Data Mining Assignment

COMP 3009

Nicholas Klvana-Hooper (19782944)

Semester 2 2021

Summary

Major Findings

Some of the major findings in the data given to us is that there is a significant imbalance between the two classes given to us. There is almost a 3 to 1 relationship between the 0 class and the 1 class.

Some other interesting finds were the distributions of the numerical values, with most of them being skewed to the right. There was also one attribute that was already distributed in a normal shape.

It was found during the classification phase that the best model to use on the dataset given to us is the kNN method with 7 neighbours, uniform weights and a p of 1. This resulted in the most accurate output for the predicted classes and also had a close to 50/50 relationship between the 0s and 1s of the classes outputted. This model also proved to be consistent enough with these results that it was chosen.

Lessons Learnt

During this assignment I learnt a lot about the workflow that data scientists go through with both classifying data but also techniques for preparing the data.

I learnt quite a lot of new things about class imbalancing and different ways of fixing this, whether through undersampling or oversampling.

I also spent a lot of time researching data transformation methods, learning how data can be distributed normally or even log-normally which means that it is still in a normal bell curve shape but skewing one way. In order to fix this sort of behaviour you have to use specific methods like box-cox that allow for this distribution to be forced and pulled back to being in the classic normal shape.

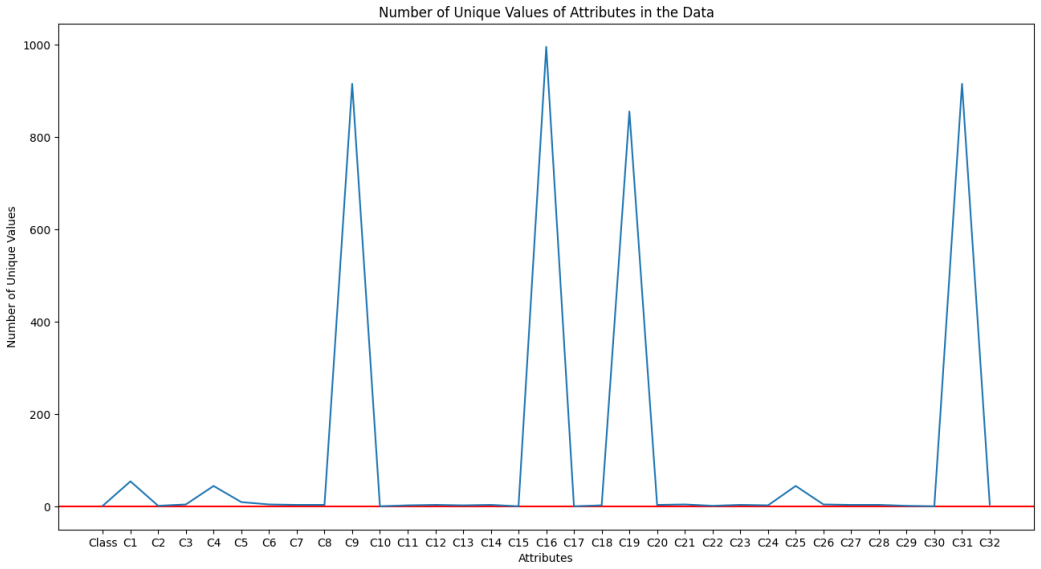
Data Preparation

Identify and remove irrelevant attributes

The first thing we can remove is the id, this is here purely for us to understand and shouldn't be used in our computations.

One other check for irrelevant attributes is that there are some columns that only have one unique attribute in their entire column. This is part of the `removeIrrelevant()` function and it outputs the following columns as only having one unique value:

```
C10 only has 1 values
C15 only has 1 values
C17 only has 1 values
C30 only has 1 values
```



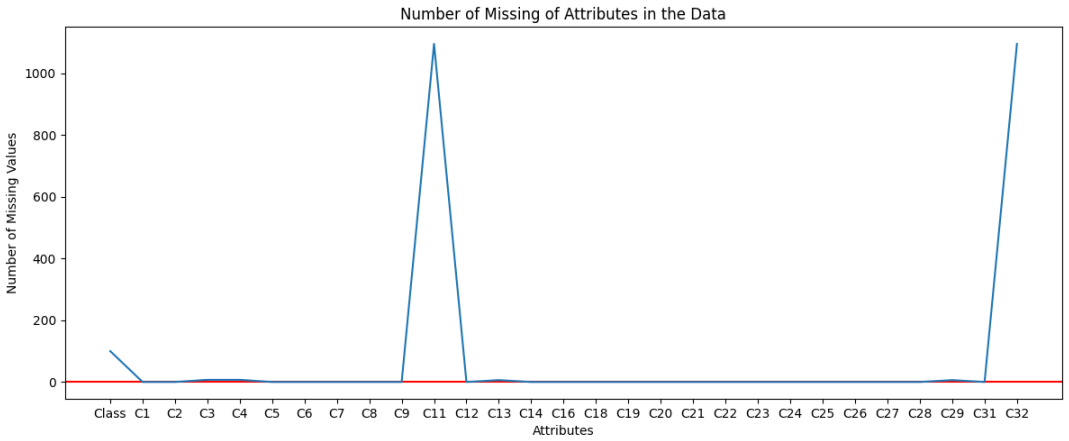
We can above this plotted across all of the attributes in the dataset, with the red line indicating those attributes that have a constant value throughout. This means that those columns can also be ignored as they are not useful for us; in my case I remove them from the dataset as they provide no value. Having attributes with a constant value will just make the classification take longer for calculations and doesn't provide the classification model with any useful material to distinguish between the two classes.

In total, at this step id, C10, C15, C17 and C30 have all been removed from the dataset.

Detecting and handling missing entries

There is a function in the main run file of this assignment that is called `missingEntries()`, you will have to give it a dataframe in order to look through. When given the whole set of data we will get the following output.

```
['Class 100', 'C3 7', 'C4 7', 'C11 1095', 'C13 6', 'C29 6', 'C32 1095']
```



The class attribute missing 100 values is expected as this is the data set that we are going to have to predict so this can be left in our dataset.

Attributes 'C11' and 'C32' only have a total of 5 values for each of these attributes. It could be assumed that this data is considered unnecessary data as it can't possibly be imputed properly and is removed. Keeping this in would provide no useful value to the classifier and trying to fill these in with a mean or mode as it would only take into account a very small amount of values.

The other attributes (C3, C4, C13 and C29) have the missing values filled in with the median or mode depending on their datatype. This is because they only have a small amount of values missing so imputing these with the mean and mode is fine as there is a large set of data to grab these values from.

Detects and handles duplicates (both instances and attributes)

There is a function called `duplicates()` which handles both duplicates in instance AND attributes. The first part of the output looks like the following:

```
C7 is same as C12
C9 is same as C31
C21 is same as C26
```

There are numerous duplicate instances, so you can see part of the output for this part of the function in the following snippet:

```
Duplicate row 900
Duplicate row 901
Duplicate row 902
Duplicate row 903
Duplicate row 904
Duplicate row 905
```

```
Duplicate row 906
Duplicate row 907
Duplicate row 908
Duplicate row 909
Duplicate row 910
Duplicate row 911
Duplicate row 912
Duplicate row 913
Duplicate row 914
Duplicate row 915
....
Duplicate row 985
Duplicate row 986
Duplicate row 987
Duplicate row 988
Duplicate row 989
Duplicate row 990
Duplicate row 991
Duplicate row 992
Duplicate row 993
Duplicate row 994
Duplicate row 995
Duplicate row 996
Duplicate row 997
Duplicate row 998
Duplicate row 999
```

You can see that this duplicate data is found from row 900 to row 999, I have just shown a snippet as to not take up more space in this report than is necessary.

Both the duplicated rows and duplicated attributes are removed from the dataset because they will provide an unwanted bias towards certain values during the classification stage.

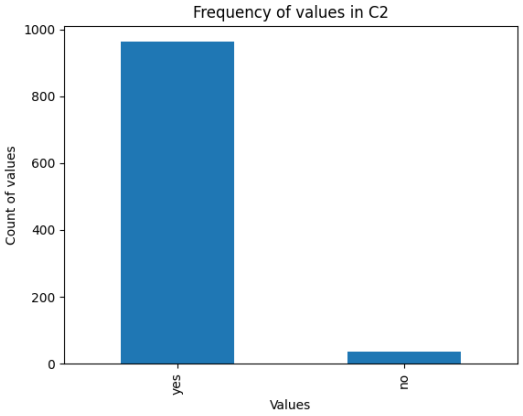
Select suitable data types for attributes

After doing the aforementioned pre-processing tasks our DataFrame has been reduced to only consist of 24 columns. If we run `df.columns` we get the following output on what attributes exist now

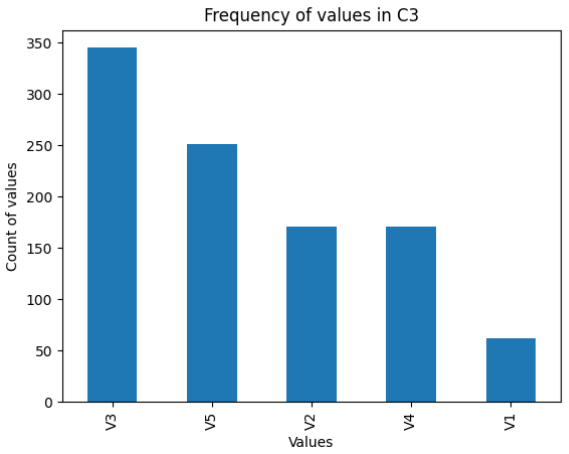
```
['Class', 'C1', 'C2', 'C3', 'C4', 'C5', 'C6', 'C8', 'C12', 'C13', 'C14',
'C16', 'C18', 'C19', 'C20', 'C22', 'C23', 'C24', 'C25', 'C26', 'C27','C28',
'C29', 'C31']
```

- Class
    - This one is our class attribute so it will just be a boolean
- C1
    - The data looks like numbers initially which are supported by the 55 distinct values that range from 18 to 75 - perhaps an age. This already is int64 so I can leave it.

- C2
    - The data in this column is either a ‘yes’ or a ‘no’ which is currently categorical. This would be better represented as a boolean so will convert it where ‘yes’ is True and ‘no’ is False.



- C3, C5, C6, C8, C12, C13, C14, C18, C22, C24, C26, C28
    - All of these columns had some number of values that had “V” followed by a number. For instance, attribute C3 had the values “V1”, “V2” up to “V5”. Since all of these are similar, just with differing numbers of values I’m grouping them here. These are originally objects so for sanity I just convert all of these to categorical values.



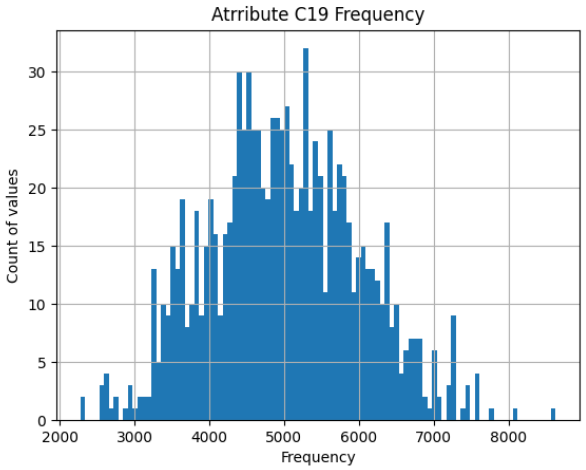
- C4
    - This data contains numbers. Looking at the data it has 45 distinct values with a range from 3 - 73 so this seems like it can stay numerical. It says float64 so just convert this back to int64.
- C16
    - These values all seem to be integers. It has a high variance and has a minimum of 1446 and a maximum of 220716. So can leave this as in int64 type.
- C19
    - This all seems to be integers. Also has high variance and a minimum of 2272 and a maximum of 8633 so can leave this as int64 in this case.
- C20, C23, C27
    - All of these have been grouped together as they have similar traits. They all have integers within a range, usually 1-4 that have no inbetween. Because of this I have converted these similar to the C3 lot, adding a “V” in front of the numbers and then making them categorical values.
- C25
    - These are integer values here as well, they have 45 or so distinct values with a range from 3 to 72. This will stay as int64.
- C29
    - This has floats currently with only 1.0 or 2.0 so can make this a boolean with 1.0 being True and 2.0 being False.

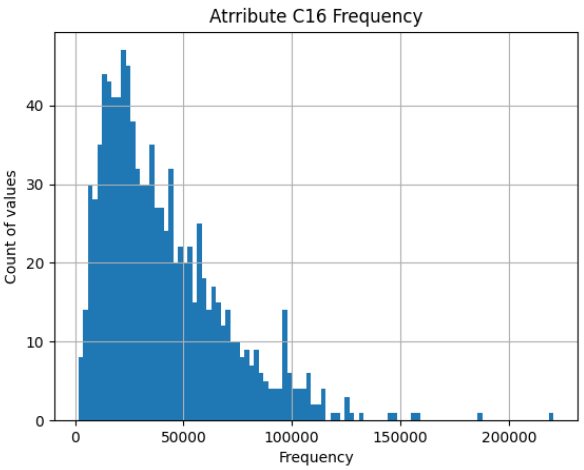
    - Due to the missing variables being filled in with the mean originally because it was numerical, there is an extra line here that rounds and applies true or false for those extra ones... does the same thing as doing mode just later on and differently.
- C31
    - Looks like integers with high variance and range from 249 to 18424, so this can stay as int64 type.

Perform data transformation

We want to investigate our numerical data to normalise it and ensure that it is in a state to perform classification on. There are six numerical attributes inside of our dataset by this point: C1, C4, C16, C19, C25 and C3.

Investigating the histogram for each of these attributes we can see C19 already looks close to the normal distribution, but all of the rest seem skewed to the right. Shown below are frequencies of certain values within the attributes C19 and C16, see the skewed nature of the C16 attribute.



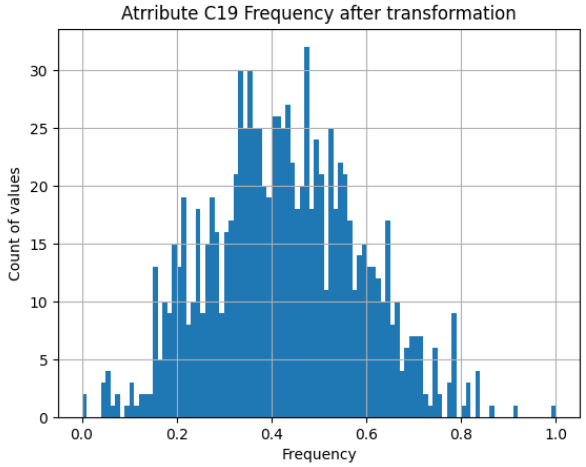


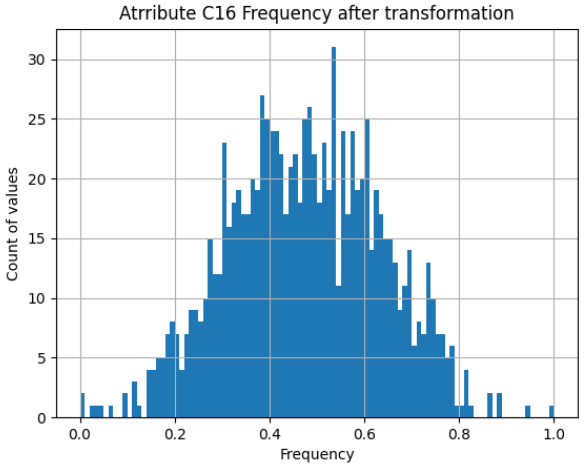
In order to rectify this, the attribute C19 was subjected to the sklearn StandardScaler function which standardises the data, in this case it was used because it doesn't play around too much with data that is skewed or has outliers.

The other attributes had the PowerTransformer function from sklearn used with the 'box-cox' method. The box-cox method is a method used in data science for any variables dependent on the dataset that are not normal, these are then transformed in such a way to approximate the normal curve as close as possible. This method only works on positive values, which is not an issue with any of the numerical values we have so this method was chosen. This is a function that deals with log-normal data such as the one above in C16 which is data that looks normal but skewed very much one way. This was used on all of the other attributes other than C19.

The last step was all of these features were put through the MinMaxScaler also from sklearn just to make sure that all of the resulting data is between the 0 and 1 range with no negatives.

You can see the effect this had on the C19 and C16 attributes below. Both are now looking very normalised in shape and all have values from 0 to 1.





This step is necessary to ensure all of the data is distributed the same so there is no bias to larger numbers in some attributes when it comes to classification. Having a large range for some values and smaller for the others will bias the larger numbers in a column. Distributing it normally and from 0 to 1 ensures all numeric values have the same shape to compare.

Feature Engineering

After hearing this term used throughout the semester it was a topic that I wanted to investigate for data preparation originally. However, after some research it quickly became apparent that this wasn't a task that I could complete for this assignment.

In order to do proper feature engineering you need to have a good understanding of the domain of the dataset. In other words, you need to know what the data is representing across all attributes to have a better understanding of what is relevant and what data types should be there.

In this assignment no knowledge of the dataset was passed to us, so this isn't a process that we can do. Doing feature engineering without some domain knowledge could end up being a detriment to the classification as we bias certain attributes and behaviours that could be completely wrong. So it is safer and in the best interest of the accuracy of the classification to not do this task.

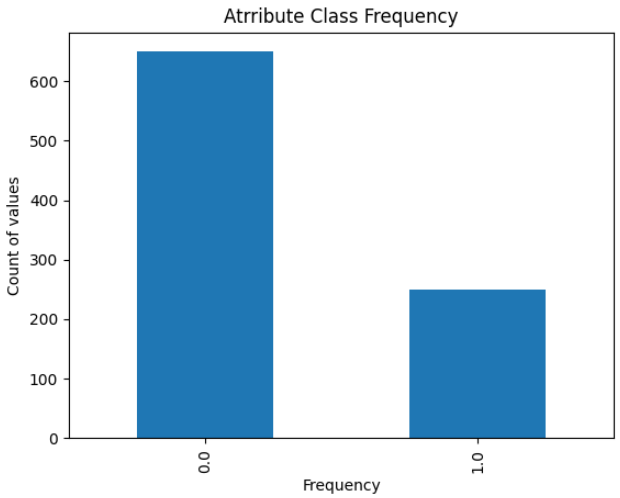
Data Classification

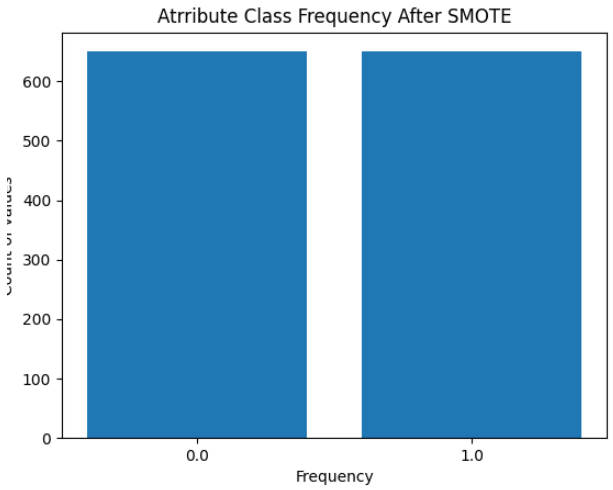
Class Imbalance

The dataset that was given to us for this assignment is unbalanced. There are 650 rows for the class ‘0’ and then 250 for ‘1’. The issue with this imbalance is that when come to classification, the model will figure out that it has a higher chance of getting a class correct if it guesses ‘0’ in this case, but we know that there are 50 occurrences of 0 and 1.

In order to fix this we need to do some form of sampling to make sure there is an equal amount of rows for each 0 and 1. To limit both to 250 in this case could create some issues with there not being enough data to train an accurate model, therefore I use an oversampling approach called SMOTE.

This approach creates “synthetic” - meaning not actually from the dataset - rows that were generated from the smaller class (in this case 1). It does this by picking a random value and then finding the nearest values, using a kNN-like style, to impose these fake rows in the data. This will result in a dataset that is completely balanced.





Model Training and Tuning

During the training stage I used a function I created called modelTuning to set out the range of potential parameters for each of the three models. This runs another function for each one that prints out statistics on each of the results as you’ll see later in this section. Each of the model functions were tested using GridSearchCV and trained on a training subset of 1040 rows. This prints out the best score received from GridSearchCV as well as the parameters and classifier that did the best.

I then also use this predicted best model to try to predict a set of 260 rows from the same subset that was kept aside. This allows another test to see how it fares on data it hasn’t seen. This will also print off how many 0s and 1s were guessed, in this case it doesn’t have to be equal as it's a random sample, but it's assumed that it would be fairly close to even.

kNN Model

I tested out three parameters for this model: n_neighbours, p and weight. The p parameter only has two values it can take, 1 or 2 which dictates what power value is used if the Minkowski distances. In terms of weights I look at both uniform and distance which dictate whether or not certain points have a higher importance to the distance than others.

The last value n_neighbours was the biggest trouble to tune. This dictates how many points around a point is considered in whether it is predicted as a class. Typically you use odd numbers for these so that there is no draw. Originally I had this checking from the values 1 all the way to 100, but after some balance issues with lower values I increased the minimum value for k.

```
Model score: 0.8048076923076923
Model parameters: {'n_neighbors': 7, 'p': 1, 'weights': 'distance'}
Model classifier: KNeighborsClassifier(n_neighbors=7, p=1,
weights='distance')
Accuracy on test set: 0.8192307692307692
0s: 96 1s: 164
Confusion matrix: [[ 84  12]
 [ 35 129]]
```

This is the output of the code for the parameters above. Two important parts of this output is that the n_neighbors gave us the minimum value it could've, and also the 0s and 1s are not balanced. Originally the neighbours was set to 1 but this caused a lot of issues with the balancing for the final prediction and as such this was raised to a level that produced more balanced and consistent results.

So despite the issues where the 0s and 1s are imbalanced **for the sampled test set** I chose these 3 parameters in my final model for this.

Naive Bayes

This model I looked at a parameter called var_smoothing. This parameter dictates a specific portion that a largest amount of variance in data can partake that is added to all the attributes in the dataset. In other words it kind of smooths the curve away from the mean for the distribution to account for rows of data that don't nicely fit.

This is a harder parameter, it is a float that has to be set but it's usually quite small. In order to create a fair and large amount of values to test on the curve I used numpy to generate 40 evenly spaced numbers between a log line from 0 to -8 decimal points.

```
Model score: 0.7105769230769232
Model parameters: {'var_smoothing': 0.03665241237079628}
Model classifier: GaussianNB(var_smoothing=0.03665241237079628)
Accuracy on test set: 0.7153846153846154
0s: 109 1s: 151
Confusion matrix: [[ 77  32]
```

```
 [ 42 109]]
```

In this case our result came out pretty consistently around this mark. It gives a good accuracy from both GridSearchCV and when predicting my test subset. You can also see this gives a semi-balanced result on the test dataset.

When compared to the final test this came out similarly balanced and was consistent so this value for var_smoothing was chosen for the assignment.

Decision Trees

For this model the same functions were used although different parameters were tried. For this one I only was looking at the 'min_samples_split' parameter. This is a number that tells the Decision Tree how the bare minimum total of values that has to occur before a split can occur on an internal node.

In this case I checked values from 2 to 15 as it can't be 1 and after trying a few other longer ranges this seemed like a more appropriate place to stop.

The following output was returned when checking these parameters:

```
Model score: 0.7423076923076923
Model parameters: {'min_samples_split': 5}
Model classifier: DecisionTreeClassifier(min_samples_split=5,
random_state=246)
Accuracy on test set: 0.7423076923076923
0s: 136 1s: 124
Confusion matrix: [[94 42]
 [25 99]]
```

As you can see the balance between the 0s and 1s for this were much more balanced than the previous kNN run through. This balance kept consistent all the way until the final prediction so I used the min_samples_split parameter as equal to 5.

Classifier Comparison

The classifier works by trying all three of the models that we created earlier and then running it over a cross validator with 10 folds. When testing each of the 3 models at this level, I checked for the accuracy, also the balance of the classes when predicting the final set of 100 rows as my two metrics.

Firstly the accuracy determines just the raw amount of rows that were correctly guessed per fold of the cross validation.

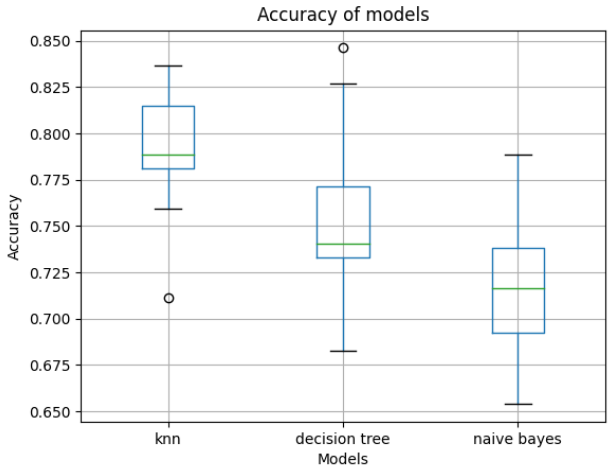
The ratio between the 0s and 1s per model was a metric I considered important as there was originally a large class imbalance so seeing how models handle the oversampled data.

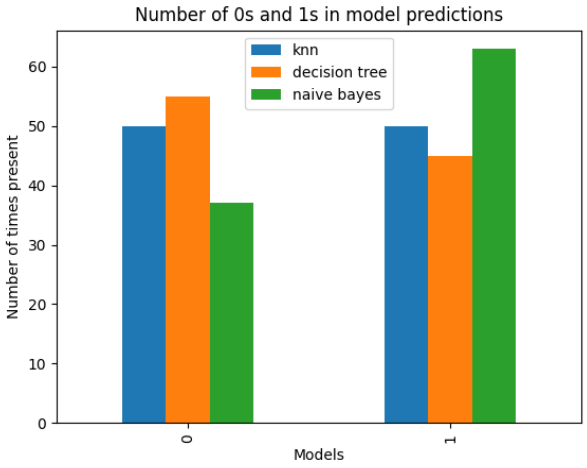
The following output shows what my program gives:

```
   (0, acc)  (1, acc)  (2, acc)
```

```
0  0.778846  0.740385  0.730769
1  0.807692  0.721154  0.750000
2  0.769231  0.798077  0.711538
3  0.701923  0.740385  0.653846
4  0.778846  0.711538  0.653846
5  0.769231  0.721154  0.634615
6  0.721154  0.730769  0.682692
7  0.798077  0.769231  0.711538
8  0.846154  0.788462  0.711538
9  0.759615  0.826923  0.788462
    0   1   2
0  45  60  36
1  55  40  64
```

In this case the 0 is kNN, 1 is the decision tree, and 2 is the Naive Bayes models. This order was chosen as this is the order I went with in terms of my favouring of models. The top half shows the accuracy across all of the 10 folds. The last little table indicates how many of each class were predicted in the final set of data.





I ended up choosing the kNN as it gives me a good accuracy every time, and always seems to beat the other two. This also gives me the closest to the balanced class output prediction on the final data set. Over many runs this seemed to be the case so it seems to be pretty consistent which you can see in the box plot above. You can also see in the number of each class predicted on the right there is an exact 50/50 split for this knn run.

The second choice I had was the Decision Tree model. This was chosen as it had the next highest accuracy, and also balance, sometimes outperforming my kNN model on both fronts, but not very often. You can also see this in the above charts with an outlier appearing higher than the accuracy for knn.

Prediction

Building the model

Both of the models are built the same way. Using the two models chosen from above we have the `KNeighborsClassifier(n_neighbors=7, weights='uniform', p=1)` and also the `DecisionTreeClassifier(min_samples_split=5, random_state=246)` model. These were both carefully tuned, with choosing appropriate hyperparameters based on the accuracy on 9 folds of a graph search, which measure both accuracy and balance of the predicted set.

Once all 3 models were tuned we got to the models you can see above plus a Naive Bayes model. The two above were chosen after a 10 fold search run a few times for consistency. They were judged on their accuracy throughout the folds as well as the balance of the classes retrieved from testing on the final unlabelled data.

Both of the models above were trained over the large dataset created from the smote function, which allowed it to run over 1300 instances with a perfectly balanced class set. This trained model was then applied over the final 100 unlabelled instances to predict the data you see in the following table.

Prediction

The following table contains the two predictions that were made from the above. Predict1 is the kNN one and Predict 2 is the Decision tree.

ID	Predict1	Predict2		ID	Predict1	Predict2		ID	Predict1	Predict2
1001	1	1		1034	0	1		1067	1	0
1002	1	1		1035	1	1		1068	0	0
1003	0	0		1036	1	0		1069	0	0
1004	0	0		1037	1	0		1070	1	1
1005	0	0		1038	0	0		1071	1	0
1006	0	0		1039	1	1		1072	1	0
1007	0	1		1040	1	0		1073	0	0
1008	0	0		1041	1	0		1074	0	0
1009	1	1		1042	1	1		1075	1	1
1010	0	0		1043	0	0		1076	1	1
1011	0	0		1044	0	1		1077	0	1
1012	0	0		1045	0	1		1078	1	0

1013	1	1		1046	1	1		1079	0	0
1014	1	0		1047	0	0		1080	1	1
1015	1	0		1048	0	0		1081	0	0
1016	0	1		1049	0	0		1082	0	0
1017	1	0		1050	0	1		1083	0	1
1018	1	1		1051	0	0		1084	0	0
1019	1	0		1052	1	1		1085	0	0
1020	0	0		1053	1	1		1086	0	0
1021	0	1		1054	1	1		1087	1	1
1022	1	0		1055	1	0		1088	0	0
1023	1	1		1056	0	0		1089	0	0
1024	0	0		1057	1	1		1090	0	0
1025	0	0		1058	1	1		1091	0	1
1026	0	0		1059	1	1		1092	1	1
1027	1	1		1060	0	0		1093	1	1
1028	0	1		1061	0	1		1094	0	0
1029	0	0		1062	1	1		1095	1	1
1030	1	0		1063	1	0		1096	1	1
1031	0	1		1064	0	0		1097	0	0
1032	1	1		1065	0	0		1098	0	1
1033	1	1		1066	1	0		1099	0	0
								1100	0	0

Accuracy of Prediction

For Predict1 which is the KNN prediction I'd say it's 64% accurate. This is taken from my submissions to the evaluation which gave me around this accuracy.

For Predict2 which is the decision tree model I'd say it's 63% accurate. I did not run this through an evaluation so this is a value that is considered slightly less than my kNN which was measured.

Comments on the Prediction

The balance for both of these is pretty close to the 50/50 on either class. This was a driving factor between a lot of my hyperparameter tuning and model deciding. There are still some inconsistencies between runs of the program which can make the outputs vary significantly sometimes.

The accuracy was measured over two evaluations with the second one doing worse because of a line I didn't realise I hadn't commented out. Because of this I only have one evaluation of my performance to actually go off so a lot of the accuracy discussion is just speculation.

Conclusion

Over the course of this report, many procedures and techniques took place to come from a dataset that had numerous issues like duplicated attributes, skewed data, missing data and finally turning this into a dataset that would be ready to have classification methods placed through it.

Classification was a big task of this report, investigating 3 different models to classify data: KNN, Naive Bayes and Decision Trees. The hyperparameters for each of these were investigated on the dataset to find their best potential, and the results of all 3 were compared with each other to find which one gives the “best” potential prediction.

In the end the KNN and Decision tree models were favoured with their parameters discussed earlier.

References - APA 7th

Pham, S (2020). *Data_mining_pracs.ipynb*. https://colab.research.google.com/drive/1P-BSz-o8FqADhEjKHZ9Ks0ZxrgTW7bTa

Geller, S (2019). *Normalization vs Standardization — Quantitative analysis*. https://towardsdatascience.com/normalization-vs-standardization-quantitative-analysis-a91e8a79cebf

Analytics Vidhya (2020). *10 Techniques to deal with Imbalanced Classes in Machine Learning*. https://www.analyticsvidhya.com/blog/2020/07/10-techniques-to-deal-with-class-imbalance-in-machine-learning/#h2_1

Pydata.org (2021). *pandas documentation*. https://pandas.pydata.org/pandas-docs/stable/index.html

SkitLearn.org (2021). *Tuning the hyper-parameters of an estimator*. https://scikit-learn.org/stable/modules/grid_search.html

SkitLearn.org(2021). *Parameter estimation using grid search with cross-validation*. https://scikit-learn.org/stable/auto_examples/model_selection/plot_grid_search_digits.html

Rençberoğlu, E (2019). *Fundamental Techniques of Feature Engineering for Machine Learning*. https://towardsdatascience.com/feature-engineering-for-machine-learning-3a5e293a5114

Plummer, A (2020). *Box-Cox Transformation: Explained*. https://towardsdatascience.com/box-cox-transformation-explained-51d745e34203