Shill Bidding Classification with Machine Learning

by
Simone Finelli - sba22524

Higher Diploma in Science in Artificial Intelligence Applications

Machine Learning

Muhammad Iqbal

CCT College

Dublin, Ireland

Table of Contents

Approach to the project and words count

For this assessment I will work with CRISP-DM methodology and, more specifically, the following sections will be covered:

- Business Understanding
- Data Understanding
- Data Preparation
- Modeling
- Evaluation
- Deployment

Given that the assessment scoring is mostly given by the Machine Learning component, a heavier focus will be given around the Data Preparation, Modeling and Evaluation sections.

The reason why the other three steps will also be included in the report is because they are still required to get good model results. i.e., without a proper preparation the model wouldn't perform as we would like to.

The following table recaps the number of words for each section excluding titles and quotes:

Topic	Words count
Business Understanding	122
Data Understanding	302
Data Preparation	671
Modeling	1754
Evaluation	448
Deployment	158
Total	3455

Figure 1. Words count table

And the following is a pie-chart view of the previous table:

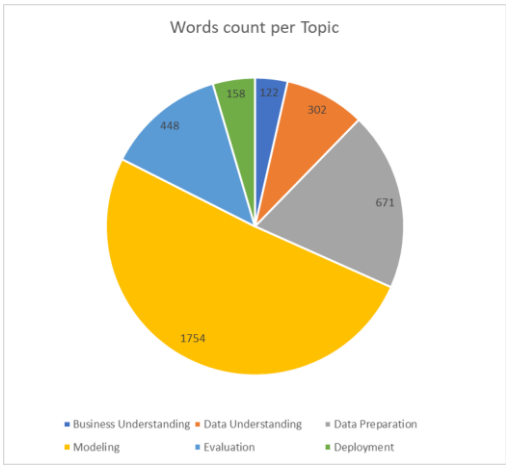

Figure 2. Words count pie-chart

Business Understanding

Business understanding is the first step in CRISP-DM. It involves gaining an understanding of the business context in which the project will be conducted. This includes understanding the business objectives of the project, the target audience, and the available resources.

- **Goal**: to predict the bids in the future, either as class 0 (normal) or 1 (anomalous). To make the goal successful we aim for a 90% accuracy with our model.
- **Audience**: it will be the CCT Staff reviewing this assignment.
- **Available resources**: the csv provided by UCI will be used as our dataset.

Python libraries that will be used:

- Pandas
- NumPy
- Matplotlib
- Seaborn
- Scikit-learn

Modelling: kNN, Random Forest and Logistic Regression for binary classification.

Data Understanding

Before importing the dataset, we recall the information we have from UCI regarding our attribute:

Attribute Information:

Record ID: Unique identifier of a record in the dataset.
Auction ID: Unique identifier of an auction.
Bidder ID: Unique identifier of a bidder.
Bidder Tendency: A shill bidder participates exclusively in auctions of few sellers rather than a diversified lot. This is a collusive act involving the fraudulent seller and an accomplice.
Bidding Ratio: A shill bidder participates more frequently to raise the auction price and attract higher bids from legitimate participants.
Successive Outbidding: A shill bidder successively outbids himself even though he is the current winner to increase the price gradually with small consecutive increments.
Last Bidding: A shill bidder becomes inactive at the last stage of the auction (more than 90\% of the auction duration) to avoid winning the auction.
Auction Bids: Auctions with SB activities tend to have a much higher number of bids than the average of bids in concurrent auctions.
Auction Starting Price: a shill bidder usually offers a small starting price to attract legitimate bidders into the auction.
Early Bidding: A shill bidder tends to bid pretty early in the auction (less than 25\% of the auction duration) to get the attention of auction users.
Winning Ratio: A shill bidder competes in many auctions but hardly wins any auctions.
Auction Duration: How long an auction lasted.
Class: 0 for normal behaviour bidding; 1 for otherwise.

Figure 3. Dataset attribute information

We now start our EDA analysis having a closer look to our dataset.

Let's import the csv data into a DataFrame, referencing “Record_ID” as the index column:

```
In [3]: # dataset import
        df = pd.read_csv("Shill Bidding Dataset.csv", index_col="Record_ID")
```

Figure 4. Dataset import

We now want to quickly explore our dataset using the head function:

```
In [4]: # return first 5 rows of the dataset
        df.head()
```

Out[4]:

Record_ID	Auction_ID	Bidder_ID	Bidder_Tendency	Bidding_Ratio	Successive_Outbidding	Last_Bidding	Auction_Bids	Starting_Price_Average	Early_Bidding	Winning_Ratio	Auction_Duration	Class
1	732	_***i	0.200000	0.400000	0.0	0.000028	0.0	0.993593	0.000028	0.666667	5	0
2	732	g***r	0.024390	0.200000	0.0	0.013123	0.0	0.993593	0.013123	0.944444	5	0
3	732	t***p	0.142857	0.200000	0.0	0.003042	0.0	0.993593	0.003042	1.000000	5	0
4	732	7***n	0.100000	0.200000	0.0	0.097477	0.0	0.993593	0.097477	1.000000	5	0
5	900	z***z	0.051282	0.222222	0.0	0.001318	0.0	0.000000	0.001242	0.500000	7	0

Figure 5. Dataset head

And return the number of columns and rows:

```
In [5]: # return number of rows and columns
        df.shape
Out[5]: (6321, 12)
```

Figure 6. Dataset shape

Let's get a better understanding of the dataset structure:

```
In [6]: # return info about the dataset
        df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 6321 entries, 1 to 15144
Data columns (total 12 columns):
 #   Column                  Non-Null Count  Dtype
---  ------                  --------------  -----
 0   Auction_ID              6321 non-null   int64
 1   Bidder_ID               6321 non-null   object
 2   Bidder_Tendency         6321 non-null   float64
 3   Bidding_Ratio           6321 non-null   float64
 4   Successive_Outbidding   6321 non-null   float64
 5   Last_Bidding            6321 non-null   float64
 6   Auction_Bids            6321 non-null   float64
 7   Starting_Price_Average  6321 non-null   float64
 8   Early_Bidding           6321 non-null   float64
 9   Winning_Ratio           6321 non-null   float64
 10  Auction_Duration        6321 non-null   int64
 11  Class                   6321 non-null   int64
dtypes: float64(8), int64(3), object(1)
memory usage: 642.0+ KB
```

Figure 7. Dataset info

And get a statistical description using the describe function:

```
In [7]: # return a statistical description
        df.describe()
```

Out[7]:

	Auction_ID	Bidder_Tendency	Bidding_Ratio	Successive_Outbidding	Last_Bidding	Auction_Bids	Starting_Price_Average	Early_Bidding	Winning_Ratio	Auction_Duration	Class
count	6321.000000	6321.000000	6321.000000	6321.000000	6321.000000	6321.000000	6321.000000	6321.000000	6321.000000	6321.000000	6321.000000
mean	1241.388230	0.142541	0.127670	0.103781	0.463119	0.231606	0.472821	0.430683	0.367731	4.615093	0.106787
std	735.770789	0.197084	0.131530	0.279698	0.380097	0.255252	0.489912	0.380785	0.436573	2.466629	0.308867
min	5.000000	0.000000	0.011765	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	1.000000	0.000000
25%	589.000000	0.027027	0.043478	0.000000	0.047928	0.000000	0.000000	0.026620	0.000000	3.000000	0.000000
50%	1246.000000	0.062500	0.083333	0.000000	0.440937	0.142857	0.000000	0.360104	0.000000	5.000000	0.000000
75%	1867.000000	0.166667	0.166667	0.000000	0.860363	0.454545	0.993593	0.826761	0.851852	7.000000	0.000000
max	2538.000000	1.000000	1.000000	1.000000	0.999900	0.788235	0.999935	0.999900	1.000000	10.000000	1.000000

Figure 8. Dataset describe

Finally, we want to doublecheck if we have null values to take into consideration:

```
In [8]: # return the sum of null values for each column
        df.isnull().sum()
```

```
Out[8]: Auction_ID              0
        Bidder_ID               0
        Bidder_Tendency         0
        Bidding_Ratio           0
        Successive_Outbidding   0
        Last_Bidding            0
        Auction_Bids            0
        Starting_Price_Average  0
        Early_Bidding           0
        Winning_Ratio           0
        Auction_Duration        0
        Class                   0
        dtype: int64
```

Figure 9. Dataset null values

Until now we can say that our dataset is:

- Made of 12 columns and 6321 rows
- Mostly made of numerical features – except Bidder_ID
- There are no null values
- Mostly made of features ranging from 0 to 1 – except Auction_ID and Auction_Duration

It’s also important to check how unbalanced our dataset is and, to do so, we will plot graph showing the count number for both classes 0 and 1.

Let's first split our dataset into X and y. X will contain all columns except the Class, y will only contain the Class:

```
In [9]:  # dataset spli between X and y
         X = df.iloc[:,:-1]
         y = df.iloc[:,-1:]
```

Figure 10. X,y generation

The following code will be used to plot on the x-axis the class and on the y-axis the number of samples:

```
In [10]: # display grapth to show class distribution across the dataset
         plt.figure(figsize=(10,10))
         sns.countplot(x="Class", data=y, order=y["Class"].unique())
         plt.xlabel("Bid class")
         plt.ylabel('Number of samples')
```

```
Out[10]: Text(0, 0.5, 'Number of samples')
```

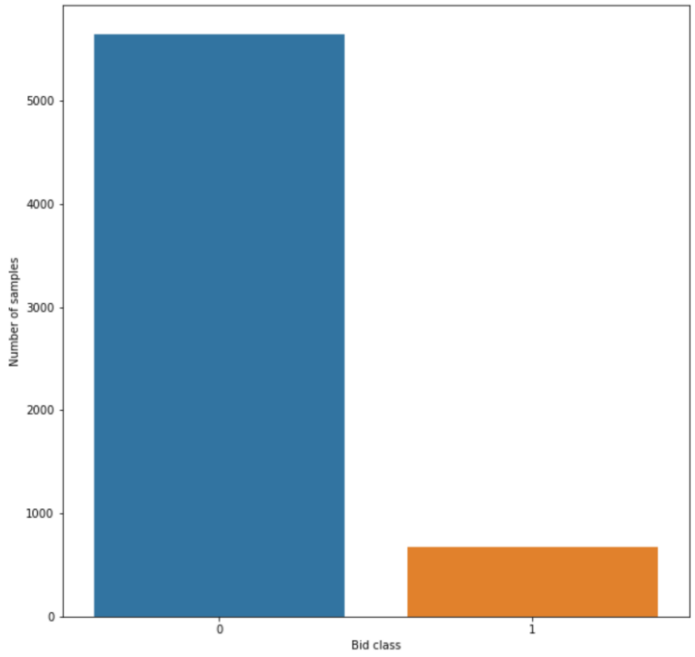

Figure 11. Class distribution

We see that more than 5000 records are classified as 0 and less than 1000 as 1. Given that this high unbalance could lead to incorrect predictions, during the data preparation phase we will make sure to balance the dataset.

Before proceeding with the data preparation phase, we want to see if features do present outliers. To do so the Box Plot graph will be used:

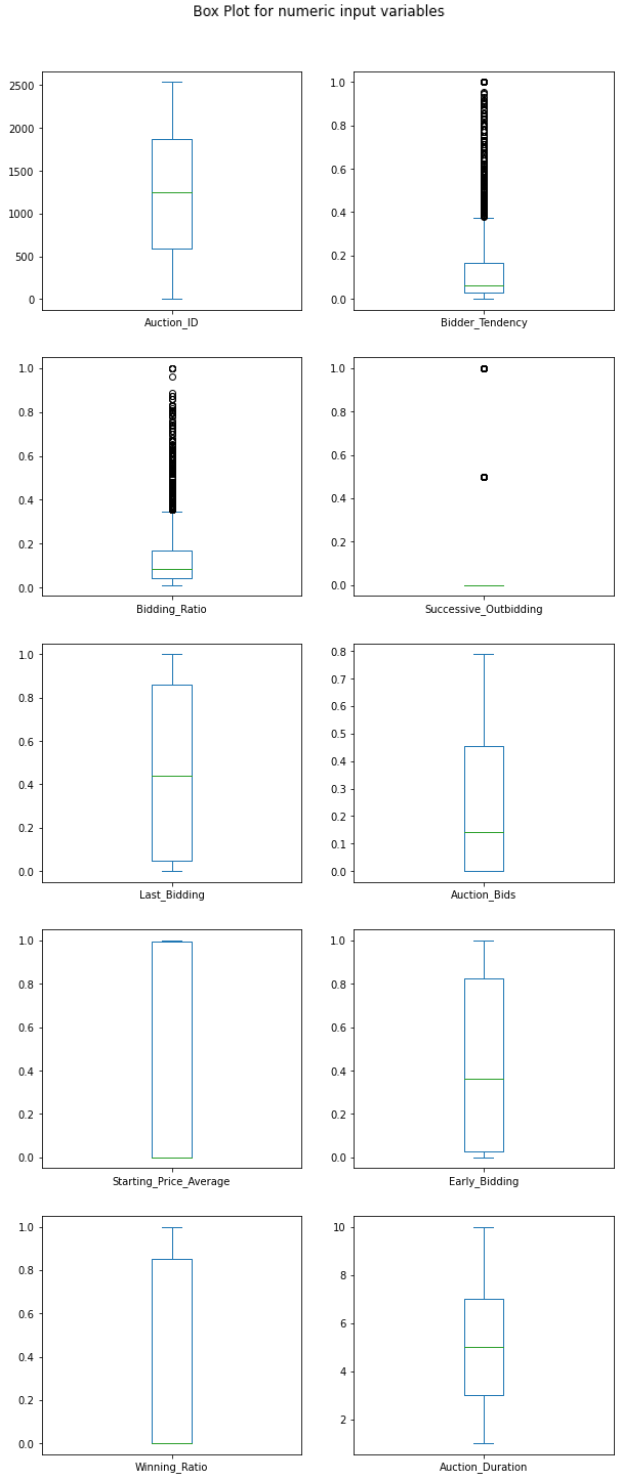

Figure 12. Features Boxplot

As a result of the Box Plot graph, we do see that Bidder_Tendency and Bidding_Ratio are the features that present the highest number of outliers.

Data Preparation

I'm assuming that Auction_ID and Bidder_ID are both variables that do not relate with the possible behaviour of a bid and, for this reason, we want to understand how important they are in our dataset.

Before to do so, during the previous phase we noticed that the Bidder_ID is the only of type Object. We now want to convert it into a categorical value and cast to an int to be able to better understand its importance.

Let's copy the Bidder_ID feature into the bidder_id dataset and cast it:

```
In [12]: # copy Bidder_ID feature into bidder_id dataset
         bidder_id = X["Bidder_ID"]

In [13]: # check the type and value of the copied feature
         bidder_id.head(), bidder_id.info()

         <class 'pandas.core.series.Series'>
         Int64Index: 6321 entries, 1 to 15144
         Series name: Bidder_ID
         Non-Null Count  Dtype
         --------------  -----
         6321 non-null   object
         dtypes: object(1)
         memory usage: 98.8+ KB

Out[13]: (Record_ID
          1    _***i
          2    g***r
          3    t***p
          4    7***n
          5    z***z
          Name: Bidder_ID, dtype: object,
          None)

In [14]: # casting of the feature from object to category and then to int
         bidder_id = bidder_id.astype('category')
         bidder_id = bidder_id.cat.codes

In [15]: # check the type and value of the manipulated feature
         bidder_id.head(), bidder_id.info()

         <class 'pandas.core.series.Series'>
         Int64Index: 6321 entries, 1 to 15144
         Series name: None
         Non-Null Count  Dtype
         --------------  -----
         6321 non-null   int16
         dtypes: int16(1)
         memory usage: 61.7 KB

Out[15]: (Record_ID
          1     302
          2     513
          3     908
          4     234
          5    1053
          dtype: int16,
          None)
```

Figure 13. "Bidder_ID" cast

Given that we were able to convert our object feature to an int, we now replace it in our X dataset:

```
In [16]: # Bidder_ID feature replacement in the original dataset
         X["Bidder_ID"] = bidder_id

In [17]: # X and y dimension verification
         X.shape, y.shape

Out[17]: ((6321, 11), (6321, 1))
```

Figure 14. "Bidder_ID" replace in X

As previously assumed, we now want to prove that Bidder_ID and Auction_ID are not relevant features when predicting the class of a bid. For this reason, we determine the features importance of our dataset to check if we can create a lighter version of it.

For this purpose, we will use Mean Decrease in Impurity (MDI) from a Random Forest Classifier. In this way, *"feature importances are provided by the fitted attribute feature_importances_ and they are computed as the mean and standard deviation of accumulation of the impurity decrease within each tree"*. (scikit-learn, n.d.)

We first define a Random Forest Classifier and train it:

```
In [18]:
# definition of the rfc
forest = RandomForestClassifier(random_state=0)
# rfc fit
forest.fit(X, y)

Out[18]:
RandomForestClassifier(random_state=0)
```

Figure 15. Random Forest Classifier train

We then extract the MDI importances using the feature_importances_ function and plot our results:

```
In [19]:
# extraction of important features mapped to input dataset columns, organized in ascending order
mdi_importances = pd.Series(forest.feature_importances_, index=X.columns).sort_values(ascending=True)
# graph plotting
ax = mdi_importances.plot.barh()
ax.set_title("Random Forest Feature Importances (MDI)")
ax.figure.tight_layout()
```

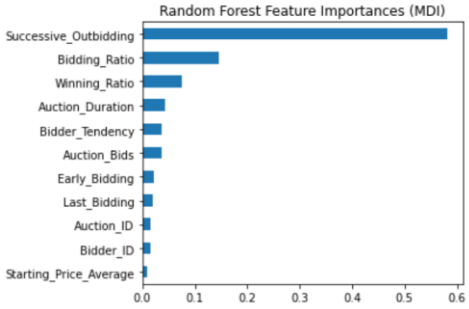

Figure 16. Features importance

As correctly guessed, Auction_ID and Bidder_ID result as low important in our dataset, for this reason we just proceed to drop them:

```
In [20]:
# we define the list of columns we want to keep
columns = ['Bidder_Tendency', 'Bidding_Ratio',
       'Successive_Outbidding', 'Last_Bidding', 'Auction_Bids',
       'Starting_Price_Average', 'Early_Bidding', 'Winning_Ratio',
       'Auction_Duration']

In [21]:
# we replace our X dataset with a subset of features as previously determined
X = X[columns]
```

Figure 17. Unimportant features drop

Another important thing that we noticed in our data understanding phase is that most of the variables do range from 0 to 1 but Auction_Duration does not. To improve modelling performance end efficiency, we want to scale our dataset so that all variables are ranging from 0 to 1.

To do so, MinMaxScaler method will be used.

Please note that we want to have a portion of our dataset that is presenting a real-world scenario. For this reason, the scaling will be performed only on the train dataset and not the test one.

We now proceed with the splitting of our dataset, using a 70% proportion for the training and a 30% one for the test:

```
In [22]:
# dataset splitting
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3, random_state = 42)
```

Figure 18. Dataset Train-Test split

The 70-30 portion has been used to make sure that we could have a good number of class 1 examples for our training.

We now apply MinMaxScaler method to make our features ranging from 0 to 1:

```
In [23]:
# feature normalization
sc = MinMaxScaler()
X_train = sc.fit_transform(X_train)
# X train dataset is reconverted into again into a Dataframe
X_train = pd.DataFrame(X_train)

# MinMaxScaler looses the actual name of features, we now rename them
for i in X_train:
    X_train = X_train.rename(columns={i: columns[i]})
```

Figure 19. Dataset scaling

Now that the normalization on the input training dataset has happened, we want to graphically visualize the result we got:

```
In [24]:
# feature distribution plotting
plt.figure(figsize=(20,10))
sns.boxplot(data=X_train)
plt.xlabel("Feature")
plt.ylabel("Distribution")

Out[24]: Text(0, 0.5, 'Distribution')
```

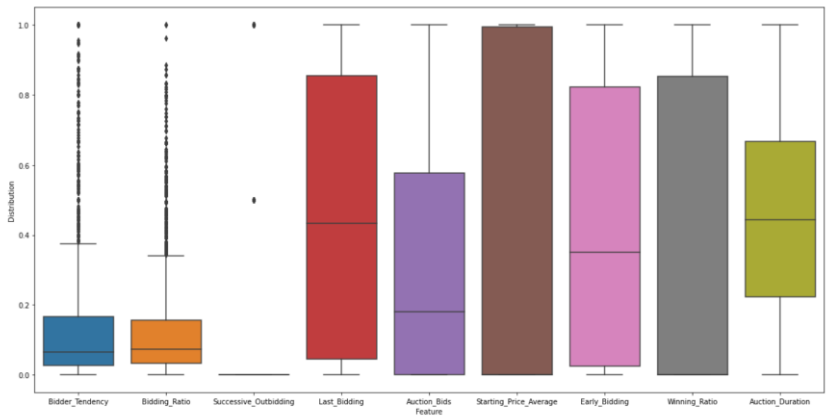

Figure 20. Scaled features visualization

As we can see on the previous graph, all variables are now in the range 0 to 1 - meaning that the standardization did work. Unfortunately, we still see outliers visible in the dataset.

Finally, we remember that we have a high unbalanced dataset - meaning that most of our rows are classified as type 0. This means that if we don't balance our training dataset to have a similar number of class 0 and 1 example, we will probably be overfitting towards the class 0. In this scenario our model will be very good at predicting the class 0 samples but not the class 1 ones (the one we are more interested in).

As discussed during our classes, there are two ways to handle unbalanced datasets:

- SMOTE
- Near Miss

"SMOTE is an over-sampling method. What it does is, it creates synthetic (not duplicate) samples of the minority class.

NearMiss is an under-sampling technique. This will make the majority class equal to minority class." (Rahim, 2018)

Before deciding on which model to try out, let's check how many samples of class 0 and 1 we have in our training dataset:

```
In [25]: # check class counts
         y_train.value_counts()

Out[25]: Class
         0        3939
         1         485
         dtype: int64
```

Figure 21. Train dataset classes distribution

If we use the SMOTE method, we will have 3939 samples for both class 0 and 1.

If we use the Near Miss method, we will have 485 samples for both class 0 and 1.

For the sake of simplicity, we will work with Near Miss methodology. This is mainly because during the modelling phase we will have to perform different trainings based on hyperparameter selection and we do not want to overload our hardware performance.

Let's apply the Near Miss method to rebalance our dataset:

```
In [26]: #smt = SMOTE()
         #X_train, y_train = smt.fit_resample(X_train, y_train)

         # method to balance our dataset
         nr = NearMiss()
         X_train, y_train = nr.fit_resample(X_train, y_train)
```

Figure 22. Dataset balancing

Let's now double check the number of classes 0 and 1 available to us:

```
In [27]: # check class counts
         y_train.value_counts()

Out[27]: Class
         0        485
         1        485
         dtype: int64
```

Figure 23. Classes distribution after balancing

Modelling

“The supervised learning approach in ML uses labeled datasets that train algorithms to classify data or predict outputs precisely. The model uses the labeled data to measure the relevance of different features to gradually improve model fit to the known outcome.” (www.alteryx.com, n.d.)

On the other hand, *“in unsupervised learning, ML algorithms are used to examine and group unlabeled datasets. Such algorithms can uncover unknown patterns in data without human supervision”*. (www.alteryx.com, n.d.)

In our case, since we already have a dataset with a target value Class, that is either 0 or 1, we will proceed with a supervised learning approach.

It's worth noting that we could also potentially use an unsupervised learning approach as a pre-processing step to better understand features relationship from our dataset before applying a supervised learning algorithm. However, I feel that Supervised Learning is the best approach to start with given the already available target value.

The following algorithms will be used and compared:

- kNN
- Random forest
- Logistic Regression

From the models we studied, also linear SVM could be used. Differently from Logistic Regression, it tries to find the “best” margin that separates the classes to reduce the risk of error on the data. It might be a very good choice given that it is less prone to outliers as it only cares about points that are close to the decision boundary.

Given that we are working with a classification problem, I believe that Logistic Regression could be an easier implementation.

KNN

k-nearest neighbours (k-NN) is a supervised learning algorithm that is used for classification and regression tasks. It is based on the idea of determining the class of a data point based on the class of its nearest neighbours.

The value of k, which represents the number of nearest neighbours to use, is chosen by the user and represents the hyperparameter to be tuned.

k-NN has the advantage of being simple to implement and able to be used for both classification and regression tasks, but it can be computationally expensive and may not scale well to large datasets.

Given that the only hyperparameter that must be selected, we will run the model using different values of k ranging from 0 to 100. We will then look at the test and training score results to determine which value of k returns a better result.

We start to define an array containing all possible values of k that will be used, and two matrixes (train_accuracy and test_accuracy) that will be used to store the accuracy results for the k^{th} value:

```
In [28]:
neighbors = np.arange(1, 100, 1)              # number of neighbors
train_accuracy = np.zeros(len(neighbors))     # Declare and initialise the matrix
test_accuracy = np.zeros(len(neighbors))      # Declare and initialise the matrix
```

Figure 24. Array definition for kNN

We now perform our model looping over possible k values. We then store the test and train accuracy results in our matrix for later analysis:

```
In [29]:
for i, k in enumerate(neighbors):                          # for loop that checks the model for neighbor values 1,2,3,...,10
    kNN = KNeighborsClassifier(n_neighbors = k)            # Initialise an object knn using KNeighborsClassifier method

    # Fit the model
    kNN.fit(X_train, y_train)                              # Call fit method to implement the ML KNeighborsClassifier model

    # Compute accuracy on the training set
    train_accuracy[i] = kNN.score(X_train, y_train)        # Save the score value in the train_accuracy array

    # Note: we are not getting 100% on the training dataset and it is normal when k is different from 1
    # here we are performing a score of each model on the training dataset

    # Compute accuracy on the test set
    test_accuracy[i] = kNN.score(X_test, y_test)           # Save the score value in the train_accuracy array
    #print(str(k) + " " + str(train_accuracy[i]) + " " + str(test_accuracy[i]))
```

Figure 25. kNN training over different Ks

At this point we want to graphically see the results. We aim at finding a value of k where training and test accuracy are similar, meaning that we are not overfitting or underfitting with our training and test datasets:

```
In [30]:
# X axis represents the value of k on which the model is trained, y axis represents the scoring result.
# This is a two line graph where blue is testing and orange is training
plt.figure(figsize = (15, 6))
plt.title('KNN accuracy with varying number of neighbors', fontsize = 20)
plt.plot(neighbors, test_accuracy, label = 'Testing Accuracy')
plt.plot(neighbors, train_accuracy, label = 'Training accuracy')
plt.legend(prop={'size': 20})
plt.xlabel('Number of neighbors', fontsize = 20)
plt.ylabel('Accuracy', fontsize = 20)
plt.xticks(np.arange(1, max(neighbors), 5), fontsize = 20)
#plt.xticks(fontsize = 20)
plt.yticks(fontsize = 20)
plt.grid()
plt.show()

'''
with k=10 we are not underfitting or overfitting given the fact that the performance of training and test model
are similar
'''
```

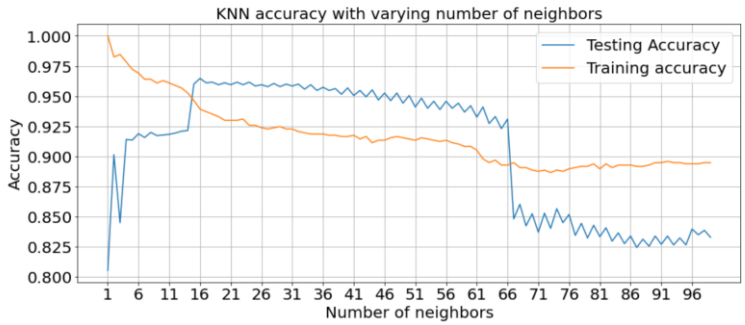

Figure 26. kNN accuracy graph for k from 1 to 100

From previous graph we notice that the two lines cross for the k value between 11-16 and 66-71. Given that accuracy results are better in the range 11-16, we want to look at the k value in that range.

Let's plot again our graph stretching the area 11-21:

```
In [31]:
# X axis represents the value of k on which the model is trained, y axis represents the scoring result.
# This is a two line graph where blue is testing and orange is training
plt.figure(figsize = (15, 6))
plt.title('KNN accuracy with varying number of neighbors', fontsize = 20)
plt.plot(np.arange(11,21), test_accuracy[10:20], label = 'Testing Accuracy')
plt.plot(np.arange(11,21), train_accuracy[10:20], label = 'Training accuracy')
plt.legend(prop={'size': 20})
plt.xlabel('Number of neighbors', fontsize = 20)
plt.ylabel('Accuracy', fontsize = 20)
plt.xticks(np.arange(11, 21, 1), fontsize = 20)
#plt.xticks(fontsize = 20)
plt.yticks(fontsize = 20)
plt.grid()
plt.show()
```

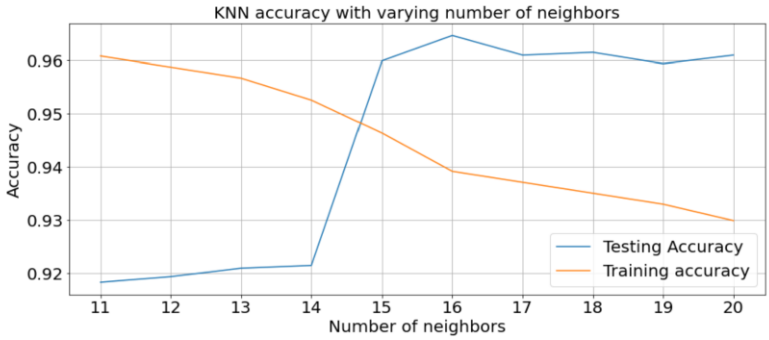

Figure 27. kNN accuracy graph for k from 11 to 20

The result for k=14 shows that we are clearly underfitting with the testing and overfitting with the training, the other way around happens with k=16. For this

reason, we pick up 15 as the value of k. We now proceed with a model training for that value to extract:

- Testing and training accuracy
- Confusion Matrix
- Classification report

Let's start with the training of the model for k=15:

```
In [32]:
# Create KNN Classifier
kNN = KNeighborsClassifier(n_neighbors = 15)

# Train the model using the training sets
kNN.fit(X_train, y_train)

# Predict the response for test dataset
y_pred = kNN.predict(X_test)
```

Figure 28. kNN training for optimal k=15

Let's now get our model accuracy:

```
In [33]:
# Model Accuracy
print('Training score: {:.3f}'.format(kNN.score(X_train,y_train)))
print('Testing score: {:.3f}'.format(kNN.score(X_test,y_test)))

Training score: 0.946
Testing score: 0.960
```

Figure 29. kNN accuracy for optimal k=15

Let's now plot our confusion matrix:

```
In [34]:
ax = plt.subplot()

# Calculate cm by calling a method named as 'confusion_matrix'
cm = confusion_matrix(y_test, y_pred)

# Call a method heatmap() to plot confusion matrix
sns.heatmap(cm, annot = True)

# Define x, y and title
ax.set_xlabel('Predicted labels')
ax.set_ylabel('True labels')
ax.set_title('Confusion Matrix')

Out[34]: Text(0.5, 1.0, 'Confusion Matrix')
```

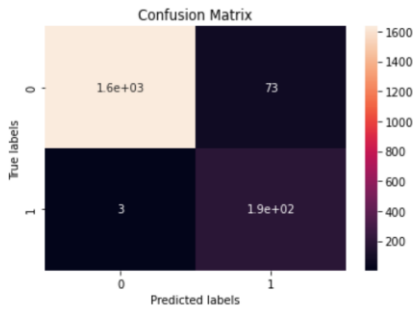

Figure 30. kNN confusion matrix

The matrix compares the actual target values with those predicted by the machine learning model. This gives us a holistic view of how well our classification model is performing and what kinds of errors it is making.

In our case we got the following results:

- True Negative (1.6e+03) - Class 0 correctly classified.
- False Positive (73) - Class 0 classified as 1.
- False Negative (3) - Class 1 classified as 0.
- True Positive (1.9e+02) - Class 1 correctly classified.

Looking at just our accuracy result we might say that we built a very good model given that we are able to correctly predict the bid class - either as 0 or 1 - 96% of times.

If we think about it, this is true but we are working with an high unbalanced training dataset where again we are very good at detecting normal behaviour bidding. Class 1 bidding are the one we are most interested in, from the plotted confusion matrix we see that only 3 anomalous bids were wrongly predicted.

But what percentage is that? Let's see it building a classification report:

```
In [35]: # Display the classification report
         print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

           0       1.00      0.96      0.98      1707
           1       0.72      0.98      0.83       190

    accuracy                           0.96      1897
   macro avg       0.86      0.97      0.90      1897
weighted avg       0.97      0.96      0.96      1897
```

Figure 31. kNN classification report

The classification report tells us that we have 1707 class 0 samples and 190 class 1 samples.

Given that we want to make sure that we can both identify 0 and 1 classes, we look at the f-1 score as a metric of evaluation where we see that we have a value of 98% for class 0 and 83% for class 1.

So far, we are happy with the results we got, let's try to see our ROC Curve.

A ROC curve is a plot that shows the performance of a binary classifier model. The x-axis shows the false positive rate, and the y-axis shows the true positive rate.

Let's plot our ROC Curve:

In [36]:
```
# ROC Curve plotting
from sklearn.metrics import roc_curve

y_pred_proba = kNN.predict_proba(X_test)[:,1]

fpr, tpr, thresholds = roc_curve(y_test, y_pred_proba)

plt.plot([0,1],[0,1],'k--')
plt.plot(fpr,tpr, label='Knn')
plt.xlabel('fpr')
plt.ylabel('tpr')
plt.title('Knn(n_neighbors=15) ROC curve')
plt.show()
```

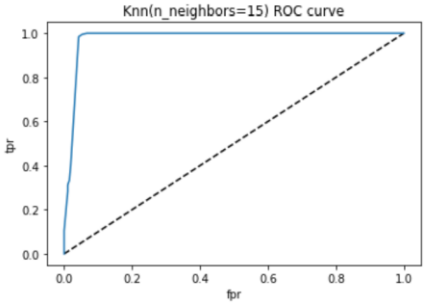

Figure 32. kNN ROC curve

And let's compute our AUC score:

In [37]:
```
#Area under ROC curve
from sklearn.metrics import roc_auc_score
roc_auc_score(y_test,y_pred_proba)
```

Out[37]:
```
0.9783630869793112
```

Figure 33. kNN AUC score

Random Forest

Random forest is an ensemble learning method used for classification and regression tasks. It is a type of decision tree algorithm that involves constructing many decision trees, each trained on a randomly selected subset of the training data and combining the predictions of these decision trees to make a final prediction.

Random forest is less prone to overfitting than a single decision tree and can be used for both classification and regression tasks, but it can be computationally expensive and may not scale well to very large datasets. Given that our dataset is not that big, we decided to test this algorithm.

Before starting, let's recall that an estimator is a decision tree that is trained on a subset of our training data. In our case we are creating 1000 of them and then averaging the performance results:

```
In [45]: #Create a Gaussian Classifier
         rfc = RandomForestClassifier(n_estimators = 1000)

         #Train the model using the training sets y_pred=clf.predict(X_test)
         rfc.fit(X_train, y_train)

         y_pred = rfc.predict(X_test)
```

Figure 34. Random Forest Classifier initialization and training

Let's now check our test and training accuracy:

```
In [46]: # Model Accuracy
         print('Training score: {:.3f}'.format(rfc.score(X_train,y_train)))
         print('Testing score: {:.3f}'.format(rfc.score(X_test,y_test)))

         Training score: 1.000
         Testing score: 0.937
```

Figure 35. Random Forest Classifier accuracy

From previous results, it is quite clear that we are overfitting towards our training dataset. Even though our testing accuracy might still look great, let's try to see the confusion matrix:

```
In [47]: ax = plt.subplot()

         # Calculate cm by calling a method named as 'confusion_matrix'
         cm = confusion_matrix(y_test, y_pred)

         # Call a method heatmap() to plot confusion matrix
         sns.heatmap(cm, annot = True)

         # Define x, y and title
         ax.set_xlabel('Predicted labels')
         ax.set_ylabel('True labels')
         ax.set_title('Confusion Matrix')

Out[47]: Text(0.5, 1.0, 'Confusion Matrix')
```

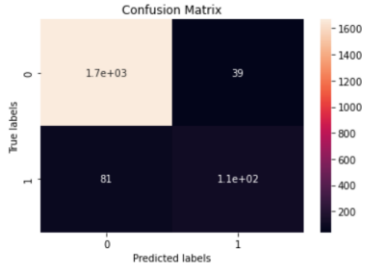

Figure 36. Random Forest Classifier confusion matrix

And our classification report:

```
In [48]: # Display the classification report
         print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

           0       0.95      0.98      0.97      1707
           1       0.74      0.57      0.64       190

    accuracy                           0.94      1897
   macro avg       0.85      0.78      0.81      1897
weighted avg       0.93      0.94      0.93      1897
```

Figure 37. Random Forest Classifier classification report

Without properly tuning our Random Forest Classifier we immediately see that, with respect to kNN, we are performing better in reducing False Positive, but we are also increasing False Negative.

Please note that instead of reducing the Flase Positive we are more interested in reducing False Negative, not increasing them. This is because it's better to have a normal bid (class 0) identified as potentially anomalous (class 1) instead of just ignoring an anomalous bid (class 1) as ignored and identified as a normal one (class 0).

For this reason, let's now try to increase our Random Forest Classifier using GridSearchCV.

To perform grid search CV, we must specify a set of values for each hyperparameter, and the algorithm generates all possible combinations of these values. For each combination of hyperparameter values, the model is trained and evaluated using cross-validation. The combination of hyperparameter values that yields the best performance is retained.

Let's not start with the definition of our hyperparameters:

```
In [49]: # Hyper parameter definition
         param_grid = {
             'n_estimators': [50, 100, 200],
             'max_features': ['auto', 'sqrt', 'log2'],
             'max_depth' : [4,5,6,7,8],
             'criterion' :['gini', 'entropy']
         }
```

Figure 38. Random Forest Classifier hyperparameters grid definition

And let's build our Cross Validation model using our training dataset:

```
In [50]:
# cross validation model building
rfc = RandomForestClassifier(random_state=42)
CV = GridSearchCV(estimator=rfc, param_grid=param_grid)
CV.fit(X_train, y_train)

Out[50]:
GridSearchCV(estimator=RandomForestClassifier(random_state=42),
             param_grid={'criterion': ['gini', 'entropy'],
                         'max_depth': [4, 5, 6, 7, 8],
                         'max_features': ['auto', 'sqrt', 'log2'],
                         'n_estimators': [50, 100, 200]})
```

Figure 39. Random Forest Classifier hyperparameters identification using GridSearchCV

We now want to print out the best hyperparameters identified:

```
In [51]:
# print best parameters
CV.best_params_

Out[51]:
{'criterion': 'entropy',
 'max_depth': 7,
 'max_features': 'auto',
 'n_estimators': 200}
```

Figure 40. Random Forest Classifier best hyperparameters

At this point we retrain our Random Forest Classifier using those hyperparameters, and see if we can improve our results:

```
In [52]:
# random forest classifier training with best hyperparameters
rfc = RandomForestClassifier(random_state=42, max_features='auto', n_estimators= 200, max_depth=7, criterion='entropy')
rfc.fit(X_train, y_train)
y_pred = rfc.predict(X_test)
```

Figure 41. Random Forest Classifier training with best hyperparameters

Let's check our test and training accuracy:

```
In [53]:
# Model Accuracy
print('Training score: {:.3f}'.format(rfc.score(X_train,y_train)))
print('Testing score: {:.3f}'.format(rfc.score(X_test,y_test)))

Training score: 0.996
Testing score: 0.948
```

Figure 42. Random Forest Classifier with best hyperparameters accuracy

From previous results we see that we are getting worst accuracy results with respect to kNN. But we also want to check how it is our confusion matrix:

```
In [54]:
ax = plt.subplot()

# Calculate cm by calling a method named as 'confusion_matrix'
cm = confusion_matrix(y_test, y_pred)

# Call a method heatmap() to plot confusion matrix
sns.heatmap(cm, annot = True)

# Define x, y and title
ax.set_xlabel('Predicted labels')
ax.set_ylabel('True labels')
ax.set_title('Confusion Matrix')

Out[54]: Text(0.5, 1.0, 'Confusion Matrix')
```

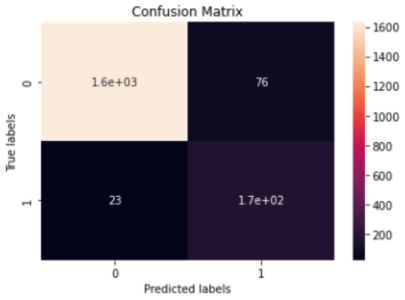

Figure 43. Random Forest Classifier with best hyperparameters confusion matrix

We can definitely see some improvements by setting hyperparameters but not getting good results as the ones returned by kNN. We still have many false negatives that we don't want.

Let's see our classification report:

```
In [55]:
# Display the classification report
print(classification_report(y_test, y_pred))
```

	precision	recall	f1-score	support
0	0.99	0.96	0.97	1707
1	0.69	0.88	0.77	190
accuracy			0.95	1897
macro avg	0.84	0.92	0.87	1897
weighted avg	0.96	0.95	0.95	1897

Figure 44. Random Forest Classifier with best hyperparameters classification report

Indeed, the f1-scorefor class 1 is improved but it is not as good as the one produced by kNN. We should be able to graphycally see this by plotting our ROC Curve:

```
In [56]:
# ROC Curve plotting
from sklearn.metrics import roc_curve

y_pred_proba = rfc.predict_proba(X_test)[:,1]

fpr, tpr, thresholds = roc_curve(y_test, y_pred_proba)

plt.plot([0,1],[0,1],'k--')
plt.plot(fpr,tpr, label='RFC')
plt.xlabel('fpr')
plt.ylabel('tpr')
plt.title('Random Forest with best param ROC curve')
plt.show()
```

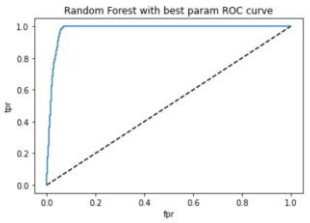

Figure 45. Random Forest Classifier with best hyperparameters ROC curve

Let's check our AUC score:

```
In [57]:
#Area under ROC curve
roc_auc_score(y_test,y_pred_proba)

Out[57]: 0.9807233373415966
```

Figure 46. Random Forest Classifier with best hyperparameters AUC score

We now want to store the result we got for further analysis:

```
In [58]:
# dict storing of knn results
rfc_result = {
    "training_score" : rfc.score(X_train,y_train),
    "test_score": rfc.score(X_test,y_test),
    "cm": cm,
    "classification_report": classification_report(y_test, y_pred),
    "roc_auc_score": roc_auc_score(y_test,y_pred_proba)
}
```

Figure 47. Random Forest Classifier with best hyperparameters results saving

Logistic Regression

Logistic regression is a statistical method for binary classification that is used to predict the probability of an instance belonging to a particular class. It uses a logistic function to map the input data to a range between 0 and 1, with the output interpreted as the probability of the input data belonging to the positive class.

It is simple to implement and interpret, but it is limited to linear decision boundaries and may not perform well on non-linearly separable data. It is also sensitive to the presence of outliers in the data.

Before trying to optimize our hyperparameters, let's try to just execute our Logistic Regression model as it is:

```
In [59]:
# Logistic regression definition and training
lr = LogisticRegression(random_state=42)
lr.fit(X_train, y_train)

y_pred = lr.predict(X_test)
```

Figure 48. Logistic Regression initialization and training

And let's print our model accuracy:

```
In [60]:
# Model Accuracy
print('Training score: {:.3f}'.format(rfc.score(X_train,y_train)))
print('Testing score: {:.3f}'.format(rfc.score(X_test,y_test)))

Training score: 0.996
Testing score: 0.948
```

Figure 49. Logistic Regression accuracy

As it exactly happened with Random Forest, it seems that we are overfitting with the training dataset being very good at classifying normal behaviour bids but not anomalous ones. To cofirm this let's see what the confusion matrix returns:

```
In [61]:
ax = plt.subplot()

# Calculate cm by calling a method named as 'confusion_matrix'
cm = confusion_matrix(y_test, y_pred)

# Call a method heatmap() to plot confusion matrix
sns.heatmap(cm, annot = True)

# Define x, y and title
ax.set_xlabel('Predicted labels')
ax.set_ylabel('True labels')
ax.set_title('Confusion Matrix')

Out[61]: Text(0.5, 1.0, 'Confusion Matrix')
```

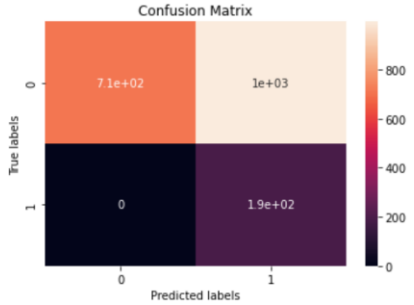

Figure 50. Logistic Regression confusion matrix

Surprisingly, we see that we have no False Negative - that is our ideal case. On the other side we have way more False Positive - meaning that often normal bid would be identified as anomalous. At this point, the classification report should give us a high recall and low precision for class 1. Let's check it:

```
In [62]: # Display the classification report
         print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

           0       1.00      0.42      0.59      1707
           1       0.16      1.00      0.28       190

    accuracy                           0.47      1897
   macro avg       0.58      0.71      0.43      1897
weighted avg       0.92      0.47      0.56      1897
```

Figure 51. Logistic Regression classification report

From the profile report we also see that recall is quite low for class 0 because, as previously said, often we are classifying bids of type 0 to bids of type 1.

Let's see if using GridSearchCV we can improve our results. Let's start with the hyperparameters definition:

```
In [63]: param_grid = {
             'solver': ['newton-cg', 'lbfgs', 'liblinear', 'sag', 'saga'],
             'C': [0.01, 0.1, 1, 2, 10, 100],
             'penalty': ['none','l1','l2']
         }
```

Figure 52. Logistic Regression hyperparameters identification using GridSearchCV

Let's train our Cross Validation model using previously defined hyperparameters:

```
In [64]: #grid = dict(solver=solvers,penalty=penalty,C=c_values)
         #grid_search = GridSearchCV(estimator=logistic_regression, param_grid=grid, n_jobs=-1, scoring='accuracy',error_score=0)

         # cross validation model building
         CV = GridSearchCV(estimator=lr, param_grid=param_grid)
         CV.fit(X_train, y_train)
```

```
Out[64]: GridSearchCV(estimator=LogisticRegression(random_state=42),
                      param_grid={'C': [0.01, 0.1, 1, 2, 10, 100],
                                  'penalty': ['none', 'l1', 'l2'],
                                  'solver': ['newton-cg', 'lbfgs', 'liblinear', 'sag',
                                             'saga']})
```

Figure 53. Logistic Regression best hyperparameters identification using GridSearchCV

Let's print the best hyperparameters found:

```
In [65]: # print best parameters
         CV.best_params_
```

```
Out[65]: {'C': 0.01, 'penalty': 'l1', 'solver': 'liblinear'}
```

Figure 54. Logistic Regression best hyperparameters

And retraining our Logistic Regression model using those:

```
In [66]:
# Logistic regression training with best hyperparameters
lr = LogisticRegression(C=0.01, penalty="l1", solver="liblinear")
lr.fit(X_train, y_train)
y_pred = lr.predict(X_test)
```

Figure 55. Regression training with best hyperparameters

Let's check our testing and training accuracy:

```
In [67]:
# Model Accuracy
print('Training score: {:.3f}'.format(lr.score(X_train,y_train)))
print('Testing score: {:.3f}'.format(lr.score(X_test,y_test)))
```

```
Training score: 0.929
Testing score: 0.968
```

Figure 56. Regression with best hyperparameters accuracy

We are now having very similar results to the ones we were getting with kNN. Here given that the delta between training and test score is higher than kNN, it might be a sign that we are not performing as good as we want but let's doublecheck our confusion matrix:

```
In [68]:
ax = plt.subplot()

# Calculate cm by calling a method named as 'confusion_matrix'
cm = confusion_matrix(y_test, y_pred)

# Call a method heatmap() to plot confusion matrix
sns.heatmap(cm, annot = True)

# Define x, y and title
ax.set_xlabel('Predicted labels')
ax.set_ylabel('True labels')
ax.set_title('Confusion Matrix')
```

```
Out[68]: Text(0.5, 1.0, 'Confusion Matrix')
```

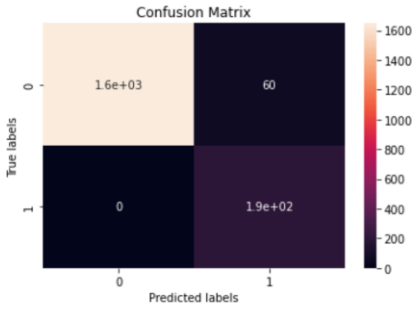

Figure 57. Regression with best hyperparameters confusion matrix

Indeed, we are still keeping False negative to zero, but we also greatly reduced the number of False positive.

Let's try to better understand our results using the classification report:

```
In [69]: # Display the classification report
         print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

           0       1.00      0.96      0.98      1707
           1       0.76      1.00      0.86       190

    accuracy                           0.97      1897
   macro avg       0.88      0.98      0.92      1897
weighted avg       0.98      0.97      0.97      1897
```

Figure 58. Regression with best hyperparameters classification report

We were able to very well increase both precision and recall for class 1. Let's see how the ROC Curve looks like:

```
In [70]: # ROC Curve plotting
         from sklearn.metrics import roc_curve

         y_pred_proba = rfc.predict_proba(X_test)[:,1]

         fpr, tpr, thresholds = roc_curve(y_test, y_pred_proba)

         plt.plot([0,1],[0,1],'k--')
         plt.plot(fpr,tpr, label='LR')
         plt.xlabel('fpr')
         plt.ylabel('tpr')
         plt.title('Logistic Regression with best param ROC curve')
         plt.show()
```

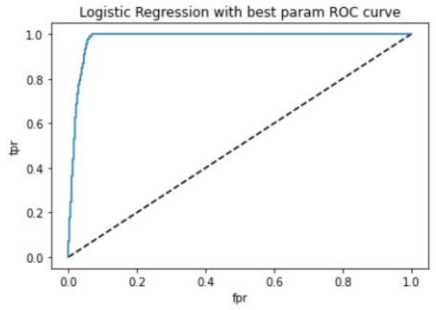

Figure 59. Regression with best hyperparameters ROC curve

And our AUC Score:

```
In [71]: #Area under ROC curve
         from sklearn.metrics import roc_auc_score
         roc_auc_score(y_test,y_pred_proba)
```

```
Out[71]: 0.9807233373415966
```

Figure 6. Regression with best hyperparameters AUC score

We now want to store the result we got for further analysis:

```
In [72]: # dict storing of knn results
         lr_result = {
             "training_score" : lr.score(X_train,y_train),
             "test_score": lr.score(X_test,y_test),
             "cm": cm,
             "classification_report": classification_report(y_test, y_pred),
             "roc_auc_score": roc_auc_score(y_test,y_pred_proba)
         }
```

Figure 60. Regression with best hyperparameters results saving

Evaluation

During the evaluation phase, a number of metrics will be used to assess the quality of the models. These metrics are accuracy, precision, recall, f1-score and AUC (area under the curve).

Let's now print out in a single table the results returned by the three models we have been working with.

Metric	kNN	Random Forest	Logistic Regression
Training Score	0.946	0.995	0.928
Test score	0.959	0.947	0.968
True Negative	1634.000	1631.000	1647.000
False Positive	73.000	76.000	60.000
False Negative	3.000	23.000	0.000
True Positive	187.000	167.000	190.000
Class 0 Precision	1.000	0.990	1.000
Class 0 Recall	0.960	0.960	0.960
Class 0 F1-score	0.980	0.970	0.980
Class 1 Precision	0.720	0.690	0.760
Class 1 Recall	0.980	0.880	1.000
Class 1 F1-score	0.830	0.770	0.860
ROC AUC Score	0.978	0.980	0.980

Figure 61. Evaluation table for models comparison

Given that this problem must be considered as an “Anomaly Detection” one, we are not really interested in looking only at the accuracy (Test score) but instead at the confusion matrix.

Let’s take into consideration results that we got from kNN. Our 0.959 Test score tells us that we can predict with a 96% accuracy our class on unsees data. This result was got since our test data is made of 1707 class 0 samples and 190 class 1 samples:

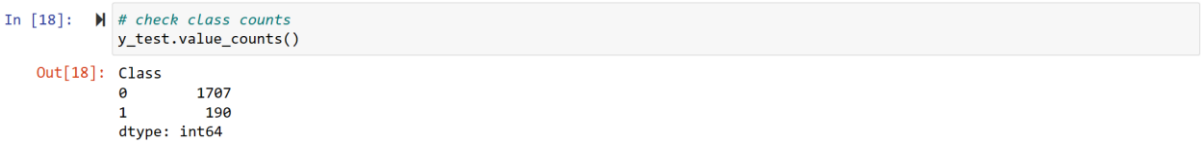

Figure 62. Class distribution on test dataset

This means that if we were able to only classify class 0 samples and not class 1 ones, we would still get a (1707*100)/1897 = 89.98% accuracy. Even though this seems a good, result, we are still not accomplishing our objective from Business Understanding.

For this reason we want to look at the model that got highest results in terms of Precision, Recall, F1-Score (for both classes) and ROC-AUC Score.

Precision is *"the ability of a classification model to return only the data points in a class"*. A high precision score indicates that the model is good at identifying the positive class, but it does not consider the number of false negatives (predictions of the negative class when the actual label is positive). (Will Koehrsen, 2022)

Recall is "*the ability of a classification model to identify all data points in a relevant class"*. A high recall score indicates that the model can identify most of the positive instances, but it does not take into account the number of false positives (predictions of the positive class when the actual label is negative). (Will Koehrsen, 2022)

The F1-score is *"a single metric that combines recall and precision using the harmonic mean"*. A high F1 score indicates that the model has both high precision and high recall. (Will Koehrsen, 2022)

The area under the curve (AUC) is a metric that summarizes the performance of a classification model - the closer to 1 the better it is.

If we look at the comparison table, Logistic Regression is clearly the model that has the best scores for all metrics. For this reason, we must say that it is the best model that was built.

Let's quickly recall the confusion matrix:

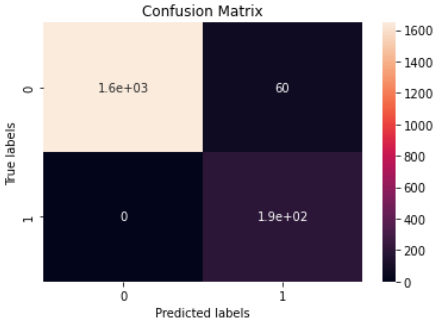

Figure 63. Best model confusion matrix

This tells us that when a class 1 bid is generated, it will always be detected by the model (100% recall). In some cases, it might happen that a class 0 is identified as 1 (it happened 60 times in our test dataset).

In simple words, it's better to have a model that sometimes identify a normal bid as an anomalous one for further analysis instead of a model that sometime classifies anomalous bids as normal ones.

Deployment

My advice for the deployment of the model is to use cloud services where the following advantages can be seen:

- Scalability allows you to scale your deployment to handle a large number of predictions.
- High availability ensures that your deployment is highly available, so you don't have to worry about your model being unavailable to your users.
- Security and compliance allows you to secure your deployed models and meet compliance requirements by using features such as Azure Active Directory and Virtual Networks.
- Monitoring and Management provides tools for monitoring and managing deployed models, so you can keep track of their performance and make updates as needed
- Provide MLOps capabilities, such as automatic model versioning, model lineage and reproducibility, and rollout and rollback of new models.

Overall, a Cloud provider makes it easy to deploy and manage machine learning models, so you can focus on building great models rather than worrying about the infrastructure.

Reference List

scikit-learn. (n.d.). *Feature importances with a forest of trees*. [online] Available at: https://scikit-learn.org/stable/auto_examples/ensemble/plot_forest_importances.html.

Rahim, S.A. (2018). SMOTE AND NEAR MISS IN PYTHON: MACHINE LEARNING IN IMBALANCED DATASETS. [online] Medium. Available at: https://medium.com/@saeedAR/smote-and-near-miss-in-python-machine-learning-in-imbalanced-datasets-b7976d9a7a79.

www.alteryx.com. (n.d.). *Supervised vs. Unsupervised Learning; Which Is Best? | Alteryx*. [online] Available at: https://www.alteryx.com/glossary/supervised-vs-unsupervised-learning#:~:text=Supervised%20and%20unsupervised%20learning%20have.

Will Koehrsen (2022). When Accuracy Isn't Enough, Use Precision and Recall to Evaluate Your Classification Model. [online] Available at: https://builtin.com/data-science/precision-and-recall.