rep("Accuracy", 8), rep("F1-score", 8), rep("AUC", 8),
rep("BAcc", 8))
models = rep(c("Decision Tree", "Naive Bayes", "Bagging", "Boosting", "Random Forest", "Balanced Random
Forest", "ANN", "XGBoost"), 6)
```

```
values = c()
for (metric in list("Recall", "Precision", "Accuracy", "F1-score", "AUC", "BAcc")) {
  for (i in 2:9) {
    values = c(values, question13.metrics[question13.metrics$Metrics == metric,i])
  }
}

data = data.frame(models,metrics,values)

ggplot(data, aes(fill=models, y=values, x=metrics)) +
  geom_bar(position="dodge", stat="identity") +
  labs(title = "Performance comparison between models")
```