CCT College Dublin

Assessment Cover Page

To be provided separately as a word doc for students to include with every submission

Module Title:	Machine Learning
Assessment Title:	CA2_ML_HDip_Lvl8
Lecturer Name:	Muhammad Iqbal
Student Full Name:	Thomas Kelly
Student Number:	sba22259
Assessment Due Date:	6th January 2023
Date of Submission:	31st December 2022
Word Count	2744

Declaration

By submitting this assessment, I confirm that I have read the CCT policy on Academic Misconduct and understand the implications of submitting work that is not my own or does not appropriately reference material taken from a third party or other source. I declare it to be my own work and that all material from third parties has been appropriately referenced. I further confirm that this work has not previously been submitted for assessment by myself or someone else in CCT College Dublin or any other higher education institution.

Figures

Business Understanding

The brief of this continuous assessment (CA), stated to use the 'Shill Bidding Dataset', to classify future bids into the classes 0 (normal) and 1 (anomalous).

From reading the brief, it was understood to use the 'Class' column, as the target variable for the models I would create. These models had to consist of classification and clustering examples. As will be demonstrated in the CA, I used 2 classification models (SVM and GNB), 1 clustering model (K-Means) and a forecasting model (LSTM RNN).

Also understood from the brief was that each model had to be optimised and compared to one another. This was completed through the use of GridSearchCV for cross-validation and hyperparameter tuning.

Data Understanding

I imported the 'Shill Bidding Dataset' into a *Jupyter notebook* using *pandas*. From my initial viewing of the dataset, I could see that there were 6321 rows and 13 columns in the dataset *(see Figure 1)*.

	Record_ID	Auction_ID	Bidder_ID	Bidder_Tendency	Bidding_Ratio	Successive_Outbidding	Last_Bidding	Auction_Bids	Starting_Price_Average	Early_Biddi
0	1	732	_***i	0.200000	0.400000	0.0	0.000028	0.000000	0.993593	0.0000
1	2	732	g***r	0.024390	0.200000	0.0	0.013123	0.000000	0.993593	0.0131
2	3	732	t***p	0.142857	0.200000	0.0	0.003042	0.000000	0.993593	0.0030
3	4	732	7***n	0.100000	0.200000	0.0	0.097477	0.000000	0.993593	0.0974
4	5	900	z***z	0.051282	0.222222	0.0	0.001318	0.000000	0.000000	0.0012
...	...	...	...	...	...	...	...	...	...	
6316	15129	760	l***t	0.333333	0.160000	1.0	0.738557	0.280000	0.993593	0.6863
6317	15137	2481	s***s	0.030612	0.130435	0.0	0.005754	0.217391	0.993593	0.0000
6318	15138	2481	h***t	0.055556	0.043478	0.0	0.015663	0.217391	0.993593	0.0156
6319	15139	2481	d***d	0.076923	0.086957	0.0	0.068694	0.217391	0.993593	0.0004
6320	15144	2481	a***l	0.016393	0.043478	0.0	0.340351	0.217391	0.993593	0.3403

6321 rows × 13 columns

Figure 1. Overview of 'Shill Bidding Dataset'

As mentioned in the brief, the column, 'Class', seemed to consist of binary values, *0* and *1*. To test this I used the *set()* function on the 'Class' column *(see Figure 2)*. This assured me that there were only 2 unique values. To see the dispersion of these values, I created a count plot graph *(see Figure 3)*. This graph showed less than 1000 instances of anomalous bids (*1*), and over 5000 normal bids. To ensure the bias in this dataset doesn't affect my model, I will use a *k-fold cross-validation*, later on, to remove bias.

```
set(df["Class"])
```

```
{0, 1}
```

Figure 2. Unique values of 'Class' column.

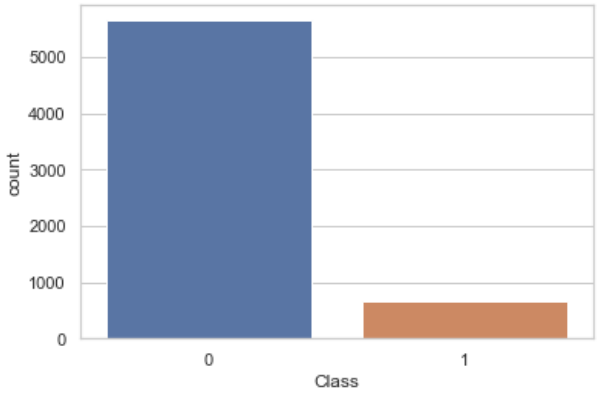

Figure 3. Count Plot of 'Class' column.

From examining the 'Successive_Outbidding' column, I was able to identify 3 unique values *(see Figure 4)*, which puzzled me, as the attribute information led me to think it would be a binary feature *(Uci.edu, 2020)*. This was something I would keep in mind moving forward.

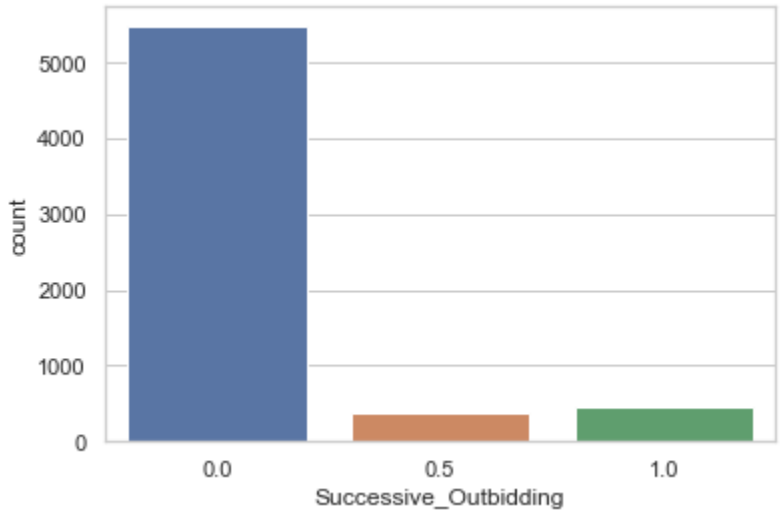

Figure 4. Unique values of 'Successive Outbidding'.

Evident from looking at the 'Bidder_ID' column. It contained the object datatype *(see Figure 5)*, this is not useful in machine learning modelling, so this would be amended in my data preparation.

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6321 entries, 0 to 6320
Data columns (total 13 columns):
 #   Column                  Non-Null Count  Dtype
---  ------                  --------------  -----
 0   Record_ID               6321 non-null   int64
 1   Auction_ID              6321 non-null   int64
 2   Bidder_ID               6321 non-null   object
 3   Bidder_Tendency         6321 non-null   float64
 4   Bidding_Ratio           6321 non-null   float64
 5   Successive_Outbidding   6321 non-null   float64
 6   Last_Bidding            6321 non-null   float64
 7   Auction_Bids            6321 non-null   float64
 8   Starting_Price_Average  6321 non-null   float64
 9   Early_Bidding           6321 non-null   float64
 10  Winning_Ratio           6321 non-null   float64
 11  Auction_Duration        6321 non-null   int64
 12  Class                   6321 non-null   int64
dtypes: float64(8), int64(4), object(1)
memory usage: 642.1+ KB
```

Figure 5. Dataset data types.

Data Preparation

Continuing on from identifying that the 'Bidder_ID' column is an object, I dropped the column, as it would be ineffective in the models *(see Figure 6)*.

```
df = df.drop(columns='Bidder_ID')
```

```
df
```

	Record_ID	Auction_ID	Bidder_Tendency	Bidding_Ratio	Successive_Outbidding	Last_Bidding	Auction_Bids	Starting_Price_Average	Early_Bidding	Winning
0	1	732	0.200000	0.400000	0.0	0.000028	0.000000	0.993593	0.000028	0.
1	2	732	0.024390	0.200000	0.0	0.013123	0.000000	0.993593	0.013123	0.
2	3	732	0.142857	0.200000	0.0	0.003042	0.000000	0.993593	0.003042	1.
3	4	732	0.100000	0.200000	0.0	0.097477	0.000000	0.993593	0.097477	1.

Figure 6. Dropped the 'Bidder_ID' column.

In some work that I found on Kaggle by 'Anastasiia Apanasiuk', which covered the same dataset, I found that she ran a decision tree classifier to find the most important features *(anastasiiaapanasiuk, 2022)*. Rather than attempting the same thing myself to yield the same results, I am using her findings. She found that 'Successive_Outbidding' was by far the most important feature and 'Early_bidding', 'Auction_ID' and 'Starting_Price_Average' were the least important. To increase the usefulness of the data, I dropped the 3 least useful columns according to Apanasiuk *(see Figure 7)*.

```
df
```

	Record_ID	Bidder_Tendency	Bidding_Ratio	Successive_Outbidding	Last_Bidding	Auction_Bids	Winning_Ratio	Auction_Duration	Class
0	1	0.200000	0.400000	0.0	0.000028	0.000000	0.666667	5	0
1	2	0.024390	0.200000	0.0	0.013123	0.000000	0.944444	5	0
2	3	0.142857	0.200000	0.0	0.003042	0.000000	1.000000	5	0
3	4	0.100000	0.200000	0.0	0.097477	0.000000	1.000000	5	0
4	5	0.051282	0.222222	0.0	0.001318	0.000000	0.500000	7	0
...	...	...	...	...	...	...	...	...	...
6316	15129	0.333333	0.160000	1.0	0.738557	0.280000	0.888889	3	1
6317	15137	0.030612	0.130435	0.0	0.005754	0.217391	0.878788	7	0
6318	15138	0.055556	0.043478	0.0	0.015663	0.217391	0.000000	7	0
6319	15139	0.076923	0.086957	0.0	0.068694	0.217391	0.000000	7	0
6320	15144	0.016393	0.043478	0.0	0.340351	0.217391	0.000000	7	0

6321 rows × 9 columns

Figure 7. Dropping 'Early_bidding', 'Auction_ID' and 'Starting_Price_Average' from DataFrame.

To prepare the dataset to get more accurate results in modelling, I scaled the data. I was unsure which type of scaler I wanted to use, so I created a version of the data using *preprocessing.scale (see Figure 8)* and a *minmax* scaler *(see Figure 9)*.

	Record_ID	Auction_ID	Bidder_Tendency	Bidding_Ratio	Successive_Outbidding	Last_Bidding	Auction_Bids	Starting_Price_Average	Early_Bidding	Winning
0	1	732	0.291571	2.070638	0.0	-1.218447	-0.907433	1.063074	-1.131055	0.
1	2	732	-0.599542	0.549957	0.0	-1.183993	-0.907433	1.063074	-1.096663	1.
2	3	732	0.001606	0.549957	0.0	-1.210517	-0.907433	1.063074	-1.123139	1.
3	4	732	-0.215868	0.549957	0.0	-0.962047	-0.907433	1.063074	-0.875118	1.
4	5	900	-0.463082	0.718921	0.0	-1.215053	-0.907433	-0.965191	-1.127866	0.
...	...	...	...	...	...	...	...	...	...	
6316	15129	760	0.968156	0.245821	1.0	0.724708	0.189609	1.063074	0.671496	1.
6317	15137	2481	-0.567969	0.021024	0.0	-1.203380	-0.055692	1.063074	-1.131102	1.
6318	15138	2481	-0.441397	-0.640141	0.0	-1.177309	-0.055692	1.063074	-1.089991	-0.
6319	15139	2481	-0.332970	-0.309559	0.0	-1.037779	-0.055692	1.063074	-1.130038	-0.
6320	15144	2481	-0.640121	-0.640141	0.0	-0.323019	-0.055692	1.063074	-0.237244	-0.

Figure 8. Preprocessing.scaled DataFrame.

	Record_ID	Auction_ID	Bidder_Tendency	Bidding_Ratio	Successive_Outbidding	Last_Bidding	Auction_Bids	Starting_Price_Average	Early_Bidding	Winning
0	1	732	0.200000	0.392857	0.0	0.000028	0.000000	0.993657	0.000028	0.
1	2	732	0.024390	0.190476	0.0	0.013124	0.000000	0.993657	0.013124	0.
2	3	732	0.142857	0.190476	0.0	0.003042	0.000000	0.993657	0.003042	1.
3	4	732	0.100000	0.190476	0.0	0.097487	0.000000	0.993657	0.097487	1.
4	5	900	0.051282	0.212963	0.0	0.001318	0.000000	0.000000	0.001242	0.
...	...	...	...	...	...	...	...	...	...	
6316	15129	760	0.333333	0.150000	1.0	0.738631	0.355224	0.993657	0.686426	0.
6317	15137	2481	0.030612	0.120083	0.0	0.005755	0.275795	0.993657	0.000010	0.
6318	15138	2481	0.055556	0.032091	0.0	0.015665	0.275795	0.993657	0.015665	0.
6319	15139	2481	0.076923	0.076087	0.0	0.068701	0.275795	0.993657	0.000415	0.
6320	15144	2481	0.016393	0.032091	0.0	0.340384	0.275795	0.993657	0.340384	0.

Figure 9. Minmax scaled DataFrame.

As can be seen in both ‘Figures 8 and 9’, there are columns from the original *DataFrame*, ‘Figure 1’, which have not been scaled, this is due to them being integer values, which are best not to be scaled.

When moving into the modelling phase, I will test both *DataFrames* above, to see which yields the most accurate results, then continue with the highest accuracy scoring *DataFrame*. My hypothesis is that the *Minmax* scaler will perform better as the data is manipulated to have greater variance, whereas in *.scale* the data is only standardised along an axis *(scikit-learn, 2022)*.

Modelling

To begin modelling, I defined my 'X' (features) and 'y' (target) labels *(see Figure 10)*. The 'X' label consisted of all of the remaining features in the dataset, except 'Class', which is the 'y' label.

```
y.head()
```

```
0    0
1    0
2    0
3    0
4    0
Name: Class, dtype: int64
```

```
X.head()
```

	Record_ID	Bidder_Tendency	Bidding_Ratio	Successive_Outbidding	Last_Bidding	Auction_Bids	Winning_Ratio	Auction_Duration
0	1	0.200000	0.400000	0.0	0.000028	0.0	0.666667	5
1	2	0.024390	0.200000	0.0	0.013123	0.0	0.944444	5
2	3	0.142857	0.200000	0.0	0.003042	0.0	1.000000	5
3	4	0.100000	0.200000	0.0	0.097477	0.0	1.000000	5
4	5	0.051282	0.222222	0.0	0.001318	0.0	0.500000	7

Figure 10. 'X' and 'y' labels

Finding the 'y' value can be seen as a classification problem, more specifically, a binary classification model, as there are only 2 values for the 'y' label.

I planned on using 2 forms of classification model to classify the 'Shill Bidding Dataset', these are; *SVM* (Support Vector Machines) and *Naive Bayes* models. These are due to a large amount of hyperparameter tuning that is possible. Naive Bayes is a strong model when features are independent, making them good for binary classification tasks, such as spam filtering, which this task requires. SVMs can make use of *five-fold cross-validation*, which I had planned on using for the cross-validation task asked for in the brief, so in my opinion, these 2 models were ideal *(Garg, 2018)*.

For the classification models, I created a train, and test split, with the test size being 30%. I would iterate on this value after the hyperparameter tuning of the models.

I created an SVM model by importing SVC from *sklearn*. When I ran the model, I printed the precision, recall and f1-score. From the untuned SVM model, the accuracy score was 91% *(see Figure 11)*. What we can see from 'Figure 11', was that the model did not get any label

classified as ‘1’ correct. This can also be seen in ‘Figure 12’ where the model incorrectly predicted that 8.91% of the results were positive when they were negative.

	precision	recall	f1-score	support
0	0.91	1.00	0.95	1728
1	0.00	0.00	0.00	169
accuracy			0.91	1897
macro avg	0.46	0.50	0.48	1897
weighted avg	0.83	0.91	0.87	1897

Figure 11. Precision, Recall and F1-Score of the untuned SVM

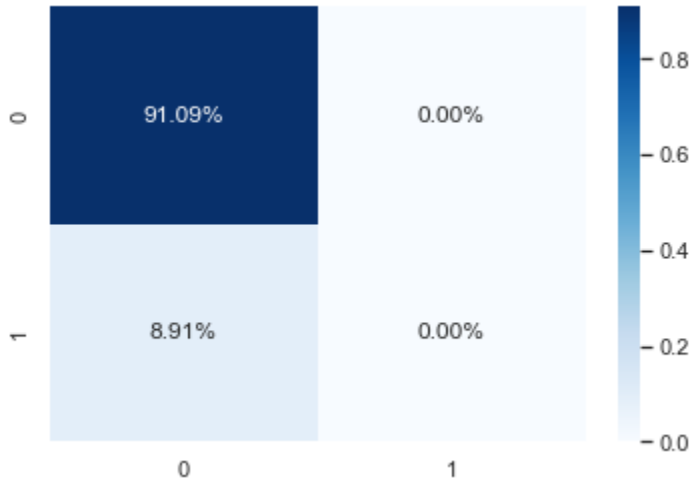

Figure 12. Confusion Matrix for untuned SVM.

After getting the results of the untuned SVM, I defined which hyper-parameters of the SVM I wished to tune. I used *GridSearchCV* to find the optimal variation of the hyperparameters I selected *(see Figure 13)*.

```
# defining parameter range
param_grid = {'C': [0.1, 1, 10, 100, 1000],
              'gamma': [1, 0.1, 0.01, 0.001, 0.0001],
              'kernel': ['rbf', 'linear']}

grid = GridSearchCV(SVC(), param_grid, refit = True, verbose = 3)

# fitting the model for grid search
grid.fit(X_train, y_train)
```

```
Fitting 5 folds for each of 50 candidates, totalling 250 fits
[CV 1/5] END ........C=0.1, gamma=1, kernel=rbf;, score=0.886 total time=   0.5s
[CV 2/5] END ........C=0.1, gamma=1, kernel=rbf;, score=0.886 total time=   0.5s
[CV 3/5] END ........C=0.1, gamma=1, kernel=rbf;, score=0.886 total time=   0.5s
[CV 4/5] END ........C=0.1, gamma=1, kernel=rbf;, score=0.885 total time=   0.5s
[CV 5/5] END ........C=0.1, gamma=1, kernel=rbf;, score=0.886 total time=   0.5s
[CV 1/5] END .....C=0.1, gamma=1, kernel=linear;, score=0.922 total time=  18.8s
[CV 2/5] END .....C=0.1, gamma=1, kernel=linear;, score=0.938 total time=  31.6s
[CV 3/5] END .....C=0.1, gamma=1, kernel=linear;, score=0.914 total time=  18.5s
[CV 4/5] END .....C=0.1, gamma=1, kernel=linear;, score=0.919 total time=  11.8s
[CV 5/5] END .....C=0.1, gamma=1, kernel=linear;, score=0.919 total time=  37.9s
[CV 1/5] END ......C=0.1, gamma=0.1, kernel=rbf;, score=0.886 total time=   0.4s
[CV 2/5] END ......C=0.1, gamma=0.1, kernel=rbf;, score=0.886 total time=   0.4s
[CV 3/5] END ......C=0.1, gamma=0.1, kernel=rbf;, score=0.886 total time=   0.4s
[CV 4/5] END ......C=0.1, gamma=0.1, kernel=rbf;, score=0.885 total time=   0.4s
[CV 5/5] END ......C=0.1, gamma=0.1, kernel=rbf;, score=0.886 total time=   0.4s
[CV 1/5] END ...C=0.1, gamma=0.1, kernel=linear;, score=0.922 total time=  18.7s
[CV 2/5] END ...C=0.1, gamma=0.1, kernel=linear;, score=0.938 total time=  31.6s
[CV 3/5] END ...C=0.1, gamma=0.1, kernel=linear;, score=0.914 total time=  18.3s
```

Figure 13. GridSearchCV finding optimal variations of SVM hyper-parameters.

The first parameter I decided to tune was which SVM kernel to use. I selected the 'RBF' and 'Linear' kernels. The 'RBF' kernel function is a generalised function which does not require much prior knowledge about the data *(Team, 2017)*. 'Linear' kernels are slightly different as they are one-dimensional in nature, and are preferred in classification problems *(Dataaspirant, 2020)*.

I decided to tune the 'C' value, which adjusts the misclassifying of the SVM hyperplane. The default value for 'C' is 1, so I iterated through 0.1, meaning a larger margin of error, then 10, 100 and 1000, meaning a much smaller margin of error *(Kent Munthe Caspersen, 2015)*.

The 'gamma' value was another value I decided to test multiple values of. This is a parameter of SVMs which use the RBF kernel. Gamma refers to the spread of the kernel's decision region. The higher the gamma, the higher the 'curve' of the decision boundary, meaning more confidently defined classifications *(Albon, 2017)*.

	precision	recall	f1-score	support
0	0.95	0.99	0.97	1728
1	0.77	0.45	0.57	169
accuracy			0.94	1897
macro avg	0.86	0.72	0.77	1897
weighted avg	0.93	0.94	0.93	1897

Figure 14. Accuracy scores of tuned SVM

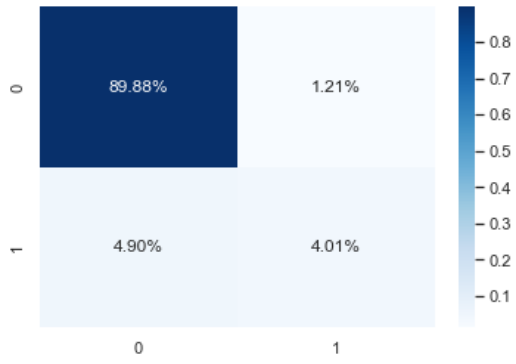

Figure 15. Tuned SVM Confusion Matrix

As can be seen in the results above *(see Figures 14 and 15)*, the accuracy score of the SVM went from 91% to 94% after hyperparameter tuning. In 'Figure 15', you can see that unlike 'Figure 12', the true positive score has risen from 0 to 4.01%, meaning some values of 1 have been correctly classified.

Figure 16 below, shows the optimal parameter values of the SVM model. I used these same parameters to train a model using the 'scale' and 'minmax' *DataFrames* I created in the data preparation phase.

```
{'C': 100, 'gamma': 1, 'kernel': 'linear'}
SVC(C=100, gamma=1, kernel='linear')
```

Figure 16. Optimal SVM parameters found by GridSearchCV

The *DataFrame* created by using 'scale' to scale the data, resulted in an accuracy score of 98% *(see Figure 17)*, this beats the 'minmax' scaled *DataFrame* which had an accuracy score of 94% *(see Figure 18)*. Going forward, I will be using the 'scale' *DataFrame*, and will be referring to this as 'DataFrame', 'df' or the 'scaled data set'.

	precision	recall	f1-score	support
0	0.98	0.99	0.99	1728
1	0.91	0.81	0.86	169
accuracy			0.98	1897
macro avg	0.94	0.90	0.92	1897
weighted avg	0.98	0.98	0.98	1897

Figure 17. 'scale' DataFrame accuracy.

	precision	recall	f1-score	support
0	0.95	0.99	0.97	1728
1	0.77	0.46	0.57	169
accuracy			0.94	1897
macro avg	0.86	0.72	0.77	1897
weighted avg	0.93	0.94	0.93	1897

Figure 18. 'minmax' DataFrame accuracy.

The second classification model I created was a *Gaussian Naive Bayes* (GNB) model. I selected this model due to its efficiency in binary classification tasks, as it is especially popular in spam filtering *(Sharma, 2021)*.

Before tuning my GNB model, I got an accuracy score of 97%, which I would consider fairly high for an untuned model *(see Figure 19)*. The confusion matrix *(Figure 20)*, shows that there are 8.75% true negatives, this was higher than that of the tuned SVM model.

	precision	recall	f1-score	support
0	1.00	0.97	0.98	1728
1	0.74	0.98	0.84	169
accuracy			0.97	1897
macro avg	0.87	0.97	0.91	1897
weighted avg	0.98	0.97	0.97	1897

Figure 19. Untuned GNB accuracy score.

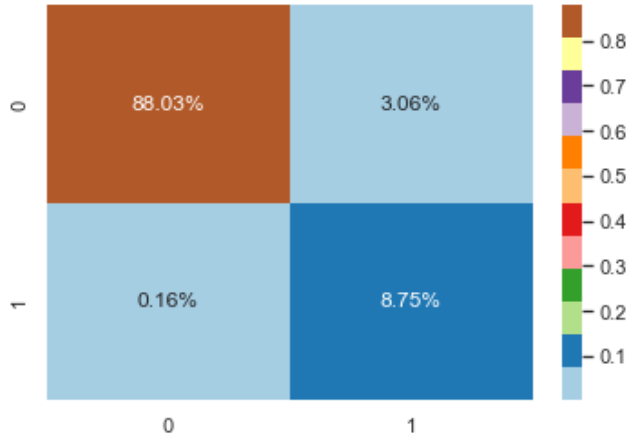

Figure 20. Confusion Matrix for untuned GNB

To tune the GNB model I used *GridSearchCV*. I iterated through the 'var_smoothing' attribute using np.logspace (as 'var_smoothing' uses log scale values), with the range between 0 and -9 *(Numpy.org, 2022)*. After tuning the GNB model using *GridSearchCV* the accuracy of the model did not improve. The accuracy score remained at 97%, as can be seen in 'Figure 21'.

	precision	recall	f1-score	support
0	1.00	0.97	0.98	1728
1	0.74	0.98	0.84	169
accuracy			0.97	1897
macro avg	0.87	0.97	0.91	1897
weighted avg	0.98	0.97	0.97	1897

Figure 21. Tuned GNB accuracy score.

The same result, of no improvement after tuning can be seen in the GNB model's confusion matrix *(see Figure 22*), which is identical to the untuned model's *(Figure 20)*. This is due to the model's initial 'var_smoothing' value being the same as the *GridSearchCV's* 'best_params_' value *(see Figure 23)*.

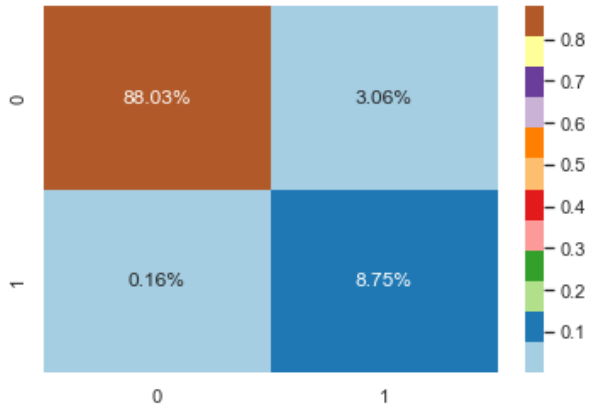

Figure 22. Tuned GNB Model's Confusion Matrix

```
{'var_smoothing': 1e-09}
GaussianNB()
```

Figure 23. 'best_params_' and 'best_estimator_' values.

Comparing both the tuned SVM and tuned GNB models; the SVM model has a 1% higher accuracy score than the GNB according to 'Figure 17'. But, the tuned GNB performs better detecting negatives ('1'), which can be seen in 'Figure 22', in comparison to 'Figure 24'. However, the SVM performs slightly better in classifying the positive values, whether true or false, this is what obtained its higher accuracy score.

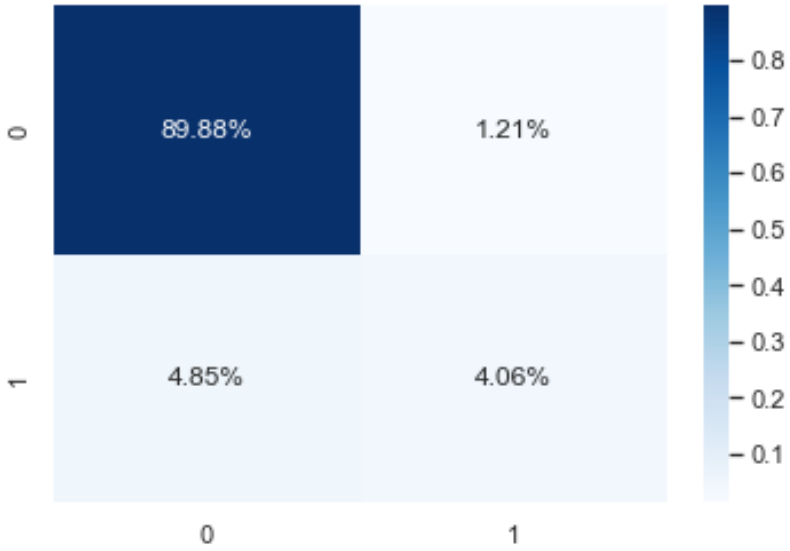

Figure 24. Scaled and tuned SVM Confusion Matrix

After creating and comparing the two classification models, I wanted to try a clustering model to see how its accuracy compared to the SVM and GNB. For the clustering model, I chose to use *K-Means* clustering, which is also an unsupervised model.

Before creating the K-Means model, I completed a principal component analysis (PCA) of the dataset. I scaled my X label using *StandardScaler*, then used the *sklearn PCA* class with 2 'n_components' to define the two principal components to be fit. From plotting components onto a graph *(see Figure 25)* we can see that they are linear in characteristic. The class assignment from the graph *(Figure 25)* shows a distribution of the normal and anomalous class values throughout the graph, which is different to what will be shown when examining the K-Means graph *(Figure 28)*.

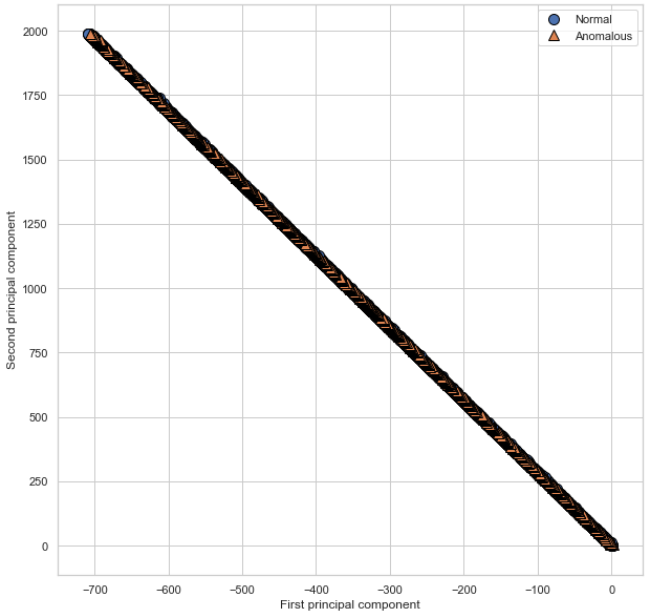

Figure 25. PCA graph

After importing KMeans from sklearn, I defined 2 'n_clusters' and fit my X_pca (principal components) to the model *(see Figure 26)*. I then assigned the PCA columns to different variables and transformed them into NumPy arrays *(see Figure 27)*.

```
kmeans_bids_2 = KMeans(n_clusters = 2)
kmeans_bids_2.fit(X_pca)
C = kmeans_bids_2.labels_
C.shape
assign_bids = C.reshape(-1, 1)
```

Figure 26. Creating K-Means model

```
# To plot the cluster data, consider two columns
X1 = X_pca[:,0]
X2 = X_pca[:,1]

# Transform into numpy array
X11 = X1[:, np.newaxis]
X22 = X2[:, np.newaxis]
```

Figure 27. X_pca to NumPy array

As seen in 'Figure 28', I mapped the clusterings from the model. The top percentage of the graph plot is seen as a cluster and the bottom as another. This is due to 2 clusters being used. The accuracy of this graph is 51% as can be seen in 'Figure 29'. As we saw in 'Figure 25', there is a dispersion of each 'Class' value throughout the graph, so the best accuracy we can expect from 2 clusters is around 51% as was shown in 'Figure 28'.

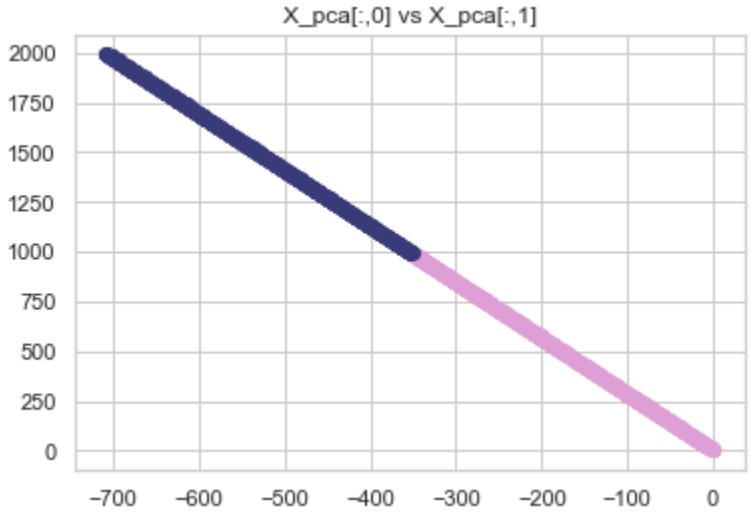

Figure 28. K-Means Cluster Graph

	precision	recall	f1-score	support
0	0.90	0.51	0.65	5646
1	0.11	0.51	0.18	675
accuracy			0.51	6321
macro avg	0.50	0.51	0.41	6321
weighted avg	0.81	0.51	0.60	6321

Figure 29. K-Means Accuracy Score

To improve the accuracy score of the K-Means model, I used GridSearchCV. I iterated over the 'n_clusters' parameter with a range of 2 to 300 *(see Figure 30)*. After running the model with GridSearchCV, the 'best_params_' value was 293 clusters, but, when attempting to view the accuracy scores, the scores were all 0. 'Figure 31' illustrates that the only score I received back was a weighted_avg score, which I cannot take as an accurate accuracy score as it is weighted.

```
# defining parameter range
param_grid = {'n_clusters': range(2, 300)}

grid = GridSearchCV(KMeans(), param_grid, refit = True, verbose = 3, cv=10)
```

Figure 30. GridSearchCV for K-Means Clustering

```
              accuracy                           0.00      6321
             macro avg       0.00      0.00      0.00      6321
          weighted avg       0.79      0.00      0.01      6321
```

Figure 31. Tuned K-Means accuracy scores.

Taking the highest accuracy score I got from the K-Means clustering models, was 51%. I would say that K-Means was not an optimal model to use for the given dataset. But, I got to experiment with a clustering algorithm and an unsupervised model, which I can compare against the supervised classification models.

The final model I created was to predict values of the 'Class' column using an LSTM recurrent neural network. I had no expectation of this model to perform well, as the dataset is not designed with forecasting in mind, and there is no linear format or time progression in the data.

To create my training data, I used the first 5000 rows of the 'Class' column and reshaped the data to be able to be split into an x and y training data split. These sets were then made into NumPy arrays (see Figure 32).

```
rnn.add(LSTM(units = 45, return_sequences = True, input_shape = (x_training_data.shape[1], 1)))
rnn.add(Dropout(0.2))

rnn.add(LSTM(units = 45, return_sequences = True))
rnn.add(Dropout(0.2))

rnn.add(LSTM(units = 45, return_sequences = True))
rnn.add(Dropout(0.2))

rnn.add(LSTM(units = 45))
rnn.add(Dropout(0.2))

# Output layer
rnn.add(Dense(units = 1))
```

Figure 32. LSTM neurons in RNN.

The initial untuned RNN model I created had 3 LSTM neurons, with units of 45, and a dropout layer of 0.2, this can be seen in 'Figure 33'. The compiler used the 'adam' optimiser and a binary cross-entropy loss function. I used a batch size of 32 and trained it with 100 epochs.

```
Epoch 100/100
155/155 [==============================] - 3s 22ms/step - loss: 0.3380 - accuracy: 0.8933
```

Figure 33. Loss and Accuracy scores of RNN model after 100 epochs.

As can be seen in 'Figure 33', the final accuracy score of the model was 0.89%, which is very low. The loss function's value at the first epoch was 0.3460, meaning there was only a 0.0080 improvement between the 100 epochs.

‘Figure 34’ below shows the predictions made by the RNN model. As can be seen, the values on the y-axis are between 0.05 and 0.13, these values ideally should be more extreme, between 0 and 1. ‘Figure 35’, shows the predicted values overlayed onto the actual values.

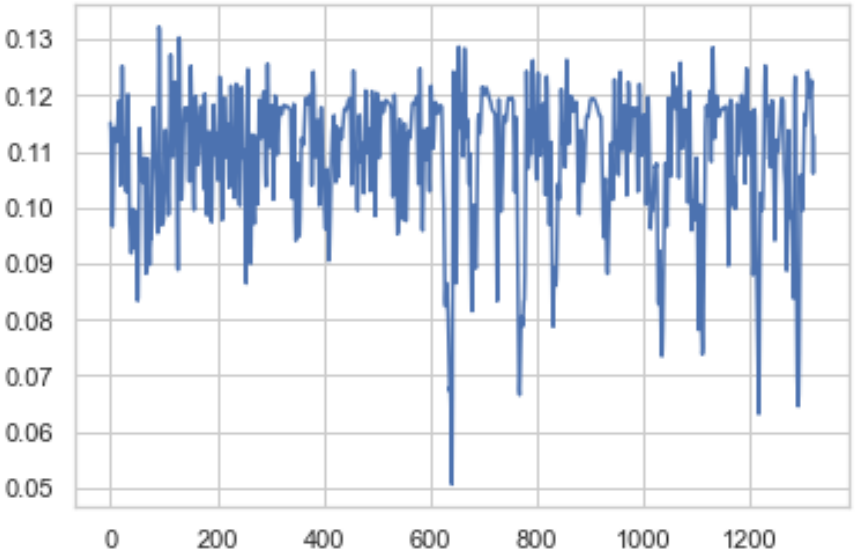

Figure 34. RNN Prediction Data

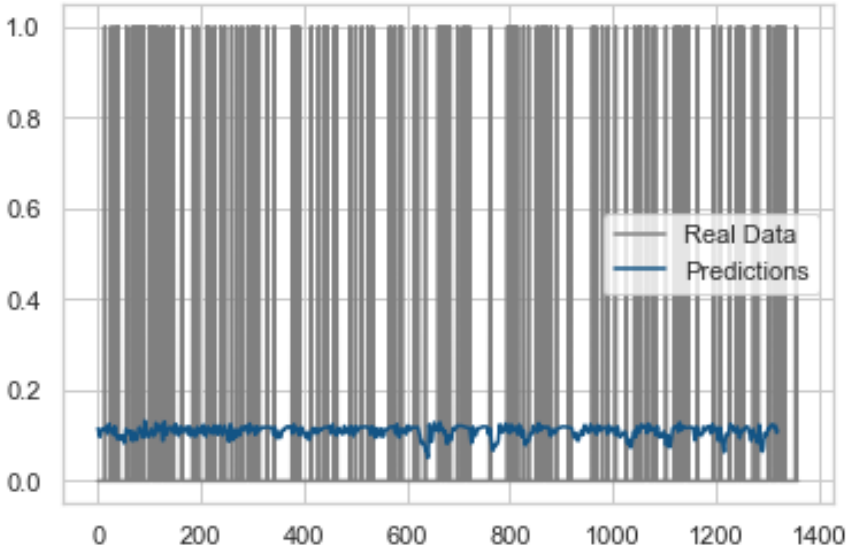

Figure 35. Real and RNN Prediction Data

In an attempt to improve the model’s performance, I used *GridSearchCV* to iterate through the batch size and the number of epochs *(Brownlee, 2022)*. This however did not yield any better results, as can be seen in ‘Figure 36’, the accuracy score remained the same for each batch and epoch value tried.

```
Best: 0.893347 using {'batch_size': 10, 'epochs': 10}
0.893347 (0.000255) with: {'batch_size': 10, 'epochs': 10}
0.893347 (0.000255) with: {'batch_size': 10, 'epochs': 50}
0.893347 (0.000255) with: {'batch_size': 10, 'epochs': 100}
0.893347 (0.000255) with: {'batch_size': 20, 'epochs': 10}
0.893347 (0.000255) with: {'batch_size': 20, 'epochs': 50}
0.893347 (0.000255) with: {'batch_size': 20, 'epochs': 100}
0.893347 (0.000255) with: {'batch_size': 32, 'epochs': 10}
0.893347 (0.000255) with: {'batch_size': 32, 'epochs': 50}
```

Figure 36. GridSearchCV used on LSTM RNN.

By changing the loss function when compiling the RNN from binary cross-entropy to mean squared error, I reduced the loss function value to 0.0950 after 100 epochs. But, the accuracy score did not change *(see Figure 37)*.

```
Epoch 100/100
155/155 [==============================] - 4s 23ms/step - loss: 0.0950 - accuracy: 0.8933
```

Figure 37. Using Mean Squared Error Loss Function on LSTM RNN.

Going back to the data prep for the RNN, I changed the X training data to take the previous 10 examples of the 'Class' value to get a prediction, instead of the last 40. This resulted in a slightly lower accuracy score for the model and a higher loss value *(see Figure 38)*.

```
Epoch 100/100
156/156 [==============================] - 2s 11ms/step - loss: 0.0954 - accuracy: 0.8930
```

Figure 38. Using the previous 10 examples instead of 40 to make predictions.

After changing the number of neuron layers and units to 60, there still was no change in the accuracy score or loss function value *(see Figure 39)*. Therefore I came to the conclusion that this model had reached the most accuracy as I could get it for predictions.

```
Epoch 50/50
155/155 [==============================] - 3s 20ms/step - loss: 0.0954 - accuracy: 0.8933
```

Figure 39. Accuracy and Loss after changing neuron layers and units.

Comparing Model Performance

Considering I got the highest accuracy scores from the SVM and GNB models, I decided to try and optimise their performance one last time by using multiple testing splits (5%, 10%, 15%, 20% and 30%) and using *five-fold cross-validation* on each split *(scikit-learn, 2013)*.

As can be seen in 'Figure 40' the SVM using a testing split of 10% gained the best results, with an accuracy score of 98%, which was identical to that of 'Figure 17', however, this did score relied less on the weighted average being higher, so I took this as the better score.

	precision	recall	f1-score	support
0	0.98	0.99	0.99	581
1	0.91	0.79	0.85	52
accuracy			0.98	633
macro avg	0.95	0.89	0.92	633
weighted avg	0.98	0.98	0.98	633

Figure 40. 10% testing Split SVM accuracy score.

When using the multiple test splits on the GNB model, the best accuracy results came from the 30% split, which was used previously *(see Figure 21)*.

To achieve better results from the two classification models there are multiple things I could have tried, such as more hyperparameter tuning, or just trying different random seeds. For the RNN, adjusting the number of neurons and changing the optimiser and loss functions could have proven beneficial, but due to time constraints and the fact that there was no increase in accuracy from previous attempts, this did not merit an attempt.

Comparing all of the models together (SVM, GNB, K-Means & LSTM RNN), it is apparent that the classification models performed far better than the clustering and prediction models. Given that this was a labelled dataset, it makes sense that the supervised models performed better, but considering that the RNN I used was a supervised model, its accuracy was disappointing. However, the supervised classification models did perform better than the unsupervised K-Means clustering model.

Hyperparameter tuning using *GridSearchCV* worked most effectively for the SVM model, resulting in an increase in accuracy of 8%, shown in 'Figure 11' and 'Figure 17'. When working to improve the results of the RNN, changing the loss function from binary cross-entropy to mean squared error, also produced a worthy improvement.

The accuracy score of 51% afforded by the K-Means clustering given the shape and relationship of the data was about the best I could obtain from it, however, it does take the form of ‘guesswork’ or randomness, as there is an almost 50% chance that the ‘Class’ attribute of a prediction is 0 or 1, as it is a binary classification problem. Therefore, I would not call this model smart, but more so lucky, if it made a correct prediction.

Conclusion

In conclusion, I am pleased with the accuracy scores of both the SVM and GNB models, as well as the hyperparameter tuning used to get higher accuracy scores. With more tuning, I feel these could be improved, but for the scope of the CA, I am content.

If I had more time on the project, I would attempt to remake the RNN or possibly use an ANN to try and make better predictions. Considering the classification models got such high accuracy scores, I felt this was unneeded, as I at least had something successful to compare the RNN to.

I would have also wished to create a regression model, as it is another form of supervised machine learning model which I feel could have performed well, but given the fact, the classification scores were so high, and there were 4 models used, an extra model was possibly unneeded. To give an opposing point, the hyperparameter tuning and optimisation of a regression model would have made a strong comparison to the other models.

References

- Albon, C. (2017). *SVC Parameters When Using RBF Kernel*. [online] Chrisalbon.com. Available at: https://chrisalbon.com/code/machine_learning/support_vector_machines/svc_parameters_using_rbf_kernel/ [Accessed 28 Dec. 2022].

- anastasiiaapanasiuk (2022). *Shill Bidding*. [online] Kaggle.com. Available at: https://www.kaggle.com/code/anastasiiaapanasiuk/shill-bidding [Accessed 28 Dec. 2022].

- Brownlee, J. (2022). *How to Grid Search Hyperparameters for Deep Learning Models in Python with Keras - MachineLearningMastery.com*. [online] MachineLearningMastery.com. Available

at:https://machinelearningmastery.com/grid-search-hyperparameters-deep-learning-models-python-keras/ [Accessed 30 Dec. 2022].

- Dataaspirant. (2020). *Seven Most Popular SVM Kernels*. [online] Available at: https://dataaspirant.com/svm-kernels/#t-1608140512018 [Accessed 28 Dec. 2022].

- Garg, R. (2018). *7 Types of Classification Algorithms*. [online] Analytics India Magazine. Available at: https://analyticsindiamag.com/7-types-classification-algorithms/ [Accessed 28 Dec. 2022].

- Kent Munthe Caspersen (2015) *What is the influence of C in SVMs with linear kernel?* [online] Cross Validated. Available at: https://stats.stackexchange.com/questions/31066/what-is-the-influence-of-c-in-svms-with-linear-kernel/159051#159051 [Accessed 28 Dec. 2022].

- Numpy.org. (2022). *numpy.logspace — NumPy v1.24 Manual*. [online] Available at: https://numpy.org/doc/stable/reference/generated/numpy.logspace.html [Accessed 29 Dec. 2022].

- scikit-learn. (2013). *3.1. Cross-validation: evaluating estimator performance*. [online] Available at: https://scikit-learn.org/stable/modules/cross_validation.html [Accessed 30 Dec. 2022].

- scikit-learn. (2022). *sklearn.preprocessing.MinMaxScaler*. [online] Available at: https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html [Accessed 28 Dec. 2022].

- scikit-learn. (2022). *sklearn.preprocessing.scale*. [online] Available at: https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.scale.html [Accessed 28 Dec. 2022].

- Sharma, P. (2021). *Implementing Gaussian Naive Bayes in Python - Analytics Vidhya*. [online] Analytics Vidhya. Available at: https://www.analyticsvidhya.com/blog/2021/11/implementation-of-gaussian-naive-bayes-in-python-sklearn/ [Accessed 28 Dec. 2022].

- Team, D. (2017). *Kernel Functions-Introduction to SVM Kernel & Examples - DataFlair*. [online] DataFlair. Available at: https://data-flair.training/blogs/svm-kernel-functions/ [Accessed 28 Dec. 2022].

- Uci.edu. (2020). *UCI Machine Learning Repository: Shill Bidding Dataset Data Set*. [online] Available at: https://archive.ics.uci.edu/ml/datasets/Shill+Bidding+Dataset# [Accessed 28 Dec. 2022].