



GROUP ASSIGNMENT
TECHNOLOGY PARK MALAYSIA
CT118-3-3-ODL

Optimization and Deep Learning

HAND OUT DATE:

HAND IN DATE:

WEIGHTAGE: 70%

**“Prediction of Chicago Criminal Arrests By Using Optimisation and
Predictive Modelling Techniques”**

No.	Name	Group Component
		25%
	Basmah Zahid	25%
		25%
		25%

Table of Contents

Part 1: Group Component.....	3
1.1 Introduction – Dataset Selection	3
1.2 Model Selection.....	5
1.2.1 Random Forest	5
1.2.2 Logistic Regression	5
1.2.2.1 Logit function.....	5
1.2.3 Decision Trees (Basmah Zahid –	6
1.2.4 Support Vector Machine	7
2.1 Findings and Discussion	8
2.1.1 Data Pre-processing	8
2.1.2 Exploratory Data Analysis	16
2.1.3 Model Construction	26
Part 2: Individual Component – Model Building	31
2.2 Model Selection.....	31
2.2.1 Random Forest (.....	31
2.2.2 Logistic Regression	36
2.2.3 Decision Trees (Basmah Zahid –	49
2.2.4 Support Vector Machine	64
Part 2: Group Component.....	71
2.3 Critical Analysis of Models	71
2.4 Conclusions	73
2.5 References.....	74

Part 1: Group Component

1.1 Introduction – Dataset Selection

The dataset collected is gathered from Kaggle online open-source dataset collection. The dataset contained information regarding the crime incident that is being reported across the City of Chicago. The data is collected from the Chicago Police Department CLEAR (Citizen Law Enforcement Analysis and Reporting) system.

The accuracy of the data is valid based on the time of tracking and might be updated from time to time due to the current investigation and preliminary crime assessment being process every day. This project will use the data from the crime data in January 2020 consisting of 19,801 rows and 22 columns. The dataset columns are being described below:

Column	Description
Unique Key	Unique identification code to identify each record in the system
Case Number	Chicago Police Department Record Division (RD) number
Date	Date the incident occurred
Block	Block address where the incident occurred
IUCR	Illinois Uniform Crime Reporting code
Primary Type	The primary category of the IUCR code
Description	The sub-category of the IUCR code
Location Description	Description of the location where the incident occurred
Arrest	Indicate if the arrest is made
Domestic	Indicate if the case is a domestic violence case
Beat	Police geographic area to indicate where the incident occurred
District	Police district area where the incident occurred
Ward	City council district where the incident occurred
Community Area	Community area where the incident occurred
FBI (Federal Bureau of Investigation) Code	Indicated the crime classification based on FBI reporting system
X Coordinate	X coordinate where the incident occurred
Y Coordinate	Y coordinate where the incident occurred

Year	Year the incident occurred
Updated On	Last updated date of the case
Latitude	The latitude where the incident occurred
Longitude	Longitude where the incident occurred
Location	Map location where the incident occurred

Table 1- Data Dictionary

The dataset will be handled with Python programming language with its libraries. The dataset is being processed with Google Collab as the IDE and all of the visualisations are being made available in Python with libraries such as sklearn, panda, matplotlib, etc. The dataset and source code is attached for checking purposes.

As the dataset contain almost 20,000 rows and 22 columns of data, the dataset might be safe to use as there are a sufficient amount of data (more than 10,000). There are some obvious noise or data that need to be removed or formatted for the analysis purpose. Therefore, the dataset will go through some data cleaning and transformation processes before it can be used for constructing the model and do optimisation. The results of the data cleaning and model construction will be covered in the chapter below.

Later, the visualisation through Exploratory Data Analysis (EDA) will provide initial insights and information related to the dataset to better understand the data and discover some assumptions and hypotheses. The cross-validation to randomly select and split the data into train and test datasets with the ratio of 80:20 will be done during the data processing to make it easier to compare and test the results of the built model.

Therefore, with the pre-processing and selection of the dataset that would be handled in the chapter and subchapter below, the dataset is suitable to be used to build the model and the results will be discussed later. The ability of the dataset to produce a good model might also be discussed below to give information related to the suitability and evaluation results.

1.2 Model Selection

1.2.1 Random Forest

Random Forest is another supervised learning algorithm. Both regression and classification dictionaries can be used to optimize this model. Moreover, it is an algorithm that is easily adaptable. When a random forest algorithm is applied it forms a tree randomly. Many trees are generated through randomly chosen data samples and they make a random forest. Predictions are obtained from each tree; best solution is chosen. (Navlani, 2018) The Divide-and-Conquer methodology of decision trees produced on a randomly divided dataset is used by Random Forest. Decision tree classifiers form a group that are called a forest. The many decision trees generated get selected through a selection indicator such as Gini index, information gain or gain ratio. (Navlani, 2018)

1.2.2 Logistic Regression

Logistic Regression was chosen to analyse whether perpetrators were arrested for their crime or not using the dataset. Logistic regression comes under a classification algorithm and is suitable for predicting binary target (y) concerning or based on independent variables (X). (Thanda, 2020) Logistic regression is a suitable prediction model to be applied to the crime dataset selected as the target variable that has been chosen for modelling is binary. (Thanda, 2020) Binary variable refers to a variable that has two outcomes such as 0 and 1, yes and no and in this case, arrested or not arrested. (Thanda, 2020) Logistic regression is a machine learning algorithm that is easy to apply and evaluate based on several evaluation metrics such as confusion matrix, ROC (Receiver Operating Characteristic) curve, mean squared error, and root mean square error. This predictive model has been applied to several studies and can be used to detect the presence of cancer and disease along with other binary targets.

1.2.2.1 Logit function

A natural log of odds is known as a logit function. Here y is a binary variable with 2 categories 0 and 1. Logit functions work best with 0 and 1. The probability P in a logit function is the probability of y = 1. (Martin, 2015)

<p>• Logit function</p> $\text{Logit}(p_i) = \ln\left(\frac{p_i}{1-p_i}\right) = f(x)$ $p_i = \frac{e^{f(x)}}{1 + e^{f(x)}}$ <p>p_i = the probability of occurrence of i</p>
--

Figure 1 - Logit Function

Therefore, for the application of a successful logit function, X needs to be variables that may affect y. (Grace-Martin, 2015) For this assignment, for example, X can be equal to the district, crime area or street name along with other variables. Y can be the binary target variable Arrest.

1.2.3 Decision Trees (Basmah Zahid)

Decision Trees are a supervised and non-parametric learning algorithm that is used in both classification and regression. They can be referred to as a CART (Classification and Regression Tree) which stands for Classification and Regression Trees (CART) and they are utilized to carry out predictions based on the historical data provided to them. Classification trees are usually applied with binary targets such as a yes/no decision while regression trees are applied towards more continuous variables. (Quantstart, 2013)

The feature space is partitioned into several basic rectangular sections by axial parallel splits in DT/CART models. The average or mode of the train data' outputs, within the partition that the new finding corresponds to, is optimized to produce a forecast for a particular sample. Of course, as with other models, in decision trees, there is always a chance of overfitting the model which can be avoided if we prune the tree. This is modifying the tree splitting method so it can provide better results. Some of the criteria that can be used to split the tree include the Gini index and entropy. The formula for the algorithms is shown below. (Gupta, 2017)

$$Entropy = -p_1 * \log_2 p_1 - \dots - p_n * \log_2 p_n$$

Figure 2 – Entropy (GreatLearning, 2020)

The n here represents the number of classes and the lower the value, the better the prediction of the model.

$$Gini = 1 - \sum_i p_i^2$$

Figure 3 - Gini Index (GreatLearning, 2020)

Gini as compared to entropy carries out the calculations faster although being similar. Here i represents the number of the classes.

1.2.4 Support Vector Machine

Support Vector Machine (SVM) is one of the machine learning classifiers and nonlinear regressions that is commonly used with its simple algorithm usually producing great accuracy with less computational introduced back in 1963. The purpose of the Support Vector Machine is to find hyperplanes into specific numbers of dimensional space to classify the data points distinctively. SVM is widely used in problems such as text categorization, image classification, character recognition, and many more (Jain, 2017).

SVM used a kernel-based technique to transform the data to find hyperplane boundaries to classify and separate the output. SVM works by identifying the optimal parameters to separate and maximize the separation of training data between the data points. As SVM can work in multi-dimensional numbers of the dataset, the point or line is built depending on that number of dimensions. So, SVM is used to find optimal hyperplane for the model prediction (Jain, 2017).

$$w^T x = 0$$

Figure 4 - Formula

The common equation of SVM hyperplane according to the above figure. The w is usually called a weight vector and x also represent the vector. So, the formula represented the dot product of two vectors (Jain, 2017). In this project, SVM will be implemented to see if the model is suitable to be used for the dataset. The SVM will also be tuned by using hyperparameter tuning to see the improvement of the accuracy score that is the evaluation matrix to see the suitability of the model.

Commonly, SVM provides among the highest accuracy score compared to other models, in this project we are going to compared it with other models, the discussion and evaluation matrix of the models will be discussed. The suitability of the SVM model to be used for the dataset can be checked based on the evaluation later.

2.1 Findings and Discussion

2.1.1 Data Pre-processing

```

1 #Importing libraries for EDA and preprocessing
2 import sklearn
3 import pandas as pd
4 import numpy as np
5 import seaborn as sns
6 import matplotlib.pyplot as plt
7 # Reading Crime Data File of the crimes of January 2021
8 df = pd.read_csv ('/content/drive/MyDrive/Colab Notebooks/ODL/Assignment_ODL/Crimes-Jan-2021-data.csv')

```

1 Review first 5 records
2 df.head(5)

Unnamed: 0	ID	Case Number	Date	Block	DIR	Primary Type	Description	Location Description	Arrest	Domestic	Beat	District	Ward	Community Area	FBI Code	X Coordinate	Y Coordinate	Year	Updated On	Latitude	Longitude	Location year month day time
0	0	12016034	1/1/2020 0:00	0180X N WINNEBAGO AVE	1153	DECEPTIVE PRACTICE	FINANCIAL IDENTITY THEFT OVER \$ 300	APARTMENT	False	False	1434	14	32.0	22.0	11	1160263.0	1912391.0	2020	03/29/2020 03:45:12 PM	41.915306	-87.686639	(41.915306, -87.686639) 2020 1 1 00:00:00
1	1	12220321	1/1/2020 0:00	0910X S DREXEL AVE	1752	OFFENSE INVOLVING CHILDREN	AGGRAVATED CRIMINAL SEXUAL ABUSE BY FAMILY MEMBER	RESIDENCE	False	True	413	4	8.0	47.0	17	1184157.0	1844395.0	2020	12/19/2020 03:45:59 PM	41.728182	-87.600985	(41.728182, -87.600985) 2020 1 1 00:00:00
2	2	12013828	1/1/2020 0:00	0440X S LAVERGNE AVE	281	CRIMINAL SEXUAL ASSAULT	NON-AGGRAVATED	APARTMENT	False	False	614	8	22.0	56.0	2	1143770.0	1874726.0	2020	03/29/2020 03:47:02 PM	41.812274	-87.748177	(41.812274, -87.748177) 2020 1 1 00:00:00
3	3	12019692	1/1/2020 0:00	0320X N LINCOLN AVE	1153	DECEPTIVE PRACTICE	FINANCIAL IDENTITY THEFT OVER \$ 300	APARTMENT	False	False	1922	19	47.0	6.0	11	1164963.0	1921507.0	2020	4/1/2020 15:50	41.940222	-87.669039	(41.940222, -87.669039) 2020 1 1 00:00:00
4	4	12036792	1/1/2020 0:00	0720X S WHIPPLE ST	1154	DECEPTIVE PRACTICE	FINANCIAL IDENTITY THEFT \$300 AND UNDER	RESIDENCE	False	False	631	8	18.0	66.0	11	1157290.0	1856526.0	2020	04/29/2020 03:53:17 PM	41.762067	-87.699077	(41.762067, -87.699077) 2020 1 1 00:00:00

Figure 5 - Importing Libraries and viewing dataset

1 Review last 5 records
2 df.tail(5)

Unnamed: 0	ID	Case Number	Date	Block	DIR	Primary Type	Description	Location Description	Arrest	Domestic	Beat	District	Ward	Community Area	FBI Code	X Coordinate	Y Coordinate	Year	Updated On	Latitude	Longitude	Location year month day time
19796	19796	11968231	01/31/2020 11:48:00 PM	0400X S SACRAMENTO AVE	1811	NARCOTICS	POSS. CANNABIS 300MS OR LESS	STREET	True	False	921	9	12.0	58.0	18	1157014.0	1877747.0	2020	2/7/2020 15:52	41.820306	-87.699516	(41.820306, -87.699516) 2020 1 31 23:48:00
19797	19797	11968255	01/31/2020 11:49:00 PM	0120X E 87TH ST	3730	INTERFERENCE WITH PUBLIC OFFICER	OBSTRUCTING JUSTICE	STREET	True	False	412	4	8.0	48.0	24	1186221.0	1847553.0	2020	2/7/2020 15:52	41.736810	-87.593325	(41.736810, -87.593325) 2020 1 31 23:49:00
19798	19798	11968303	01/31/2020 11:50:00 PM	0480X W WELLINGTON AVE	496	BATTERY	AGGRAVATED DOMESTIC BATTERY KNIFE/CUTTING INST	VEHICLE NON-COMMERCIAL	False	True	2521	25	31.0	19.0	04B	1143021.0	1919496.0	2020	2/7/2020 15:52	41.935142	-87.749808	(41.935142, -87.749808) 2020 1 31 23:50:00
19799	19799	11968986	01/31/2020 11:50:00 PM	0330X N Halsted St	890	THEFT	FROM BUILDING	BASE OR TAVERN	False	False	1925	19	44.0	6.0	6	1170328.0	1922582.0	2020	2/7/2020 15:52	41.943056	-87.649363	(41.943056, -87.649363) 2020 1 31 23:50:00
19800	19800	11968238	01/31/2020 11:55:00 PM	0680X S AVENUE H	405	BATTERY	DOMESTIC BATTERY SIMPLE	APARTMENT	True	True	432	4	19.0	52.0	08B	1202788.0	1840552.0	2020	2/7/2020 15:52	41.717192	-87.532899	(41.717192, -87.532899) 2020 1 31 23:55:00

Figure 6 - viewing last 5 records


```

1 #datatypes to see types of data
2 df.dtypes

Unnamed: 0          int64
ID                  int64
Case Number         object
Date               object
Block              object
IUCR               object
Primary Type        object
Description         object
Location Description object
Arrest             bool
Domestic           bool
Beat              int64
District           int64
Ward              float64
Community Area     float64
FBI Code           object
X Coordinate       float64
Y Coordinate       float64
Year              int64
Updated On         object
Latitude          float64
Longitude         float64
Location           object
year              int64
month             int64
day              int64
time              object
dtype: object

```

Figure 7 - Exploring Datatypes

```

#to view columns and rows
df.shape

(19801, 27)

```

Figure 8 - Dataset Rows and Column

#dataset summary - can be used to replace missing values and clean data
df.describe()

	Unnamed: 0	ID	Beat	District	Ward	Community Area	X Coordinate	Y Coordinate	Year	Latitude	Longitude	year	month	day
count	19801.000000	1.980100e+04	19801.000000	19801.000000	19801.000000	19801.000000	1.965200e+04	1.965200e+04	19801.0	19652.000000	19652.000000	19801.0	19801.0	19801.000000
mean	9900.000000	1.193715e+07	1140.326448	11.173729	23.341346	36.905055	1.104962e+06	1.886102e+06	2020.0	41.843053	-87.670165	2020.0	1.0	15.799758
std	5716.200676	5.093355e+05	694.238255	6.935818	13.892011	21.563978	1.616986e+04	3.136727e+04	0.0	0.086260	0.058867	0.0	0.0	9.173730
min	0.000000	2.488900e+04	111.000000	1.000000	1.000000	1.000000	1.092047e+06	1.814512e+06	2020.0	41.645796	-87.934567	2020.0	1.0	1.000000
25%	4950.000000	1.104704e+07	611.000000	6.000000	10.000000	23.000000	1.153286e+06	1.858727e+06	2020.0	41.767748	-87.712522	2020.0	1.0	8.000000
50%	9900.000000	1.195508e+07	1023.000000	10.000000	24.000000	32.000000	1.166775e+06	1.892894e+06	2020.0	41.861834	-87.663594	2020.0	1.0	16.000000
75%	14850.000000	1.196305e+07	1713.000000	17.000000	34.000000	54.000000	1.176447e+06	1.908580e+06	2020.0	41.904659	-87.627845	2020.0	1.0	24.000000
max	19800.000000	1.237339e+07	2535.000000	31.000000	50.000000	77.000000	1.204801e+06	1.951493e+06	2020.0	42.022548	-87.525663	2020.0	1.0	31.000000

Figure 9 - Describing Dataset

```

1 df['Arrest'].describe()
2 #describe target records

count      19801
unique         2
top         False
freq       15291
Name: Arrest, dtype: object

```

Figure 10 - Describing Arrest before label encoding

```

1 #changing datatype for clearer EDA
2 df["Arrest"]=df["Arrest"].astype('category')
3 df["Domestic"]=df["Domestic"].astype('category')
4 df["Primary Type"]=df["Primary Type"].astype('category')
5 #updated data types
6 df.dtypes

Unnamed: 0      int64
ID              int64
Case Number     object
Date            object
Block           object
IUCR            object
Primary Type    category
Description     object
Location Description  object
Arrest          category
Domestic        category
Beat            int64
District        int64
Ward            float64
Community Area  float64
FBI Code        object
X Coordinate    float64
Y Coordinate    float64
Year            int64
Updated On      object
Latitude        float64
Longitude       float64
Location        object
year            int64
month           int64
day             int64
time            object
dtype: object

```

Figure 11 - Changing Data types of Arrest, Domestic and Primary Type

Data types for Arrest, Domestic and Primary type were changed to the category for clearer EDA (Exploratory Data Analysis) ahead.

Splitting, Recoding and Adding Columns

```
1 #splitting block into block number, direction and street name for exploration
2 #NEWS - North, East, West and South
3 bs = pd.DataFrame(df.Block.str.split(' ',2).tolist(),columns = ['Block_Num','NEWS','Street_Name'])
4 bs.head(10)
```

	Block_Num	NEWS	Street_Name
0	018XX	N	WINNEBAGO AVE
1	091XX	S	DREXEL AVE
2	044XX	S	LAVERGNE AVE
3	032XX	N	LINCOLN AVE
4	072XX	S	WHIPPLE ST
5	072XX	S	UNIVERSITY AVE
6	032XX	N	SHEFFIELD AVE
7	022XX	N	LONG AVE
8	022XX	W	111TH ST
9	020XX	N	LOCKWOOD AVE

Figure 12 - splitting

Splitting of variable “Block” was done for a clearer representation of street names when conducting EDA.

```
#joining split columns from block and original file using concatenate
#renaming to crimes
df = pd.concat([df, bs], axis=1)
df.head(5)
```

ID	Location Description	Arrest	Domestic	Beat	District	Ward	Community Area	FBI Area Code	X Coordinate	Y Coordinate	Year	Updated On	Latitude	Longitude	Location	year	month	day	time	Block_Num	NEWS	Street_Name
NL FY JR 00	APARTMENT	False	False	1434	14	32.0	22.0	11	1160263.0	1912391.0	2020	03/26/2020 03:45:12 PM	41.915306	-87.686639	(41.915306069, -87.686639247)	2020	1	1	00:00:00	018XX	N	WINNEBAGO AVE
ID NL FY LY JR	RESIDENCE	False	True	413	4	8.0	47.0	17	1184157.0	1844395.0	2020	12/19/2020 03:45:59 PM	41.728192	-87.600985	(41.728192429, -87.600985433)	2020	1	1	00:00:00	091XX	S	DREXEL AVE
N- ID	APARTMENT	False	False	814	8	22.0	56.0	2	1143770.0	1874726.0	2020	03/28/2020 03:47:02 PM	41.812274	-87.748177	(41.81227369, -87.748176594)	2020	1	1	00:00:00	044XX	S	LAVERGNE AVE
FY JR 00	APARTMENT	False	False	1922	19	47.0	6.0	11	1164983.0	1921507.0	2020	4/1/2020 15:50	41.940222	-87.669039	(41.940221932, -87.669039008)	2020	1	1	00:00:00	032XX	N	LINCOLN AVE
NL FY DO JR	RESIDENCE	False	False	831	8	18.0	66.0	11	1157290.0	1856526.0	2020	04/29/2020 03:53:17 PM	41.762067	-87.699077	(41.762066981, -87.699077348)	2020	1	1	00:00:00	072XX	S	WHIPPLE ST

Figure 13 - Adding split cells to the original dataset

```
1 # Coding Arrest, Domestic and Primary Type to new columns containing recoded values for modeling purposes.
2 df["Arrest_New"] = df["Arrest"].cat.codes
3 df["Domestic_New"] = df["Domestic"].cat.codes
4 df["Crime_Type"] = df["Primary Type"].cat.codes
5 df.head()
```

Figure 14 - recoding

Block_Num	NEWS	Street_Name	Arrest_New	Domestic_New	Crime_Type
018XX	N	WINNEBAGO AVE	0	0	9
001XX	S	DREXEL AVE	0	1	19
044XX	S	LAVERGNE AVE	0	0	7
032XX	N	LINCOLN AVE	0	0	9
072XX	S	WHIPPLE ST	0	0	9

Figure 15 - Recoding Arrest, Domestic and Primary type

Arrest, Domestic and Primary types were recorded and recoded values were put in new columns for modelling purposes.

Identifying for Missing Values

```

1 #check for missing values in each column
2 df.isnull().sum()

```

Unnamed: 0	0
ID	0
Case Number	0
Date	0
Block	0
IUCR	0
Primary Type	0
Description	0
Location Description	117
Arrest	0
Domestic	0
Beat	0
District	0
Ward	0
Community Area	0
FBI Code	0
X Coordinate	149
Y Coordinate	149
Year	0
Updated On	0
Latitude	149
Longitude	149
Location	149
year	0
month	0
day	0
time	0
Block_Num	0
NEWS	0
Street_Name	0
Arrest_New	0
Domestic_New	0
Crime_Type	0
dtype: int64	

Figure 16 - Identifying Missing Values

Variables: Location Description has 117 values missing whereas X Coordinate, Y Coordinate, Latitude, Longitude and Location all have 149 values missing.

```
#drop rows with missing values
df_drop = df.dropna()
```

Figure 17 - Rows with missing values dropped

```
1 #drop original columns of split/ recoded data to avoid redundancy.
2 df = df_drop.drop(['Unnamed: 0', 'Block_Num', 'NEWS', 'month', 'Year', 'year', 'Case Number', 'Date'], axis=1)
```

Figure 18 - drop original columns that have been recoded or replaced

```
1 #showing missing value after dropping rows
2 df.isnull().sum()

ID                0
Block             0
IUCR             0
Primary Type     0
Description       0
Location Description 0
Arrest           0
Domestic         0
Beat            0
District         0
Ward            0
Community Area   0
FBI Code        0
X Coordinate     0
Y Coordinate     0
Updated On      0
Latitude        0
Longitude       0
Location        0
day            0
time           0
Street_Name     0
Arrest_New     0
Domestic_New    0
Crime_Type     0
dtype: int64
```

Figure 19 - after removal of missing values

```

def category(crime_type):
    if 'AIRPORT' in crime_type: #FINDING IN OLD COLUMN
        return 'AIRPORT' #NEW NAME IN COLUMN
    elif 'AIRCRAFT' in crime_type:
        return 'AIRPORT'
    elif 'COIN OPERATED MACHINE' in crime_type:
        return 'ATM'
    elif 'ATM' in crime_type:
        return 'ATM'
    elif 'AUTO' in crime_type:
        return 'AUTO'
    elif 'BARBER' in crime_type:
        return 'BARBER SHOP'
    elif 'CHA' in crime_type:
        return 'CHA'
    elif 'CHURCH' in crime_type:
        return 'PLACE OF WORSHIP'
    elif 'COLLEGE' in crime_type:
        return 'EDUCATIONAL INSTITUTION'
    elif 'DAY CARE CENTRE' in crime_type:
        return 'EDUCATIONAL INSTITUTION'
    elif 'LIBRARY' in crime_type:
        return 'EDUCATIONAL INSTITUTION'
    elif 'COMMERCIAL/BUSINESS OFFICE' in crime_type:
        return 'BUSINESS OFFICE'
    elif 'CTA' in crime_type:
        return 'CTA'
    elif 'FACTORY /MANUFACTURING BUILDING' in crime_type:
        return 'FACTORY'
    elif 'GOVERNMENT BUILDING / PROPERTY' in crime_type:
        return 'PROPERTY'
    elif 'GROCERY FOOD STORE' in crime_type:
        return 'STORE'
    elif 'DRUG STORE' in crime_type:
        return 'STORE'
    elif 'DEPARTMENT STORE' in crime_type:
        return 'STORE'
    elif 'LIQUOR STORE' in crime_type:
        return 'STORE'
    elif 'TAVERN' in crime_type:
        return 'STORE'
    elif 'BAR OR TAVERN' in crime_type:
        return 'STORE'
    elif 'PAWN SHOP' in crime_type:
        return 'STORE'
    elif 'RETAIL STORE' in crime_type:
        return 'STORE'
    elif 'CONVENIENCE STORE' in crime_type:
        return 'STORE'
    elif 'SMALL RETAIL STORE' in crime_type:
        return 'STORE'
    elif 'CLEANING STORE' in crime_type:
        return 'STORE'
    elif 'HIGHWAY / EXPRESSWAY' in crime_type:
        return 'HIGHWAY'
    elif 'HOSPITAL' in crime_type:
        return 'MEDICAL'
    elif 'MEDICAL' in crime_type:
        return 'MEDICAL'
    elif 'HOTEL' in crime_type:
        return 'HOTEL'
    elif 'LAKEFRONT' in crime_type:
        return 'LAKEFRONT'
    elif 'JAIL / LOCK-UP FACILITY' in crime_type:
        return 'JAIL'
    elif 'MOVIE HOUSE' in crime_type:
        return 'ENTERTAINMENT CENTRE'
    elif 'NEWSSTAND' in crime_type:
        return 'ENTERTAINMENT CENTRE'
    elif 'NURSING' in crime_type:
        return 'NURSING HOME'
    elif 'OTHER' in crime_type:
        return 'TRANSPORTATION'
    elif 'PARKING LOT' in crime_type:
        return 'PARKING LOT'

```

Figure 20 - categorizing primary type by crime type

Above step for categorizing primary type was done for clearer EDA. Only a portion of the code is shown here, the remaining is displayed in the google colab workspace.

```

1 df['Crime_Area'] = df['Location Description'].apply(category)
2 df.dtypes

```

ID	int64
Block	object
IUCR	object
Primary Type	category
Description	object
Location Description	object
Arrest	category
Domestic	category
Beat	int64
District	int64
Ward	float64
Community Area	float64
FBI Code	object
X Coordinate	float64
Y Coordinate	float64
Updated On	object
Latitude	float64
Longitude	float64
Location	object
day	int64
time	object
Street_Name	object
Arrest_New	int8
Domestic_New	int8
Crime_Type	int8
Crime_Area	object
dtype:	object

Figure 21 - renaming location description as crime area and giving it a category data type

Category data type was given so the variables can be analysed as categories during EDA.

2.1.2 Exploratory Data Analysis

The initial data investigations process to perform exploratory data analysis is to find the initial information, insights, and patterns to spot and discover if there are any anomalies, early hypotheses, and assumptions by using the help of graphical visualisation and statistics (Patil, 2018). The EDA process in this project is being done with Python programming language with Matplotlib and Seaborn library.

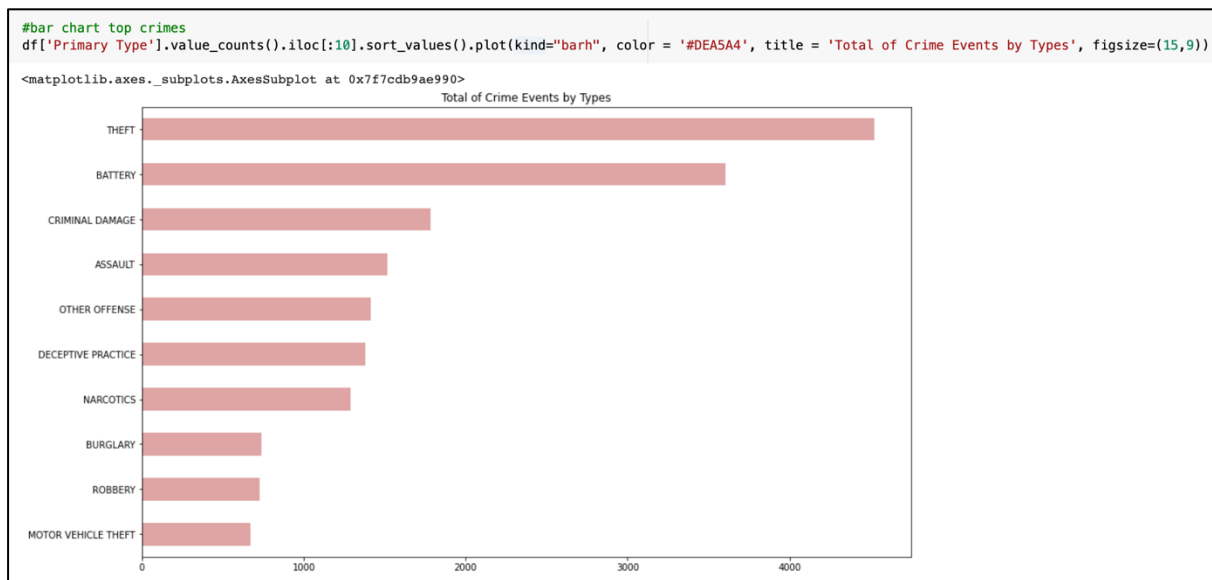


Figure 22 – bar chart for top crimes

The bar chart from the figure above shown the primary type of crimes events in ascending order. As can be seen based on the above plot, most of the crimes are theft accounted for more than 4,000 total cases in January 2020. The second position went to battery and criminal damage.


```
df['Crime_Area'].value_counts().nlargest(10).plot(kind='barh', figsize=(15,9), color = '#FBB62')
plt.title("Top 10 Areas Most Prone To Crime")
plt.ylabel("Area")
plt.xlabel("Number of crimes");
```

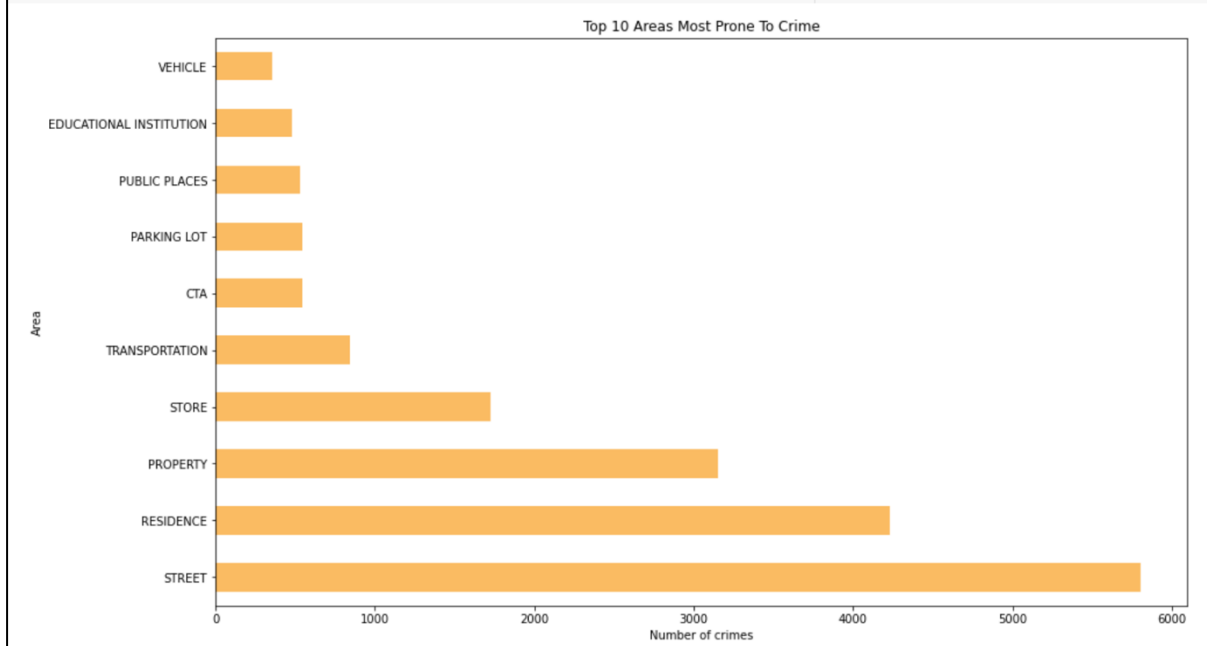


Figure 23 - bar chart for prone areas to crime

The figure from the bar chart above shown the crime area which is highly prone to crime in descending order from the least to most. The street area is the most area with criminal activity with almost 6,000 cases in January 2020 related to crimes.

```
df["Street_Name"].value_counts().iloc[100:150]
```

FRANCISCO AVE	50
MAPLEWOOD AVE	49
JUSTINE ST	49
ADDISON ST	49
ARMITAGE AVE	49
JEFFERY BLVD	49
SACRAMENTO AVE	49
KEELER AVE	49
LARAMIE AVE	49
DIVERSEY AVE	48
HERMITAGE AVE	48
VINCENNES AVE	48
FAIRFIELD AVE	48
OGDEN AVE	47
KILDARE AVE	47
CLAREMONT AVE	47
ELSTON AVE	47
WILSON AVE	46
LAVERGNE AVE	46
CARPENTER ST	46
THROOP ST	46
LOWE AVE	46
ELLIS AVE	46
BISHOP ST	45
AVERS AVE	45
INGLESIDE AVE	45
OAKLEY AVE	44
ARTESIAN AVE	44
DREXEL AVE	44
KILBOURN AVE	43
LOCKWOOD AVE	43
CHRISTIANA AVE	43
MARQUETTE RD	42
EXCHANGE AVE	42
HONORE ST	42
KIMBALL AVE	42
ADA ST	42

Figure 24 - streets where crimes occur

As street related to most of the criminal cases, the figure above shown the specific street name with the number of total crimes committed in the relevant street. As we can see, Fransisco Avenue has the highest number of crimes while as we can see, the other street also has a high number of crimes committed.

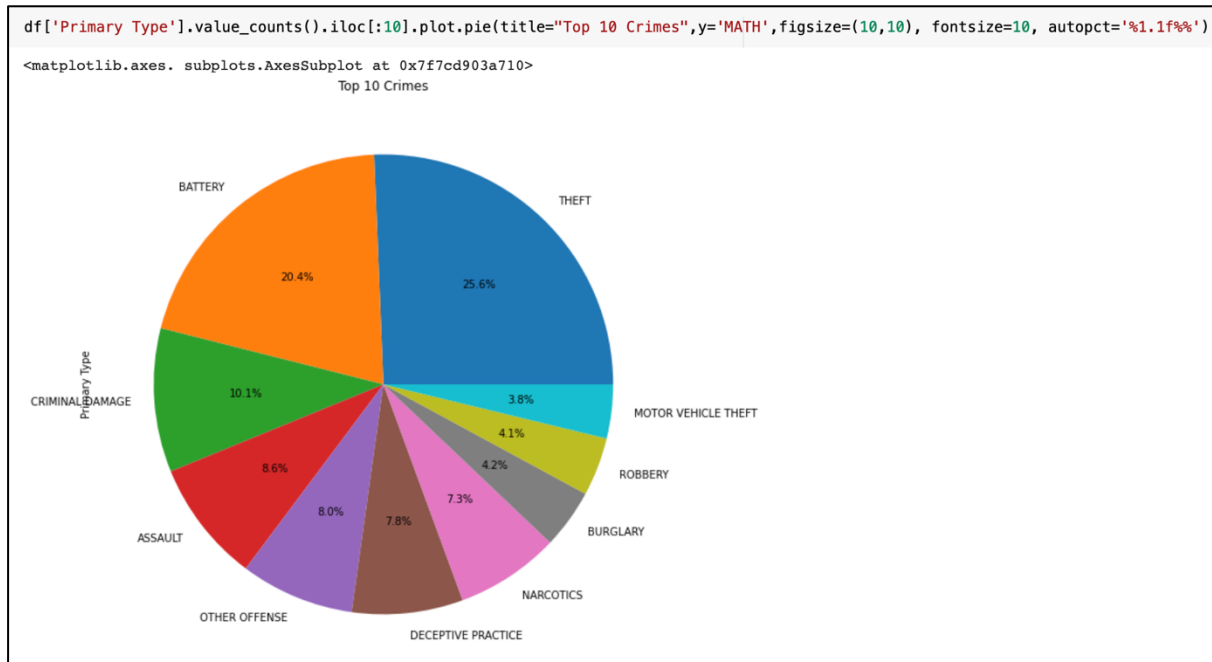


Figure 25 - piechart to show top 10 types of crimes

The pie chart from the figure above shown the top 10 primary types of crime based on the number percentage of the type of crime. As can be seen, theft accumulated of more than a quarter of the committed crime type.

```
df.Street_Name.value_counts().nlargest(40).plot(kind='bar', figsize=(15,9))
plt.title("Top 40 Crimes / Street Name")
plt.ylabel("Total Crimes")
plt.xlabel("Street");
```

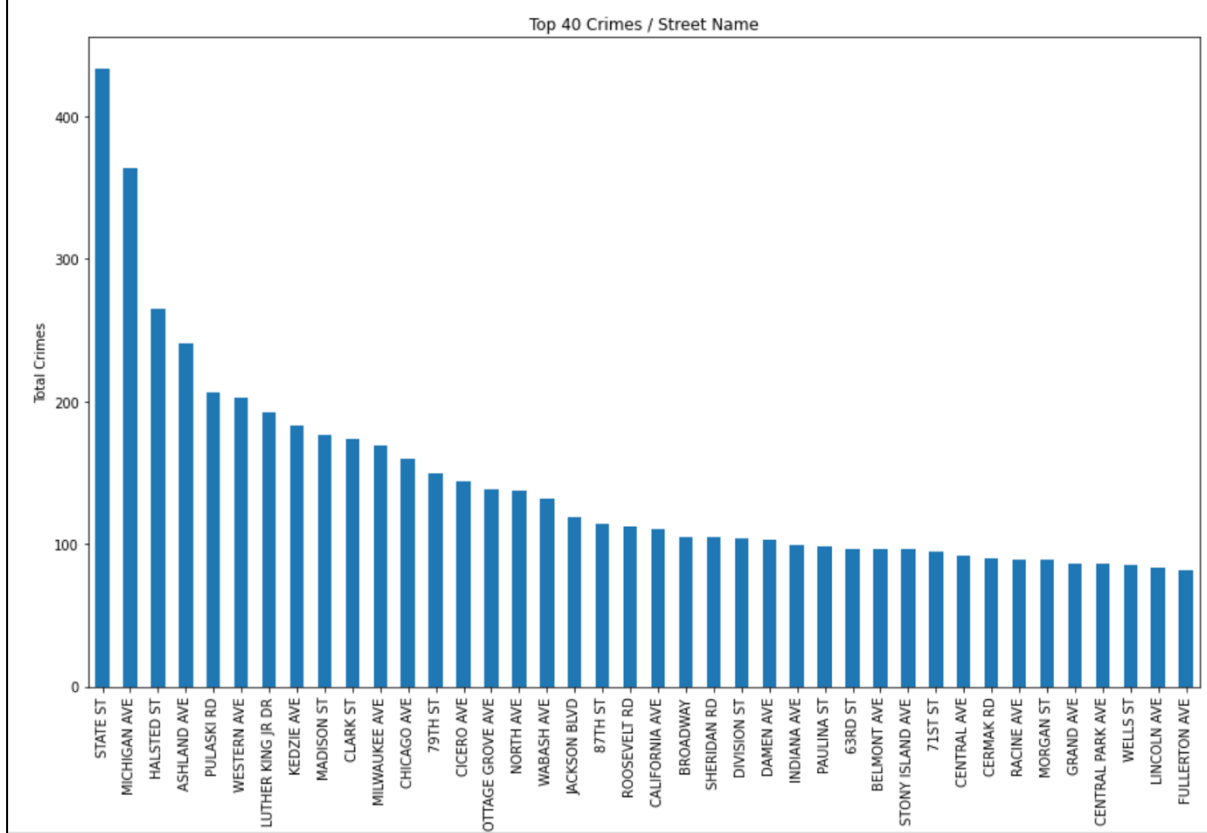


Figure 26 - top 40 crimes per street crime

The bar chart from the figure above shown the highest to lowest number of crimes by the street name. The top 40 street is being sorted and we can see that state street have the highest number of crimes with more than 400 total crimes.

```
#check Domestic Violence Cases
```

```
fig = plt.figure(figsize = (10,8))  
df['Domestic'].value_counts(normalize = True).plot(kind='bar', color= ['blue','red'], alpha = 0.9, rot=0)  
plt.title('Domestic Violence')  
plt.show()
```

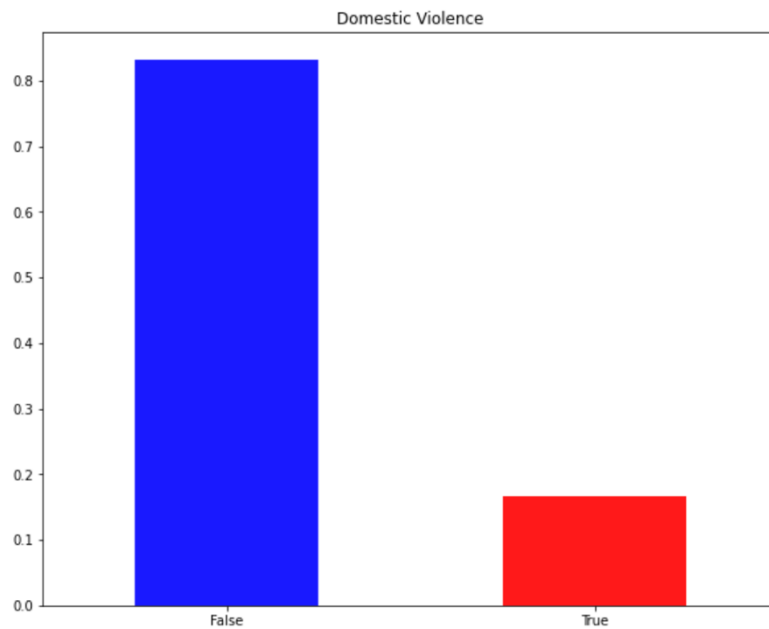


Figure 27 - distribution of domestic violence cases

The bar chart from the above figure represented the domestic violence cases in the month. True value means the cases reported are regarding domestic violence cases accumulated less than 20% of total cases while the blue false represented nondomestic violence case of more than 80% of the total cases.

```
#check Arrested Cases
```

```
fig = plt.figure(figsize = (10,8))  
df['Arrest'].value_counts(normalize = True).plot(kind='bar', color= ['red','green'], alpha = 0.9, rot=0)  
plt.title('Arrested')  
plt.show()
```

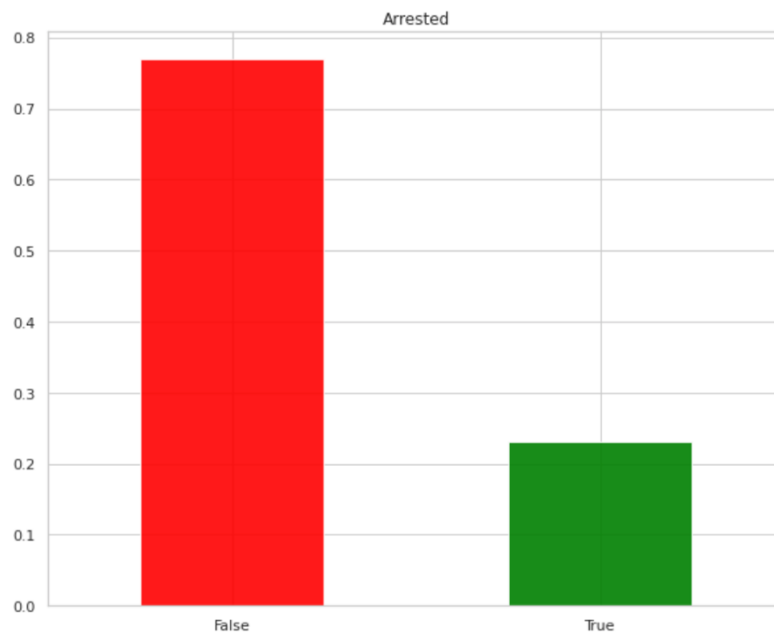


Figure 28 - distribution of arrested cases

The bar chart from the figure above shown the cases with suspected individuals being arrested or not. This column also will be used as the target variable for prediction model construction later. The higher red bar represent the cases that are not arrested which is more than 70% of the total cases. while the green bar represented that the suspect has been arrested and being investigated or charged for the cases, there are more than 20% of the cases has the suspected individuals being arrested.

```
df.Crime_Area.value_counts().nlargest(40).plot(kind='bar', figsize=(15,9))
plt.title("Top Crime Area")
plt.ylabel("Number of Crimes");
plt.xlabel("Area");
```

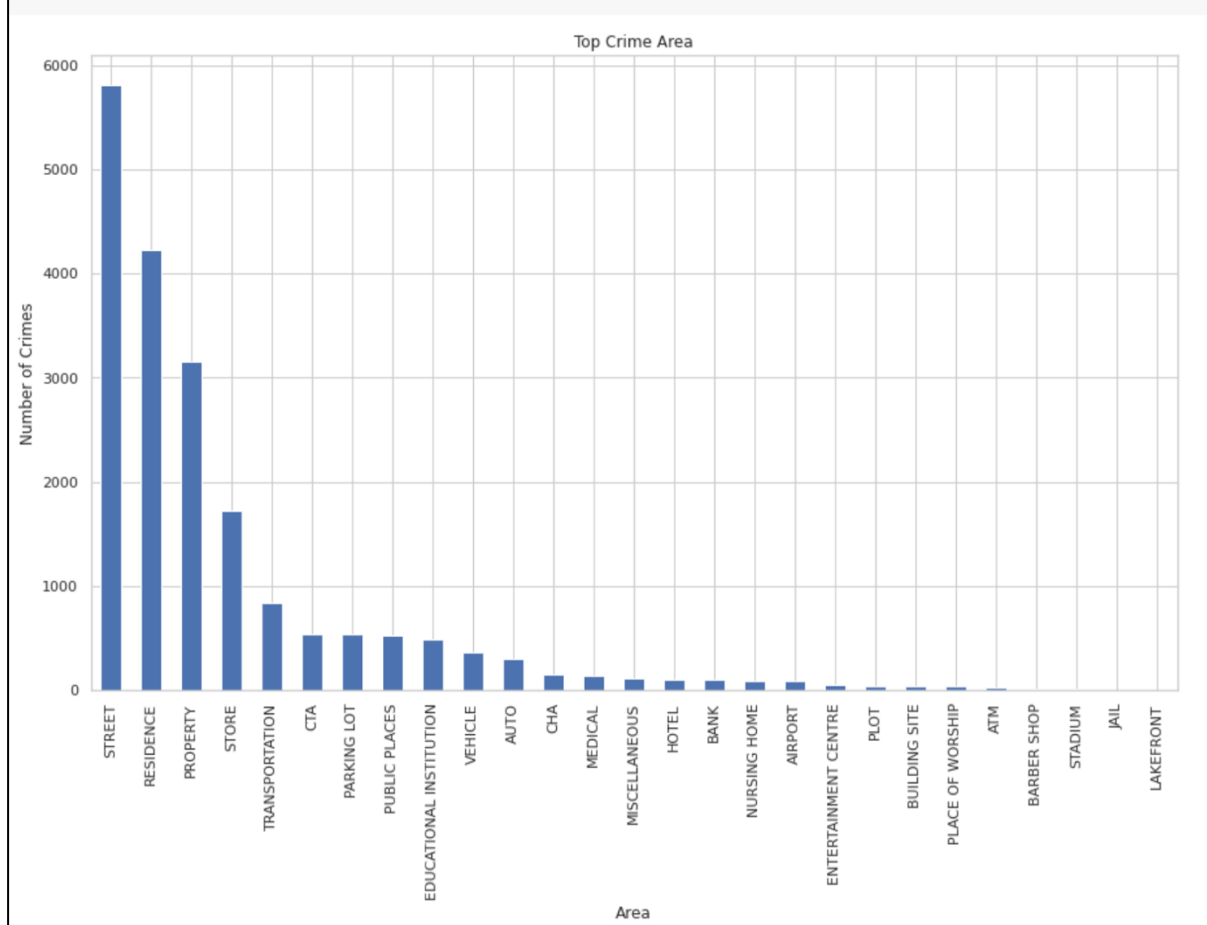


Figure 29 - top crimes areas

The bar chart from the figure above shown the top crime area that crime has happened in the past month of January 2020. Street is the area with the highest crime cases as it is an open public space, while residence and property stands as the second and third highest crime area respectively.

```
#District Numbers
```

```
df['District'].hist(bins=50, figsize=(12,10))
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f7cd8adc6d0>
```

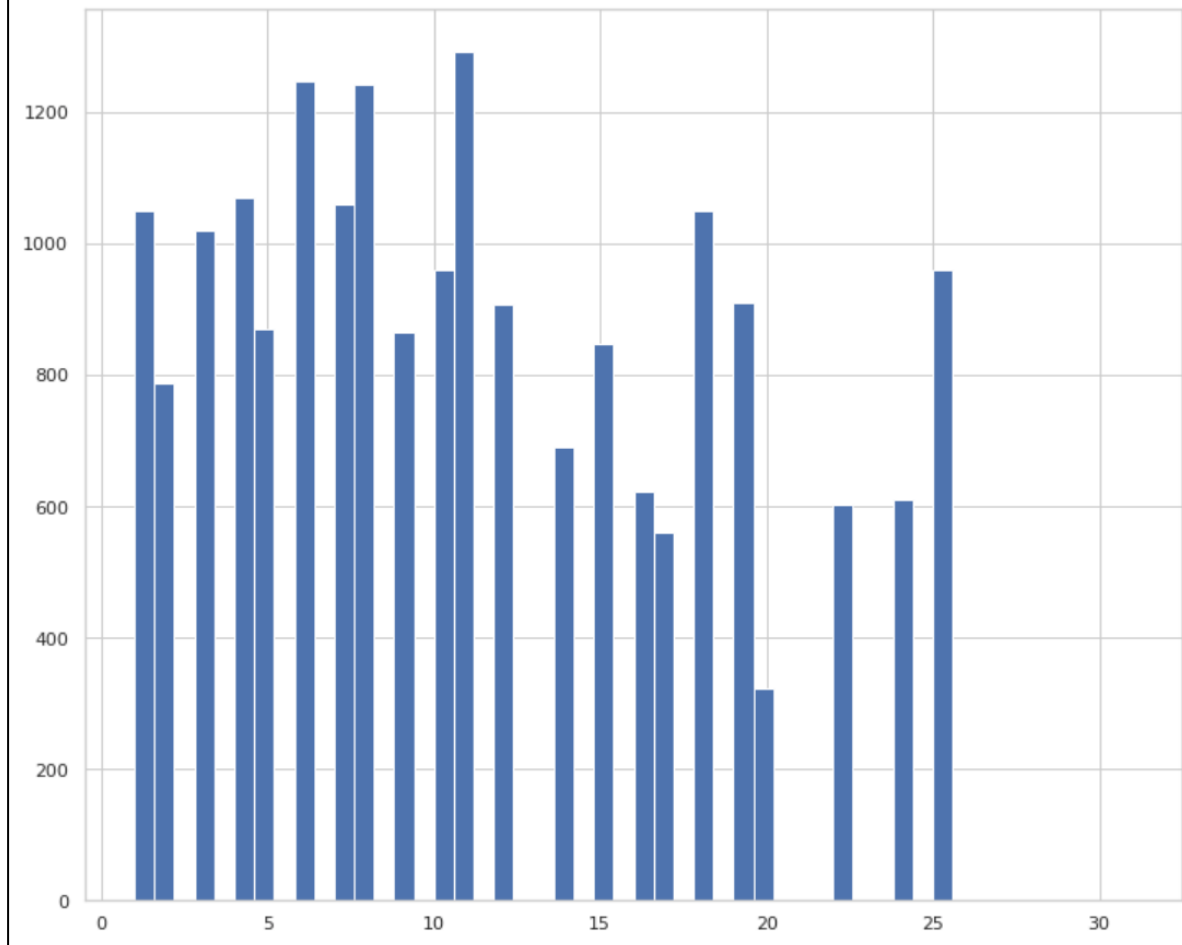


Figure 30 – histogram for district number

The bar chart above shown the number of cases based on the district numbering, we can see from the chart that district with the highest number of cases are district 11, the other district number 6 and 8 also have some of the highest number of cases with all having more than 1200 total cases each.

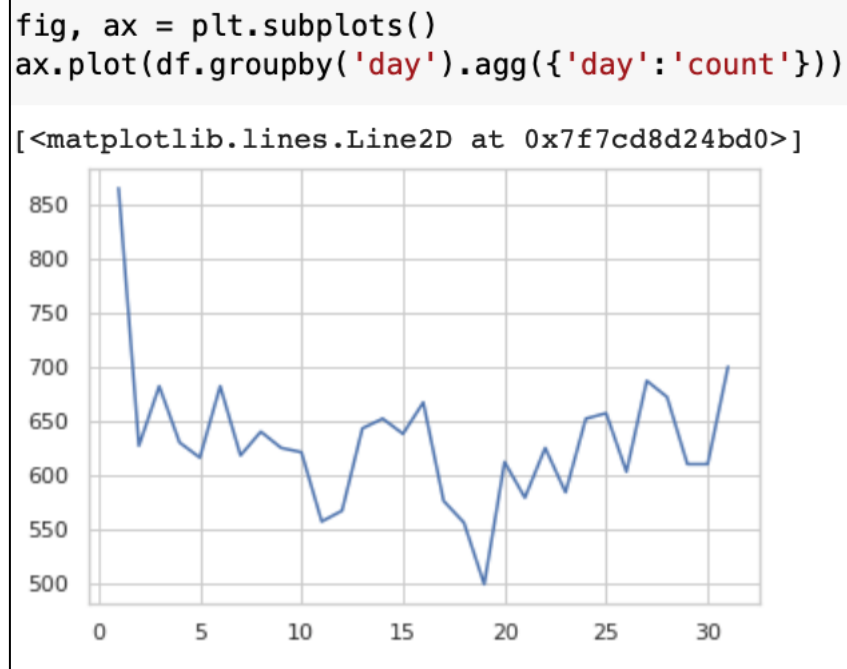


Figure 31

The line plot from the figure above shown the number of cases based on the date. As can be seen form the figure, the number of cases arises during the beginning of the month and gradually decreases. The lowest cases can be seen during the period between mid of the month to end of the month, date 15 to 20 January 2020. As the cases started to rise back up to the end of the month.

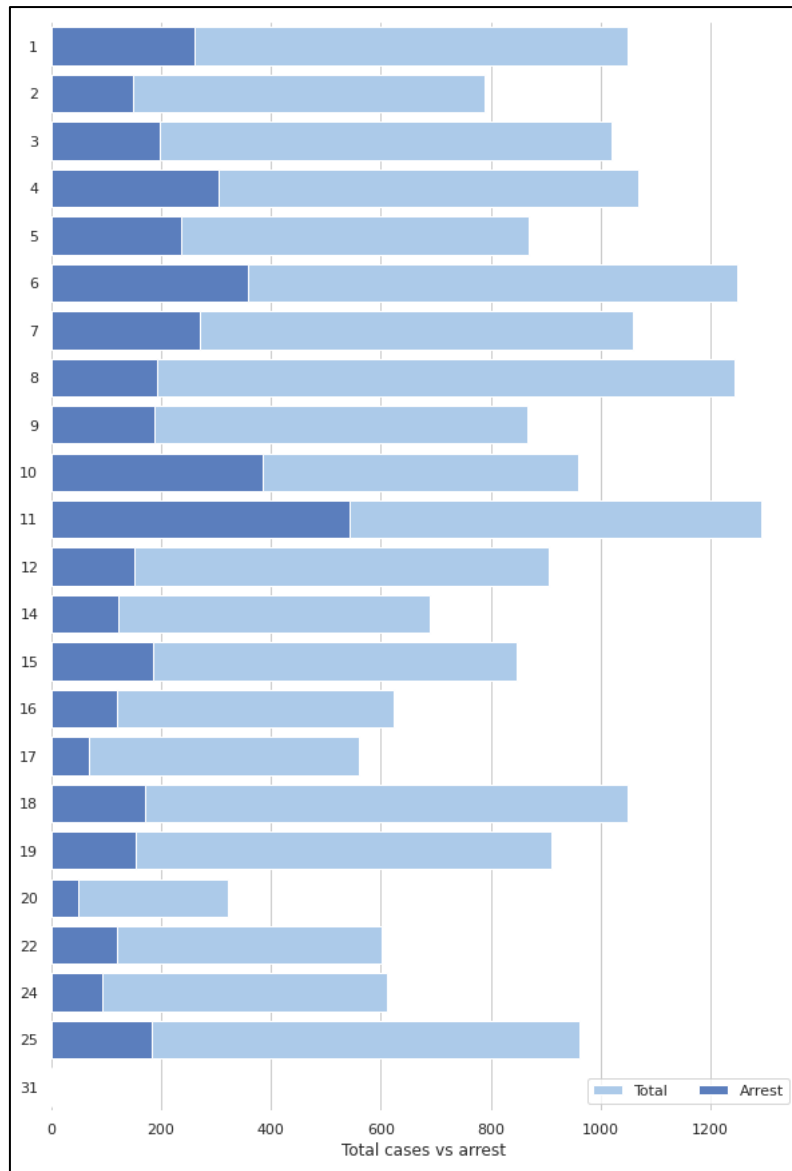


Figure 32 - total number of cases vs arrested stacked bar chart

The stacked bar chart from the figure above shown the total cases by district number, comparing the cases by the one being arrested out of the cases in total. We can see a significant number of arrests are being made in district 11 where the total cases are also high compared to any other district.

2.1.3 Model Construction

```
[ ] 1 #categorical data to numeric
    2 df['Street_Name'] = df['Street_Name'].astype('category')
    3 df['Crime_Area'] = df['Crime_Area'].astype('category')
    4 df["CrimeArea_New"] = df["Crime_Area"].cat.codes
    5 df["StreetName_New"] = df["Street_Name"].cat.codes
```

As the machine cannot process categorical data, some columns such as Street Name and Crime Area has been changed into numerical data for an easier process.

```
1 #unnecesary columns have been dropped
2 df = df.drop(['ID', 'IUCR', 'Arrest', 'Domestic', 'Block', 'Description', 'FBI Code', 'X Coordinate', 'Y Coordinate', 'Updated On', 'Location', 'time', 'Street_Name', 'Crime_Area'], axis=1)
```

Figure 33 - dropping redundant/unused columns

The model construction and processing only required some important columns. Therefore, the rest of the columns will be dropped for easier and faster processing.

```
[ ] 1 # encode Primary Type column to be numerical
    2
    3 from sklearn.preprocessing import LabelEncoder
    4 le = LabelEncoder()
    5 df['Primary Type'] = le.fit_transform(df['Primary Type'])

[ ] 1 # encode Location Description column to be numerical
    2 df['Location Description'] = le.fit_transform(df['Location Description'])
```

Figure 34 - encoding of categorical variables

By using label encoding, the primary type and location description column has been encoded to change its category into the numerical variable.

```
1 # partitioning independent and dependent data
2
3 X=df.drop('Arrest_New',axis=1)
4 y=df['Arrest_New']

1 # check if any column are having constant variance
2
3 from sklearn.feature_selection import VarianceThreshold
4 vt=VarianceThreshold(threshold=0.0)
5 vt.fit_transform(X)
6 vt.get_support()

array([ True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True])
```

Figure 35 - partitioning data set and checking any variance

During the feature processing, the target variable that is chosen is Arrest_new to help determine the possibility of being arrested based on certain factors. Arrest contain 0 and 1 value representing if the suspected criminals in the specific case have been arrested or not.

```

1 #Using Pearson Correlation
2 plt.figure(figsize=(12,10))
3 cor = df.corr()
4 sns.heatmap(cor, annot=True, cmap=plt.cm.RdYlGn)
5 plt.show()

```

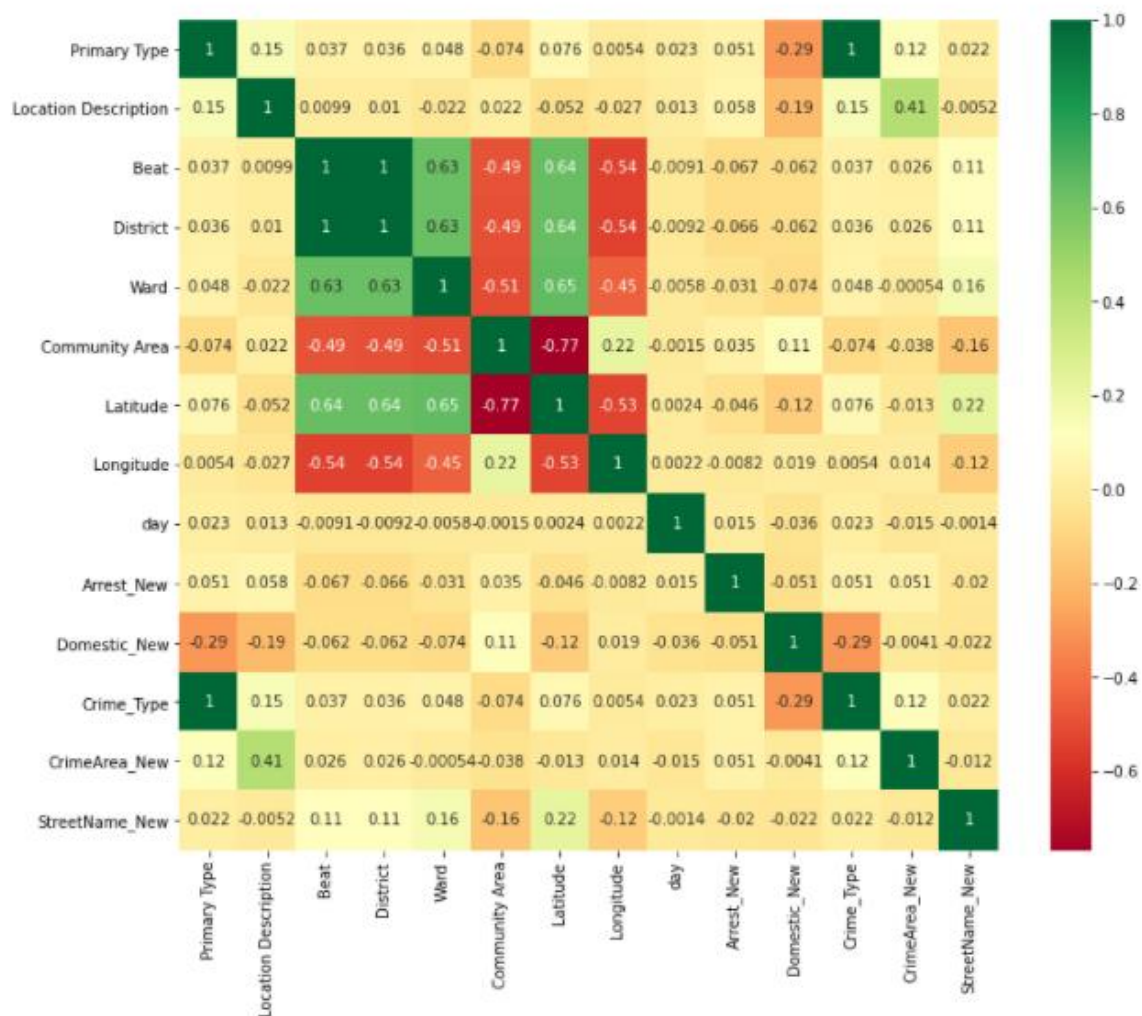


Figure 36 - Pearson correlation to show the correlation between variables

The correlation between the variable is made with a Pearson correlation plot which shows the correlation index between variables and how likely it is connected. The correlation index run between -1 to 1 negative value representing it is very unlikely or zero possibility if the columns are related to each other. On the other hand, a positive value representing the likeliness of a column to have a relationship with other columns.

```
[ ] 1 # check if any column are having correlation coefficient value of more than 0.9
2
3 def correlation_check(df,val):
4     corr_matrix=df.corr()
5     features=set()
6     for i in range(len(corr_matrix)):
7         for j in range(i):
8             if corr_matrix.iloc[i,j]>0.9:
9                 features.add(corr_matrix.columns[i])
10    return features

[ ] 1 # more than 0.9 correlation coefficient check
2
3 columns=correlation_check(X,0.9)
4 columns

{'Crime_Type', 'District'}

[ ] 1 # remove 'Crime Type' and 'District' column as it is highly correlated with other column
2
3 X.drop(columns,axis=1,inplace=True)
```

Figure 37 - dropping of columns with a correlation greater than 0.9

Based on the Pearson correlation plot, it can be seen that there are columns with more than 0.9 correlation coefficients with each other. As they are likely to have a relationship, we are going to drop the column as it might affect the results later.

```
[ ] 1 X.shape

(19552, 11)
```

Figure 38 - new shape of a dataset

The new shape of the dataset after the processing for model construction is now 19,552 rows and 11 columns.

```
1 # Scale
2 from sklearn import preprocessing
3 X = preprocessing.scale(X)

[ ] 1 # Feature Scaling
2 from sklearn.preprocessing import StandardScaler
3 ss=StandardScaler()
4 X_norm=ss.fit_transform(X)
```

Figure 39 – feature scaling and normalization

The feature scaling and normalisation is being made to transform the dataset to be able to process the model later. The processing of the dataset is using pre-processing and standard scaler from the sklearn library in Python.

```
[ ] 1 # cross validation
    2
    3 from sklearn.model_selection import train_test_split
    4 X_train,X_test,y_train,y_test=train_test_split(X_norm,y,test_size=0.2,random_state=0)
    5 X_train.shape,X_test.shape

((15641, 11), (3911, 11))

[ ] 1 # train and test row number
    2
    3 y_train.shape,y_test.shape

((15641,), (3911,))
```

Figure 40 - cross-validation and splitting of the dataset

The dataset is being split into train and test datasets with the ratio of 80:20 of train and test split. The new dataset shape for the training dataset is 15,641 rows and 11 columns of data and 3,911 rows and 11 columns of data for the test dataset. The train test cross-validation is used to compare and test the results of the model later. Therefore, after the model construction processing is done, the dataset is now ready to be used and implemented with predictive models as being mentioned above.

Part 2: Individual Component – Model Building

2.2 Model Selection

2.2.1 Random Forest

Modelling

```
[71] #importing Model
      from sklearn.ensemble import RandomForestClassifier
```

Figure 41 Importing Model

Using Sklearn library, RandomForestClassifier was imported.

```
[43] #Definig model and fitting to training data
      rfreg = RandomForestClassifier(n_estimators = 10, random_state=0)

      rfreg.fit(X_train, y_train)
```

Figure 42 Define model and fitting to training data

Declaring Random Forest Model and fitting it to the training data.

```
[72]
      # Model Evaluation metrics
      from sklearn.metrics import accuracy_score, recall_score, precision_score, f1_score
      print("==== BEFORE TUNING ==== \n")
      print(f'Train Accuracy -      : {rfreg.score(X_train, y_train):.6f}')
      print('Test Accuracy Score   : ' + str(accuracy_score(y_test, y_predict_rfr)))
      print('Precision Score       : ' + str(precision_score(y_test, y_predict_rfr)))
      print('Recall Score          : ' + str(recall_score(y_test, y_predict_rfr)))
      print('F1 Score              : ' + str(f1_score(y_test, y_predict_rfr)))

      # Random forest Classifier Confusion matrix
      from sklearn.metrics import confusion_matrix, classification_report
      print('\nConfusion Matrix : \n' + str(confusion_matrix(y_test, y_predict_rfr)))
```

Figure 43 Model Evaluation Matrix

The above screenshot shows codes of printing evaluation metrics such as train data accuracy, test data accuracy, precision score, recall score and F1 score before tuning of the model. The above code also prints confusion matrix.

```

==== BEFORE TUNING ====

Train Accuracy -      : 0.982418
Test Accuracy Score   : 0.8532344668882639
Precision Score       : 0.7954144620811288
Recall Score          : 0.4961496149614962
F1 Score              : 0.6111111111111112

Confusion Matrix :
[[2886  116]
 [ 458  451]]

```

Figure 44 Results for evaluation before tuning

The results show that training accuracy calculated for Random Forest was 0.98 whereas Test accuracy was 0.85. The precision score is 0.79. Recall score calculated was 0.49 and lastly the F1 score was 0.61. These results shown in the above figure were calculated before the tuning of the model.

```

[ ] #TUNING
    from sklearn.model_selection import GridSearchCV

    n_estimators = [int(x) for x in np.linspace(start = 10, stop = 80, num = 10 )]
    max_features = ['auto', 'sqrt']
    max_depth = [2, 4]
    min_samples_split = [2,5]
    min_samples_leaf = [1,2]
    bootstrap = [True , False]

```

Figure 45 Tuning

Using sklearn library, GridSearchCV was imported. Firstly, all the individual hyperparameters were declared. Random Forest algorithm is groups of trees. n_estimators determine the number of trees in the random forest model (Saxena, 2020). In this case generator was used generate numbers between 10 and 80 and only 10 of them will be selected to set the number of trees.

Max_features are the number of features that random forest model is allowed to use in separate trees. 'auto' selects all the features in each tree that makes sense. No restrictions are put on tree when auto method is selected. When sqrt method is selected, square root of total number of features is taken in individual run. (Srivastava, 2015)

Max_depth is the maximum depth of a tree in random forest model, and it shows how long the path between root node and leaf node can be (Saxena, 2020). In this case the maximum depth can either be 2 or 4.

The minimum observations required in any node in the decision tree of the random forest model are set by the parameter : Min_sample_split in this case the minimum sample set has been set to 2 or 5. Increasing this parameter stops the model from overfitting. (Saxena, 2020).

Minimum number of the samples after splitting a decision tree node in the random forest are identified by the Min_samples_leaf parameter. In this case this parameter has the values of 1 or 2. (Saxena, 2020).

Bootstrap parameter is used so that the samples are taken from original data with the replacement if bootstrap is true. If it is false then the sample taken and the original training data size is the same. (StackExchange, 2018)

```
# Putting parameters into a grid
param_grid = {'n_estimators' : n_estimators,
              'max_features' : max_features,
              'max_depth' : max_depth,
              'min_samples_split' : min_samples_split,
              'min_samples_leaf' : min_samples_leaf,
              'bootstrap' : bootstrap,

              }

print(param_grid)
random_grid = param_grid
```

Figure 46 Defining parameters

The above figure shows how parameters were put into parameter grid using dictionary.

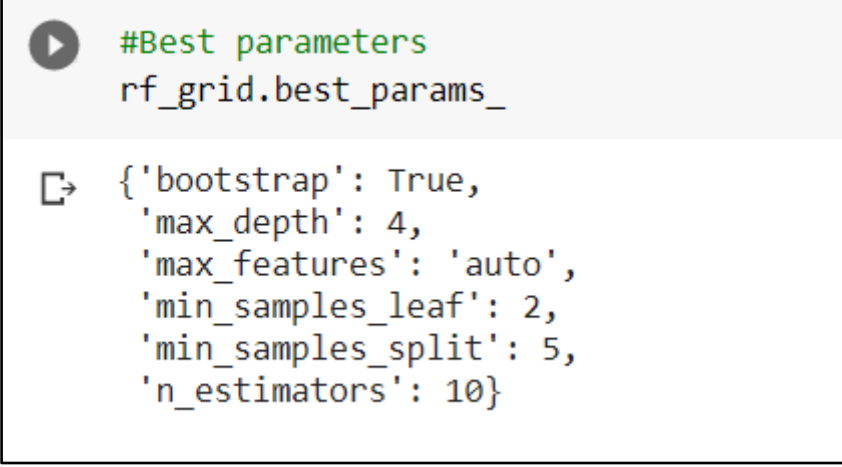
```
#loading Random Forest Model
rfreg2 = RandomForestClassifier()

[ ] #Generating GridSearchCV instance
rf_grid = GridSearchCV(estimator= rfreg2, param_grid= random_grid, cv = 3, verbose = 2, n_jobs=4)

[ ] #Fitting training model to Random Forest model using GridSearchCV
rf_grid.fit(X_train, y_train)
```

Figure 47 Training models on selected parameters

The above screenshot shows that Random Forest Model was loaded. After which GridSearchCV was generated where rfreg2 has been taken as the first argument that is the estimator, second argument is the parameter grid, verbose argument shows the details when the algorithm is running and the n_jobs will help running the GridSearchCV first so that all the combinations are run properly. Finally, GridSearch is fit to the train data.

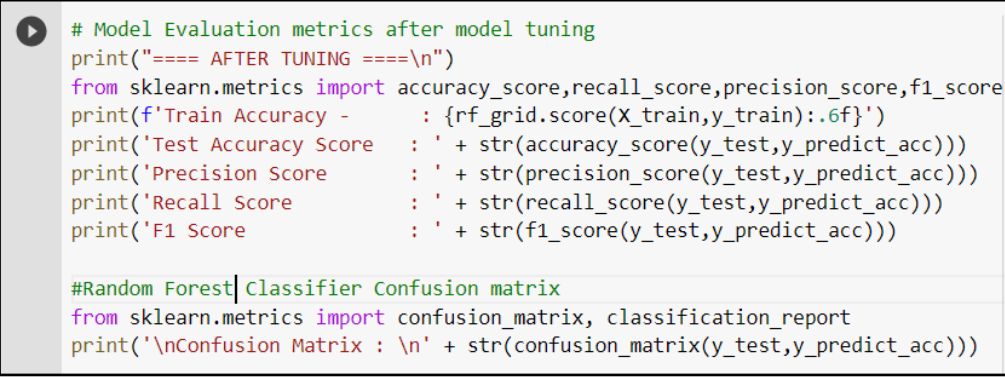


```
#Best parameters
rf_grid.best_params_

{'bootstrap': True,
 'max_depth': 4,
 'max_features': 'auto',
 'min_samples_leaf': 2,
 'min_samples_split': 5,
 'n_estimators': 10}
```

Figure 48 Best Parameters selected by gridsearch CV for tuning

The above figure shows the best parameters in the combinations that were provided previously. These parameters were used to build the final model



```
# Model Evaluation metrics after model tuning
print("==== AFTER TUNING ====\n")
from sklearn.metrics import accuracy_score, recall_score, precision_score, f1_score
print(f'Train Accuracy - : {rf_grid.score(X_train, y_train):.6f}')
print('Test Accuracy Score : ' + str(accuracy_score(y_test, y_predict_acc)))
print('Precision Score : ' + str(precision_score(y_test, y_predict_acc)))
print('Recall Score : ' + str(recall_score(y_test, y_predict_acc)))
print('F1 Score : ' + str(f1_score(y_test, y_predict_acc)))

#Random Forest Classifier Confusion matrix
from sklearn.metrics import confusion_matrix, classification_report
print('\nConfusion Matrix : \n' + str(confusion_matrix(y_test, y_predict_acc)))
```

Figure 49 Evaluation after tuning model

The above screenshot shows codes of printing evaluation metrics such as train data accuracy, test data accuracy, precision score, recall score and F1 score after tuning of the model. The above code also prints confusion matrix.

```

==== AFTER TUNING ====

Train Accuracy -      : 0.830765
Test Accuracy Score   : 0.8261314241881872
Precision Score       : 0.8566978193146417
Recall Score          : 0.3025302530253025
F1 Score              : 0.4471544715447154

Confusion Matrix :
[[2956  46]
 [ 634 275]]

```

Figure 50 Evaluation Matrix after tuning

After tuning the model, it is seen in the above screenshot that the Train Accuracy has decreased to 0.83 along with the Test Accuracy to 0.82. Whereas the precision score increased to 0.85 after tuning of the model. After tuning, the recall score and F1 scored seemed to decrease from 0.49 to 0.30 and 0.61 to 0.44 respectively.

Evaluation

```

==== BEFORE TUNING ====

Train Accuracy -      : 0.982418
Test Accuracy Score   : 0.8532344668882639
Precision Score       : 0.7954144620811288
Recall Score          : 0.4961496149614962
F1 Score              : 0.6111111111111112

Confusion Matrix :
[[2886 116]
 [ 458 451]]

```

Figure 51 - Evaluation Matrix chosen before tuning

Accuracy value shows which percentage of the values were predicted correctly. Before, tuning the accuracy was 85.3% which is relatively good. The train accuracy was calculated to make sure that there is no overfitting. However, a train accuracy of 98.2% shows that the model before tuning was overfitted. Recall refers to the true positive values calculated by the model which is at 49.6%.

```

==== AFTER TUNING ====

Train Accuracy -      : 0.830765
Test Accuracy Score   : 0.8261314241881872
Precision Score       : 0.8566978193146417
Recall Score          : 0.3025302530253025
F1 Score              : 0.4471544715447154

Confusion Matrix :
[[2956  46]
 [ 634 275]]

```

Figure 52 - Evaluation matrix after tuning

After tuning the accuracy of the model along with recall is seen to have decreased at the percentage of 82.6% and 30.2% respectively. However, a improvement can be seen in the train accuracy which has gotten lower and is of the percentage of 83.1% this means that the tuning treated the overfit model as the train percentage has become more realistic than before tuning. Therefore, it can be concluded that the model has been tuned by the parameters in a way that it prevents the model from overfitting. The accuracy of 82.6% may be the actual accuracy of the model as the data is not overfit.

2.2.2 Logistic Regression (

Original Model building

▼ **LOGISTIC REGRESSION** - Aisha Binti Muhammad Fakhar Iqbal TP050393

```

[ ] from sklearn.linear_model import LogisticRegression
    from sklearn import metrics
    from sklearn.metrics import accuracy_score, recall_score, precision_score, f1_score, mean_squared_error
    classification_report, confusion_matrix, r2_score

```

Figure 53 Import necessary components

Importing Necessary library parameters and evaluation matrix.

```

#Logistic Regression
logreg = LogisticRegression()

#Training Logistic Regression Model
logfit = logreg.fit(X_train, y_train)

#Predicting on X_test (predicted y value)
y_pred = logreg.predict(X_test)

```

Figure 54 Default Logistic Regression Model

```
#Generating Classification Report for logistic regression model
print(classification_report(y_test,y_pred))

#f1-score shows that 1 is not being predicted by the model - suggests imbalance in Data.
```

	precision	recall	f1-score	support
0	0.77	1.00	0.87	3002
1	0.00	0.00	0.00	909
accuracy			0.77	3911
macro avg	0.38	0.50	0.43	3911
weighted avg	0.59	0.77	0.67	3911

Figure 55 evaluating model through classification report

The classification table above shows that the original logistic regression data is not predicting occurrences of 1. This may suggest a class imbalance This can be further confirmed by the bar chart below:



Figure 56 Imbalance in target variable data

Therefore, it can be said that there may be an imbalance of data that may cause biased results in favour of (0) “No Arrest” when conducting a logistic regression because there is not enough data for (1) “Arrest”. To fix this resampling of data is necessary.

Over Sampling for Logistic Regression Model

SMOTE or Synthetic Minority Over-Sampling Technique will be used to oversample the data. Synthetic data will be generated for (1) the “Arrested” category which is the minority class. The

(1) “Arrested” category is around 22% of the whole dataset. Therefore, this can be considered as a mild imbalance of data. (Wijaya, 2020)

```
#Over Sampling done using SMOTE Technique - for building a better prediction model
from imblearn.over_sampling import SMOTE

# Seed used by the random number generator set to 0 - random gen needs a number to start generating numbers from
sm = SMOTE(random_state = 0)

#Oversampling training data only - test set to be used for predictions
X_train_os, y_train_os = sm.fit_sample(X_train, y_train)

#ploting oversampled trained data to see no imbalance
sns.countplot(y_train_os).set(title='Oversampled Arrest_New', xlabel='(0) Not Arrested / (1) Arrested', ylabel='Count')
```

Figure 57 SMOTE used for oversampling

SMOTE used and random state set to 0 so that the algorithm can select numbers 0 onwards for random sampling.

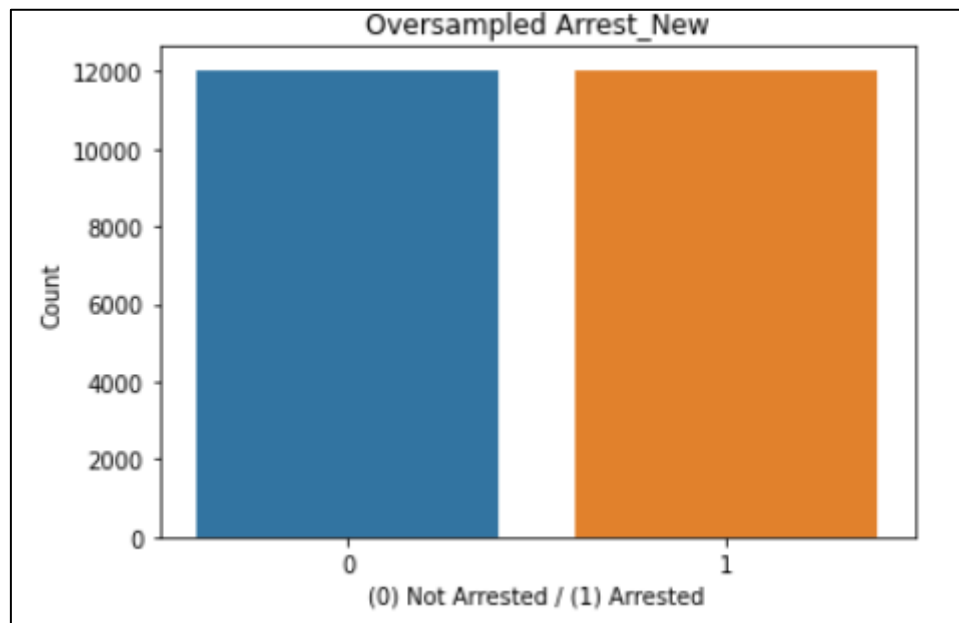


Figure 58 Data no longer imbalanced

Bar chart showing that data is no longer imbalanced

```
#creating object for Logistic Regression with default parameters
os_logreg = LogisticRegression()

# Training Logistic Regression with oversampled train data
logfit = logreg.fit(X_train_os, y_train_os)

#viewing default parameters of os_logreg
logfit

LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                    intercept_scaling=1, l1_ratio=None, max_iter=100,
                    multi_class='auto', n_jobs=None, penalty='l2',
                    random_state=None, solver='lbfgs', tol=0.0001, verbose=0,
                    warm_start=False)
```

Figure 59 Logistic Regression with Over Sampled Data and default Parameters

Logistic Regression with Over Sampled Data and default Parameters

```
#Predicting on X_test (predicted y value)
y_pred_os = logreg.predict(X_test)
```

Figure 60 Fitting model

```
# Generating classification report on oversampled data
print('**** AFTER SAMPLING & BEFORE TUNING ****')
print('*****')
print(" ")

print(classification_report(y_test,y_pred_os))

# The report shows improvement in predicting 1 as compared to logistic regression built with imbalanced data
# Accuracy however is low at 56.2%
```

```
**** AFTER SAMPLING & BEFORE TUNING ****
*****
```

	precision	recall	f1-score	support
0	0.81	0.56	0.66	3002
1	0.28	0.58	0.38	909
accuracy			0.56	3911
macro avg	0.55	0.57	0.52	3911
weighted avg	0.69	0.56	0.60	3911

Figure 61 Classification Table before tuning

```
# Displaying Accuracy, Precision, Recall and F1 Score in more detail
print('**** AFTER SAMPLING & BEFORE TUNING ****')
print('*****')
print(" ")

print('Accuracy Score : ' + str(accuracy_score(y_test,y_pred_os)))
print('Precision Score : ' + str(precision_score(y_test,y_pred_os)))
print('Recall Score : ' + str(recall_score(y_test,y_pred_os)))
print('F1 Score : ' + str(f1_score(y_test,y_pred_os)))

**** AFTER SAMPLING & BEFORE TUNING ****
*****
```

```
Accuracy Score : 0.5620046024034774
Precision Score : 0.2827027027027027
Recall Score : 0.5753575357535754
F1 Score : 0.37912287060529176
```

Figure 62 Accuracy, Precision, Recall, F1 Score before tuning

High accuracy and a high recall value are good. Above the Accuracy score is 56% which is not good but also not bad as it is still more than 50% score and the recall is 57% which is also a

good or bad score as it shows that 57% of the data was captured as actual positive. As Recall is predicted for the (1) class label it shows that some predictions are being made on that class label as oversampling was applied.

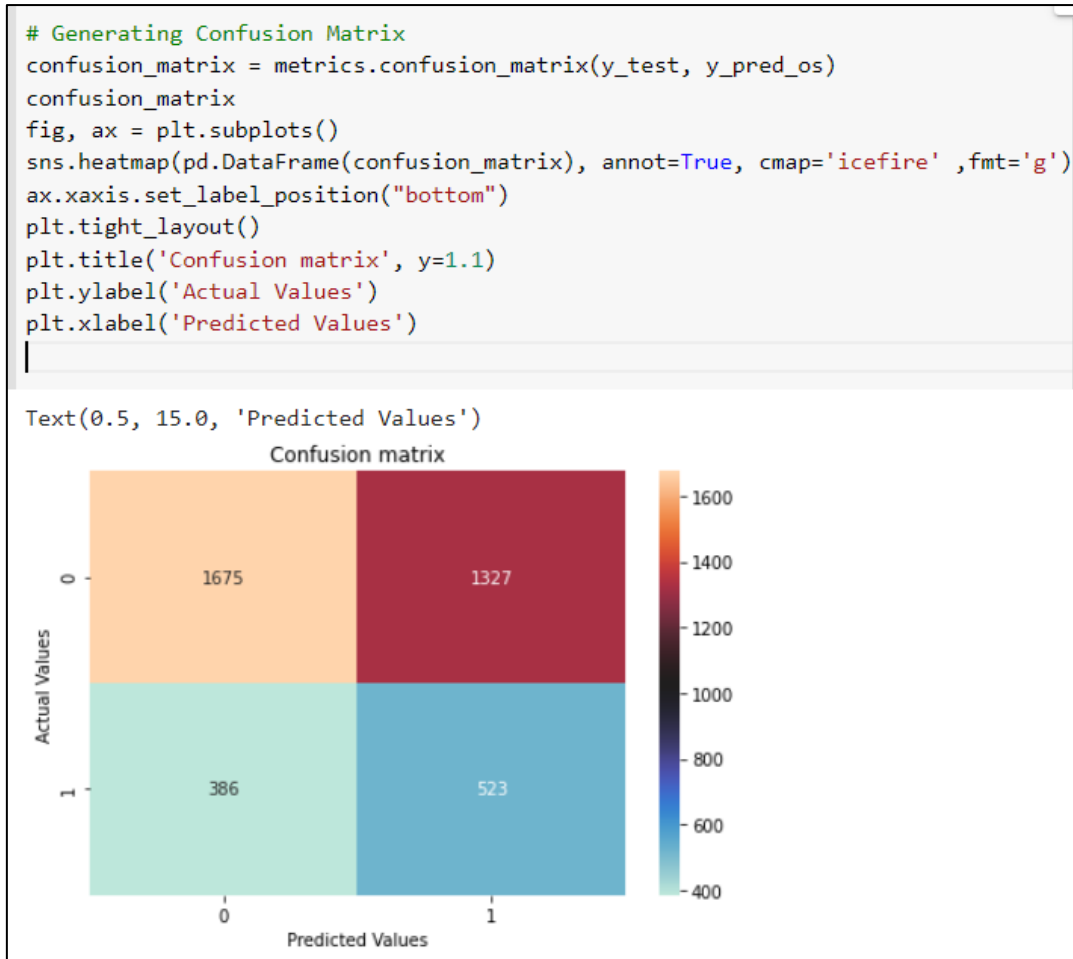


Figure 63 Confusion Matrix - Log Reg with Over Sampled Data.

Confusion Matrix	Result
True Negative Value:	1675
True Positive Value:	523
False Negative Value	386
False Positive Value:	1327
Correctly Predicted Values:	2198
Incorrectly Predicted Values:	1713


```

TN = confusion_matrix[0,0]
TP = confusion_matrix[1,1]
FN = confusion_matrix[1,0]
FP = confusion_matrix[0,1]

print('**** AFTER SAMPLING & BEFORE TUNING ****')
print('*****')
print(" ")
print('Model Accuracy: ', (TP+TN)/(TP+TN+FP+FN))
print('Misclassification Rate:', 1-(TP+TN)/(TP+TN+FP+FN))
print('True Positive Rate:', TP/(TP+FN))
print('True Negative Rate:', TN/(TN+FP))
print('Positive Predicted Value:', TP/(TP+FP))
print('Negative Predicted Value:', TN/(TN+FN))

```

Figure 64 Evaluation Matrix related to Confusion Matrix

```

**** AFTER SAMPLING & BEFORE TUNING ****
*****

Model Accuracy: 0.5620046024034774
Misclassification Rate: 0.4379953975965226
True Positive Rate: 0.5753575357535754
True Negative Rate: 0.5579613590939374
Positive Predicted Value: 0.2827027027027027
Negative Predicted Value: 0.8127122755943716

```

Figure 65 Results of Confusion Matrix

Accuracy as shown before is 56.2% so that is how accurate the model is. A misclassification rate of 43.8% shows that 43.8% of the values were misclassified. This is not a very good value as it shows that the model is not very accurate.

True Positive Rate or Recall is 57.5% which is slightly closer to 100% or 1.00. This is not the worst value, but it is not the best either as the best value would be closer to 1 whereas the worst is closer to 0. This value means that 57.2% of the values were predicted as correct positive predictions.

The negative rate or rate of the correct negative predicted value is 55.8% which is slightly worse than the Recall value means that 55.8% of the values were correctly predicted as negative by the model. However, this is not the best value as it is closer to 50% than to 100% or 1.00.

Positive predicted value or Precision is the probability of correct positive predicted values, the value is 28.3% this is not a good value as it is closer to 0 than to 1 or 100%. The probability of a correct positive prediction that is closer to 0 is not good.

The negative predicted value is the number of correct negative predicted values, the value is 81.3% which is closer to 100% is the best value. Meaning the model is more likely to predict a correct negative value than a correct positive value.

```
#Evaluation Matrix
print('**** AFTER SAMPLING & BEFORE TUNING ****')
print('*****')
print(" ")
print('Model Accuracy: ', (TP+TN)/(TP+TN+FP+FN))
print('Misclassification Rate:', 1-(TP+TN)/(TP+TN+FP+FN))
print('Mean Squared Error: ', metrics.mean_squared_error(y_test, y_pred_os))
print('Root Mean Squared Error: ', np.sqrt(metrics.mean_squared_error(y_test, y_pred_os)))
print('R-Squared: ', metrics.r2_score(y_test, y_pred_os))

**** AFTER SAMPLING & BEFORE TUNING ****
*****

Model Accuracy: 0.5620046024034774
Misclassification Rate: 0.4379953975965226
Mean Squared Error: 0.4379953975965226
Root Mean Squared Error: 0.6618122071981769
R-Squared: -1.4551080357869224
```

Figure 66 Chosen evaluation matrix for Oversampled Data Log Reg

Mean Square Error of 43.8% is a value that is closer to 0 than to 1 or 100% therefore, it is a good mean square error. If the mean square error is high, it means that there is a high error in the model predictions. there is only a 43.8% difference between actual and predicted values.

RMSE value can be seen to be closer to 1 than to 0 at 0.66 means that it is not a good model.

R Squared is the measure of variance between actual values and predicted values. However, the R squared value here is negative, therefore, this means that the model fit is not good.

Meta parameter selection - Logistic Regression

```
# Logistic Regression for tuning
t_logreg = LogisticRegression()
```

Figure 67 Logistic Regression model declared for model tuning

```
# Parameters selected for Grid Search to go through and find best ones
param_grids = [
    {
        'penalty':['l1','l2','elasticnet','none'],
        'C': np.logspace (-4,4,20),
        'solver':['lbfgs','newton-cg','liblinear', 'sag','saga'],
        'class_weight':[{0:0.6, 1:0.4}],
    }
]
```

Figure 68 Parameters mentioned from which GridSearchCV

```
from sklearn.model_selection import GridSearchCV
grid = GridSearchCV(t_logreg, param_grid = param_grids, cv = 3, verbose = True, n_jobs = -1)
```

Figure 69 Performing GridSearch CV

```
best_grid = grid.fit(X_train_os,y_train_os)

Fitting 3 folds for each of 400 candidates, totalling 1200 fits
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 2 concurrent workers.
[Parallel(n_jobs=-1)]: Done 164 tasks      | elapsed:    3.8s
[Parallel(n_jobs=-1)]: Done 764 tasks      | elapsed:   21.7s
[Parallel(n_jobs=-1)]: Done 1200 out of 1200 | elapsed:   35.6s finished
```

Figure 70 Fitting model to see best results

```
best_grid.best_estimator_

LogisticRegression(C=0.00026366508987303583, class_weight={0: 0.6, 1: 0.4},
                    dual=False, fit_intercept=True, intercept_scaling=1,
                    l1_ratio=None, max_iter=100, multi_class='auto', n_jobs=None,
                    penalty='l2', random_state=None, solver='liblinear',
                    tol=0.0001, verbose=0, warm_start=False)
```

Figure 71 Best Parameters selected by GridSearchCV for Logistic Regression.

“C” value needs to be a float value. Np.logspace was used to set a range of values that the “C” value could choose from. To encourage high regularization, the lowest “C” value that gives the best model will be selected by the Grid Search CV.

“class_weight” class weight refers to the weighing that will be applied to class labels such as (0) Not Arrest and (1) Arrest. Then when the model fitting is done the negative log-likelihood will be applied to both class labels. Class weight of “balanced” was also tested but did not give the best result. The above class weight was chosen through some trial and error, testing out different class weights which gives the best result in terms of accuracy. Setting a class weight for logistic regression can improve the performance of the model as compared to a model that has no setting of a class weight (Brownlee, 2020)

“Penalty” refers to the regularization of the logistic regression model based on l1, l2.

L1 also known as Lasso Regression uses the L1-norm function which can also be called the LAD of least absolute deviation. When L1 regulation is used, the model is less likely to overfit. Thus, creating better predictions. (Aditya, 2018)

L2 also known as Ridge Regression uses the L2 norm function also called LSE or least square error. This has been chosen by the grid search below as the best penalty to be applied to the regression model for the best result. (Aditya, 2018)

$$w^* = \text{minimization of } \sum_i \ln[1 + \exp(-z_i)] + \lambda \sum (w_j)^2$$

$\sum (w_j)^2$ is a **regularization term** and $\sum [\ln(1 + \exp(-z_i))]$ is the **Loss term**.
 λ is a hyper parameter.

Figure 72 (Aditya, 2018)

As the Grid search has selected L2, it will make sure that the model doesn't overfit.

“Solver” many solvers have been listed in the parameters to be selected from by the Grid Search CV. This parameter is used to optimize the logistic regression model. Liblinear has been chosen by the Grid Search CV as this dataset is not very large. (Hale, 2019)

Model tuning - Logistic Regression

Grid Search CV was used to increase the accuracy of the model. Grid Search CV is used to perform hyperparameter tuning for the logistic regression model. Grid Search CV goes through predefined parameters which are defined using the logistic regression dictionary. This function goes through mentioned hyperparameters and selects the best parameters for the best prediction results for the model.

Below the best parameters chosen by the grid search CV have been applied to the logistic regression.

```
# Applying selected Parameters to Logistic Regression

t_logreg = LogisticRegression(C=0.00026366508987303583, class_weight={0: 0.6, 1: 0.4},
                             dual=False, fit_intercept=True, intercept_scaling=1,
                             l1_ratio=None, max_iter=100, multi_class='auto', n_jobs=None,
                             penalty='l2', random_state=None, solver='liblinear',
                             tol=0.0001, verbose=0, warm_start=False)

fit = t_logreg.fit(X_train_os, y_train_os)
```

Figure 73 Applying best parameters to logistic regression.

```
# Training Logistic Regression with new predicted value of y on tuned model
new_y_pred = best_grid.predict(X_test)
```

Figure 74 fitting a logistic regression model

```
#Calculating f1 score, precision and recall
print('**** AFTER TUNING ****')
print('*****')
print(" ")

print(classification_report(y_test,new_y_pred))
```

```
**** AFTER TUNING ****
*****
```

	precision	recall	f1-score	support
0	0.78	0.92	0.84	3002
1	0.36	0.15	0.21	909
accuracy			0.74	3911
macro avg	0.57	0.53	0.53	3911
weighted avg	0.68	0.74	0.70	3911

Figure 75 Classification Report of tuned model

The classification report for the tuned model can be seen above.

```
print('**** AFTER TUNING ****')
print('*****')
print(" ")
acc = metrics.accuracy_score(y_test, new_y_pred)
print("Accuracy:", acc)
print("Precision:",metrics.precision_score(y_test, new_y_pred))
print("Recall:",metrics.recall_score(y_test, new_y_pred))
print("Misclassification Score", 1 - acc)
```

```
**** AFTER TUNING ****
*****
```

```
Accuracy: 0.7386857581181283
Precision: 0.3554987212276215
Recall: 0.15291529152915292
Misclassification Score 0.2613142418818717
```

Figure 76 Accuracy Precision Recall listed for the tuned model.

It can be seen that tuning the model has made the accuracy of the model 73.8% this is a good accuracy score and means that tuning the model has boosted the accuracy of the model. Recall can be seen to have decreased however to 15.3%.

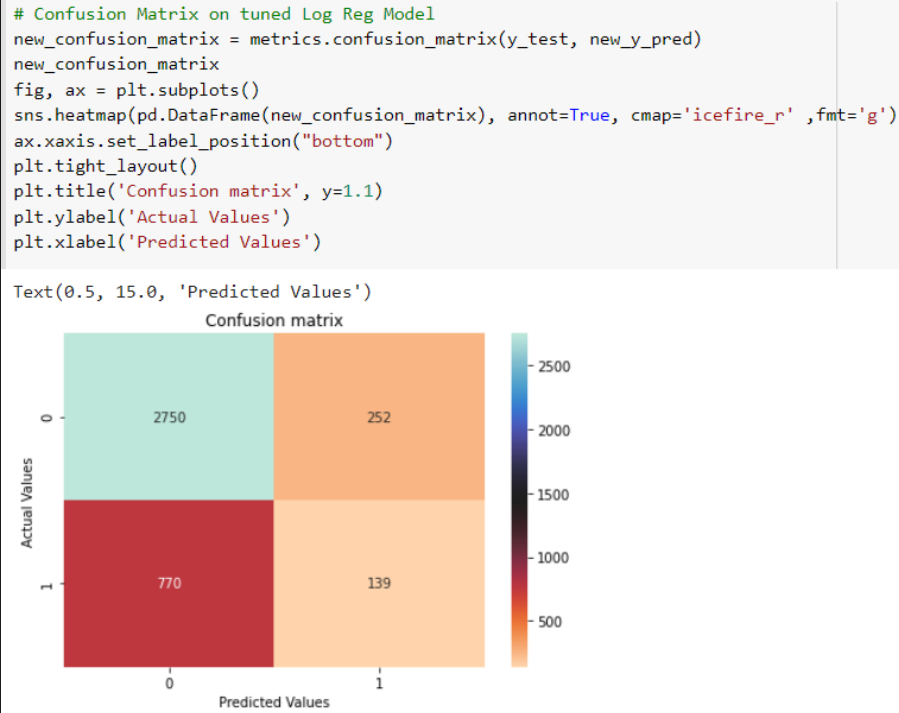


Figure 77 Confusion Matrix for Tuned model.

Confusion Matrix	Result
True Negative Value:	2750
True Positive Value:	139
False Negative Value	770
False Positive Value:	252
Correctly Predicted Values:	2889
Incorrectly Predicted Value:	1020

```
] print('**** AFTER TUNING ****')
print('*****')
print(" ")
print ('Model Accuracy: ', (TP_t+TN_t)/(TP_t+TN_t+FP_t+FN_t))
print ('Misclassification Rate:', 1-(TP_t+TN_t)/(TP_t+TN_t+FP_t+FN_t))
print ('True Positive Rate:', TP_t/(TP_t+FN_t))
print ('True Negative Rate:', TN_t/(TN_t+FP_t))
print ('Positive Predicted Value:', TP_t/(TP_t+FP_t))
print ('Negative Predicted Value:', TN_t/(TN_t+FN_t))

**** AFTER TUNING ****
*****

Model Accuracy: 0.7386857581181283
Misclassification Rate: 0.2613142418818717
True Positive Rate: 0.78125
True Negative Rate: 0.3554987212276215
Positive Predicted Value: 0.916055962691539
Negative Predicted Value: 0.15291529152915292
```

Figure 78 Results

The misclassification rate has decreased from before, meaning the tuned model has fewer misclassified values as the misclassification rate is now 26.1%. Therefore, the model is more accurate now.

True positive rate or recall has also increased to 78.1% which means that 78.1% of the values were predicted as correct positive predictions.

The negative rate value is 15.3% which is worse than the Recall value means that 15.3% of the values were correctly predicted as negative by the model.

The positive predicted value is 91.6% the probability of correct positive predicted values. This is closer to 100% or 1.00 than 0 so this is the best correct positive predictions probability. As compared to the 15.3% probability of a negative predicted value.

```
# Evaluation Matrix
print('**** AFTER TUNING ****')
print('*****')
print(" ")
print('Model Accuracy: ', (TP_t+TN_t)/(TP_t+TN_t+FP_t+FN_t))
print('Misclassification Rate:', 1-(TP_t+TN_t)/(TP_t+TN_t+FP_t+FN_t))
print('Mean Squared Error: ', metrics.mean_squared_error(y_test, new_y_pred))
print('Root Mean Squared Error: ', np.sqrt(metrics.mean_squared_error(y_test, new_y_pred)))
print('R-Squared: ', metrics.r2_score(y_test, new_y_pred))

**** AFTER TUNING ****
*****

Model Accuracy:  0.7386857581181283
Misclassification Rate: 0.2613142418818717
Mean Squared Error:  0.26131424188187163
Root Mean Squared Error:  0.5111890471067153
R-Squared:  -0.464752138105216
```

Figure 79 Evaluation Matrix for Tuned model.

The mean square value has decreased after model tuning. This is a good sign as it means that there is a low error when building the model. There is only a difference of 26.1% between original and predicted values. RMSE value is 0.51 so there is an improvement. R square value has become closer to 1 therefore, the model is now better fitted than before.

Model Evaluation - Logistic Regression

	precision	recall	f1-score	support
0	0.77	1.00	0.87	3002
1	0.00	0.00	0.00	909
accuracy			0.77	3911
macro avg	0.38	0.50	0.43	3911
weighted avg	0.59	0.77	0.67	3911

Figure 80 Log Reg before Sampling or Tuning

```

**** AFTER SAMPLING & BEFORE TUNING ****
*****

```

	precision	recall	f1-score	support
0	0.81	0.56	0.66	3002
1	0.28	0.58	0.38	909
accuracy			0.56	3911
macro avg	0.55	0.57	0.52	3911
weighted avg	0.69	0.56	0.60	3911

Figure 81 Log Reg classification table after sampling

```

**** AFTER TUNING ****
*****

```

	precision	recall	f1-score	support
0	0.78	0.92	0.84	3002
1	0.36	0.15	0.21	909
accuracy			0.74	3911
macro avg	0.57	0.53	0.53	3911
weighted avg	0.68	0.74	0.70	3911

Figure 82 Log Reg classification table after tuning

It was seen that before sampling was done the model was not predicting (1) “Arrested” cases at all as the recall value was 0. However, the model had high accuracy. But the goal of conducting a predictive model is to find out what are the possibilities for both class labels of (0) Not Arrested and (1) Arrested. Moreover, the probability of the model predicting only (0) Not Arrested was 100% which is not ideal.

After Sampling was done, there was a decrease in the accuracy of the model however, the model was now predicting some outcomes of (1) “Arrested”. Moreover, the model probability was almost similar for both classes, the probability of a 0 “Not Arrested” being predicted was 56% whereas the probability of model predicting a (1) “Arrest” was 58%. This was the model selected for tuning.

After Tuning, there was an increase in the accuracy of the model as the accuracy was now 74%. The model was now predicting outcomes of both 0 and 1 with the recall values for both. There was a 92% probability of (0) Not Arrested being predicted by the model and a 15% chance of (1) “Arrested” being predicted by the model.


```

**** AFTER SAMPLING & BEFORE TUNING ****
*****

Model Accuracy: 0.5620046024034774
Misclassification Rate: 0.4379953975965226
Mean Squared Error: 0.4379953975965226
Root Mean Squared Error: 0.6618122071981769
R-Squared: -1.4551080357869224

```

Figure 83 Evaluation Matrix Before Tuning

```

**** AFTER TUNING ****
*****

Model Accuracy: 0.7386857581181283
Misclassification Rate: 0.2613142418818717
Mean Squared Error: 0.26131424188187163
Root Mean Squared Error: 0.5111890471067153
R-Squared: -0.464752138105216

```

Figure 84 Evaluation Matrix after Tuning

Error Rates for both the models before and after tuning will be discussed here. It can be seen that the misclassification rate improved after tuning to a 26% error rate which means that the model had lesser errors in prediction than before when the error rate was 43%. Moreover, the MSE, RMSE and R2 values also got better after tuning as the values went from 44%, 66%, -1.4 to 26%, 51% and -0.4 respectively. The model demonstrated a significant improvement and both models predicted outcomes of 0 and 1. The model after tuning would be used to make better predictions of criminal arrests.

2.2.3 Decision Trees (Basmah Zahid)

Default Decision Tree

The decision tree that will be implemented is the decision tree classifier as our target for this dataset has been converted to a binary 0 or 1 variable, through label encoding. The model is first implemented with default parameters.

```

1 #importing all the libraries that will be used below
2 from sklearn.tree import DecisionTreeClassifier,tree
3
4 #oversampling class
5 from imblearn.over_sampling import SMOTE
6 #counter class to see distribution of 0 and 1 in dataset
7 from collections import Counter
8 #class for model tuning and hyper parameters
9 from sklearn.model_selection import GridSearchCV,KFold
10 #metrics for evaluation
11 from sklearn.metrics import accuracy_score,confusion_matrix,classification_report,f1_score,precision_score,recall_score,roc_curve, auc, roc_auc_score
12
13 #for graphs
14 from matplotlib import pyplot as plt
15 import itertools

```

Figure 85 - importing relevant libraries

```

1 #pure tree
2 dd = DecisionTreeClassifier()
3 dd.fit(X_train,y_train)

```

DecisionTreeClassifier(ccp_alpha=0.0, class_weight=None, criterion='gini',
max_depth=None, max_features=None, max_leaf_nodes=None,
min_impurity_decrease=0.0, min_impurity_split=None,
min_samples_leaf=1, min_samples_split=2,
min_weight_fraction_leaf=0.0, presort='deprecated',
random_state=None, splitter='best')

```

1 #Accuracy Score
2 dd_pred = dd.predict(X_test)
3 print(classification_report(y_test,dd_pred))

```

	precision	recall	f1-score	support
0	0.88	0.88	0.88	3002
1	0.60	0.61	0.60	909
accuracy			0.81	3911
macro avg	0.74	0.74	0.74	3911
weighted avg	0.82	0.81	0.81	3911

Figure 86 - default decision tree classifier model and classification table

The model is firstly fit with the split dataset of train and test and run to show its default parameters. Next, we carry out displaying its classification report which consists of its recall score at 61%, precision at 60%, accuracy at 81% and f1-score at 60%.

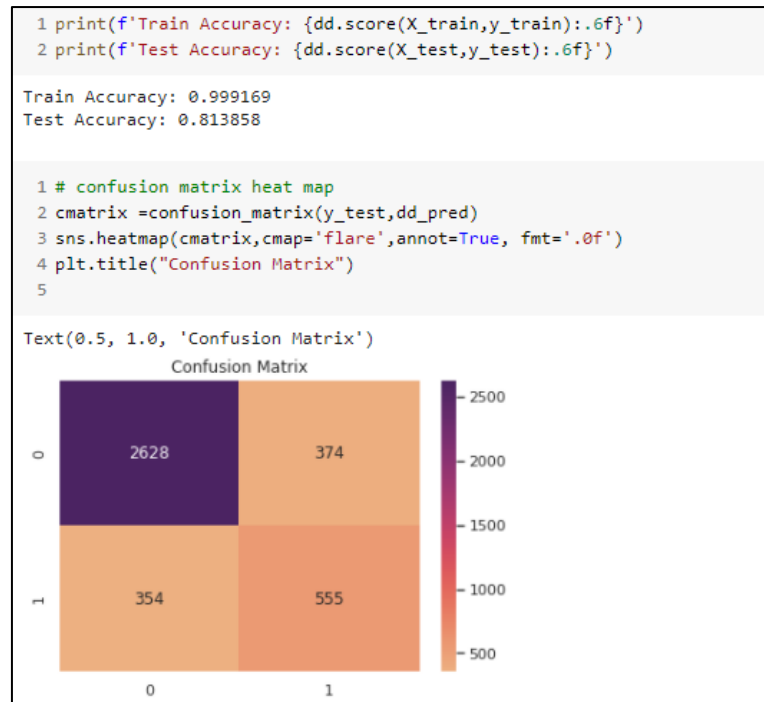


Figure 87 - Accuracy Scores and Confusion Matrix

The score of train accuracy and test accuracy is not the best and suggests overfitting of the model. Moreover, during the EDA process, it was noted that the dependent variable ‘Arrest_New’ seemed to show an unbalanced distribution. The values are approximately 0.77 and 0.23, respectively.

The confusion matrix also shows us certain values.

- **TP – True positives** > this is where the predicted is yes arrest and they do get arrested. The value for this is 555.
- **TN – True negative** > is where the model predicts no arrest, and our actual value is also no. This is 2628.
- **FP- False Positive** > also known as a type I error is where the prediction is arrest but they are not actually arrested. This is 374.
- **FN – False Negative** > also known as a Type II error. This is where the prediction is no arrest, but they were actually arrested. This is 354.

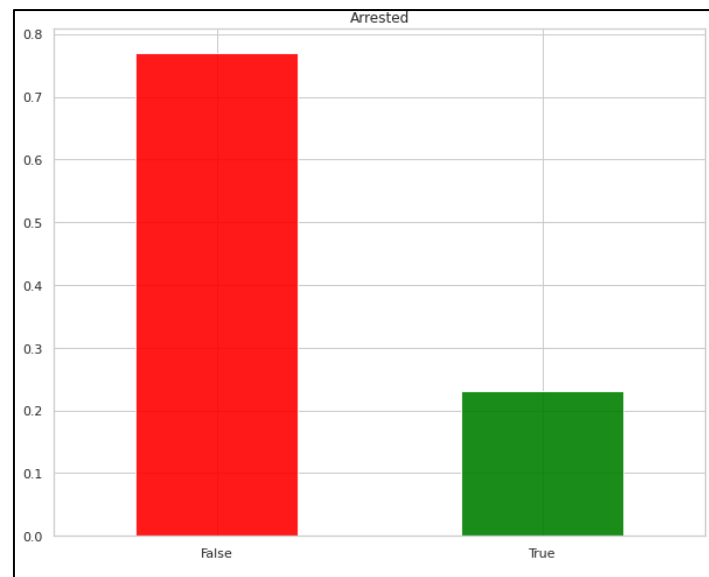


Figure 88 - Target class bar chart

```
1 #this shows an imbalance in numbers of 0 and 1 so we will do over sampling
2 print(sorted(Counter(df['Arrest_New']).items()))
[(0, 15045), (1, 4507)]
```

Figure 89 - distribution of 0 and 1

The above image shows the proof of an unbalanced dataset. This can be corrected through the use of oversampling of the dataset.

Oversampling of Unbalanced data

Oversampling is a technique where the data from minority classes is continuously duplicated and added to the training data. It is usually used when the algorithm may get affected due to skewed distribution like our dataset and is suitable for models such as decision trees. (Brownlee, 2020) Here the minority class is the 'True' or 'Arrested'.

```
#transforming dataset to more balanced dataset
#from imblearn.over_sampling import SMOTE
smote = SMOTE(random_state = 2)
X_train_oversample, y_train_oversample = smote.fit_sample(X_train, y_train.ravel())
```

Figure 90 – oversampling [SMOTE]

SMOTE is also known as 'Synthetic Minority Over-Sampling Technique'. The random_state was tested with various values such as 0, 1 and 2. 2 produced the best results in the evaluation method.

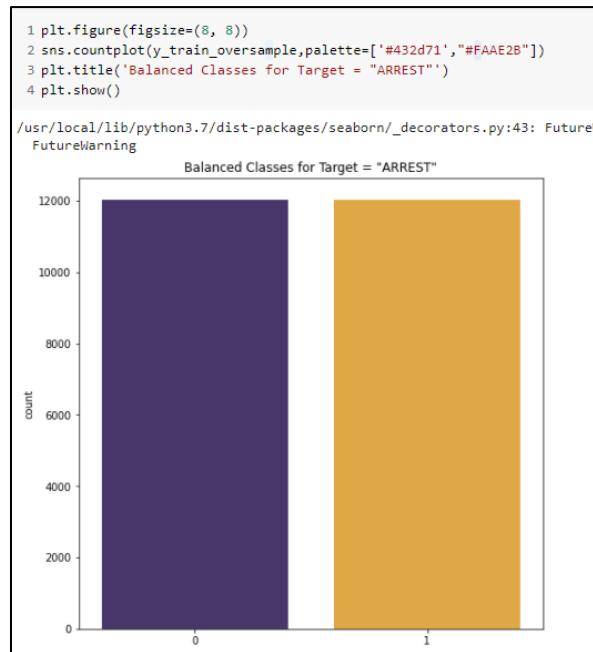


Figure 91 - bar plot showing balanced data

A bar chart displays the balanced dataset with '0' corresponding to No Arrest and '1' to Arrest.

Default Decision Tree with Oversampled Data

The decision tree with the oversampled train dataset is fit for modelling purposes. The overfitting is still there however now we can use meta parameters to tune our model to provide us better results. After oversampling, the values for these metrics remained similar with minor changes, however now hyperparameters can be used to tune the model.

```

1 #decision tree with default paramters
2 dc = DecisionTreeClassifier()
3 #fitting decision tree with over sampled train data
4 dc.fit(X_train_oversample,y_train_oversample)

```

```

DecisionTreeClassifier(ccp_alpha=0.0, class_weight=None, criterion='gini',
                        max_depth=None, max_features=None, max_leaf_nodes=None,
                        min_impurity_decrease=0.0, min_impurity_split=None,
                        min_samples_leaf=1, min_samples_split=2,
                        min_weight_fraction_leaf=0.0, presort='deprecated',
                        random_state=None, splitter='best')

```

Figure 92 - decision tree with oversampled data

```

1 #oversampled dataset's accuracy scores
2 print(f'Train Accuracy: {dc.score(X_train_oversample,y_train_oversample):.5f}')
3 print(f'Test Accuracy: {dc.score(X_test,y_test):.5f}')

```

```

Train Accuracy: 0.99946
Test Accuracy: 0.80159

```

Figure 93 - train and test accuracy

```

1 #Model Evaluation before any tuning is performed
2 #classification table, confusion matrix
3 #prediction test
4 y_pred=dc.predict(X_test)
5
6 #confusion matrix calculator for other metrics
7 conmm =confusion_matrix(y_test,y_pred)
8 TP = conmm[0,0] # true positive
9 TN = conmm[1,1] # true negatives
10 FP = conmm[0,1] # false positives
11 FN = conmm[1,0] # false negatives
12 Total = TP+TN+FP+FN
13
14 #all metrics before tuning
15 Specificity = (TN)/(TN+FP)
16 Sensitivity = (TP)/(TP+FN)
17 Misclassification = 1-(accuracy_score(y_test,y_pred))
18 print('Specificity      : ',Specificity)
19 print('Sensitivity      : ',Sensitivity)
20 print('Misclassification Rate:',Misclassification)
21 print('Accuracy        : ', accuracy_score(y_test,y_pred))
22 print('Precision       : ', precision_score(y_test,y_pred)) #positive predicted value
23 print('Recall         : ', recall_score(y_test,y_pred)) #true positive rate
24 print('F1-Score       : ', f1_score(y_test,y_pred))
25
26 #classification report
27 print(classification_report(y_test,y_pred))

```

Specificity	:	0.5667001003009027			
Sensitivity	:	0.8819492107069321			
Misclassification Rate:	:	0.19841472769112756			
Accuracy	:	0.8015852723088724			
Precision	:	0.5667001003009027			
Recall	:	0.6215621562156216			
F1-Score	:	0.5928646379853094			
		precision	recall	f1-score	support
0		0.88	0.86	0.87	3002
1		0.57	0.62	0.59	909
accuracy				0.80	3911
macro avg		0.72	0.74	0.73	3911
weighted avg		0.81	0.80	0.80	3911

Figure 94 – other metrics and classification table

We can see the accuracy has remained similar at 80.1% while precision is at 57% which can be better. Recall and F1- Score is at 62% and 59% respectively. Although the result remains slightly similar, we can now select the parameters and tune the model with the balanced data for a better result.

The misclassification rate can be seen at almost 20% which has the potential to be lower but it informs us regarding the number of records the model has not accurately predicted.

Precision informs us that the classifier has correctly classified the predictions of the class, Recall lets us know how many of the specific classes such as '1' is found over the total number of elements of the class, F1-score is the mean of the precision and recall while accuracy determines how accurate the model is.

The specificity is at 57% while sensitivity is at 88%. The former is used to evaluate the model's capability to foresee the true negative of the categories while the latter is used to assess the model's power to foresee the true positives. Both of these are used to help plot the AUC-ROC Curve. (Mitrani, 2019)

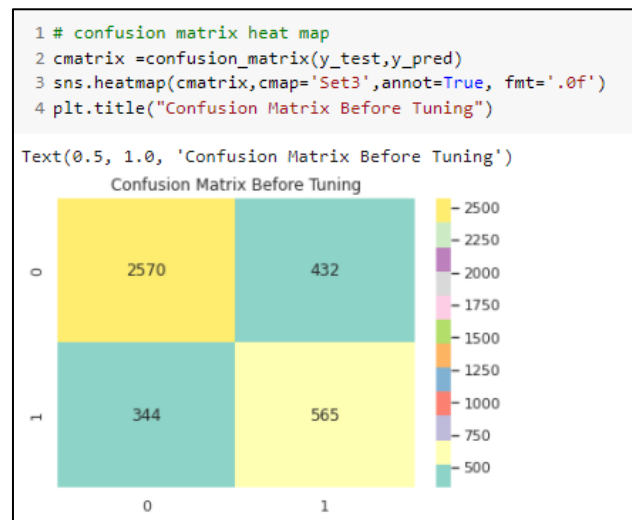


Figure 95 - confusion matrix

Meta Parameters

The first step in constructing our model will be selecting the hyperparameters. The more hyperparameters we have, the slower the tuning process hence we have gone through different types of parameters and chosen the four which work best with the model and produce the best results.

```
1 #K-fold cross validation
2 kf = KFold(n_splits =5, random_state =47)
```

Figure 96 - K fold cross-validation

The kfold is used in the grid search in the cross-validation splitting strategy or cv. We have kept it as 5 for our grid search method. It presents train/test indices for dividing data into train and test sets then divides the dataset into k consecutive folds. Each fold is then validated once, with the remaining k - 1 folds forming the training dataset. (ScikitLearn, 2020)

```
#dictionary for hyper parameters and model tuning
param_dist= {
    "criterion":["gini", "entropy"],
    "max_depth": [1,2,3,4,5,6,7,8],
    "min_samples_leaf":[2,5,10,25,50,75,100],
    "min_samples_split":[2,5,10,25,50,75,100],
}
```

Figure 97 - dictionary for grid search with hyperparameters

Param_dist is the dictionary with the list of possible parameters to tune our decision tree model. Criterion, max_depth, min_samples_leaf and min_samples_split was chosen as the best parameters to model with. Criterion is kept is either Gini or entropy. The difference between both is very minimal except for entropy taking more time to compute.

The max_depth is up to 7 due to the size of the dataset. The more the depth, the more knowledge we learn about the data. The min_samples_leaf is the minimum number of samples that allow it to be a leaf node hence the max is at 100, it is said that if this is increased, then there is a possibility of underfitting while for min_samples_split, it is the minimum number of samples to split the node and the maximum is kept at 100. As said for the above parameter, the higher the value, the more chances of the model worsening. (Fraj, 2017) By setting these parameters, it reduces the consumption of memory and complexity as well as controls the size of the tree. (SckitLearn, 2020)

```
#by using n_jobs = -1, we telling to use all processor to finish fast
#cv = kfold validation, will do 5 kfold validation, best value for cv is 3/5/10
grid = GridSearchCV(estimator = dc ,param_grid = param_dist,cv = kf, verbose = 2, n_jobs=-1)
```

Figure 98 - grid search CV for tuning

Grid search is a method for tuning hyperparameters that can help you build and test models for any blend of parameter sets per grid. (Saini, 2020) The parameters of the estimator utilized to apply these techniques are tuned by cross-validated grid-search over a parameter grid. For verbosity, the higher it is, it will give us more messages. At 2 it will display the computation for each fold.


```
1 #fit of model
2 grid.fit(X_train_oversample,y_train_oversample)
```

```
Fitting 5 folds for each of 784 candidates, totalling 3920 fits
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 2 concurrent workers.
[Parallel(n_jobs=-1)]: Done 128 tasks      | elapsed:    2.3s
[Parallel(n_jobs=-1)]: Done 612 tasks     | elapsed:   15.5s
[Parallel(n_jobs=-1)]: Done 1424 tasks    | elapsed:   59.3s
[Parallel(n_jobs=-1)]: Done 2556 tasks    | elapsed:   2.0min
[Parallel(n_jobs=-1)]: Done 3920 out of 3920 | elapsed:   4.1min finished
GridSearchCV(cv=KFold(n_splits=5, random_state=47, shuffle=False),
             error_score=nan,
             estimator=DecisionTreeClassifier(ccp_alpha=0.0, class_weight=None,
                                              criterion='gini', max_depth=None,
                                              max_features=None,
                                              max_leaf_nodes=None,
                                              min_impurity_decrease=0.0,
                                              min_impurity_split=None,
                                              min_samples_leaf=1,
                                              min_samples_split=2,
                                              min_weight_fraction_leaf=0.0,
                                              presort='deprecated',
                                              random_state=None,
                                              splitter='best'),
             iid='deprecated', n_jobs=-1,
             param_grid={'criterion': ['gini', 'entropy'],
                        'max_depth': [1, 2, 3, 4, 5, 6, 7, 8],
                        'min_samples_leaf': [2, 5, 10, 25, 50, 75, 100],
                        'min_samples_split': [2, 5, 10, 25, 50, 75, 100]},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
             scoring=None, verbose=2)
```

Figure 99 - fitting model to get best parameters

Result of grid search with the ranking of best parameter for modelling																	
score_df = pd.DataFrame(grid.cv_results_)																	
score_df																	
mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_criterion	param_max_depth	param_min_samples_leaf	param_min_samples_split	params									
0	0.005885	0.002874	0.001520	0.000104	gini	1	2	2	{criterion: 'gini', max_depth: 1, min_sam...	0.000487	0.019473	0.012414	0.000010	0.000000	0.407871	0.248525	0.007
1	0.047382	0.010113	0.001725	0.000181	gini	1	2	5	{criterion: 'gini', max_depth: 1, min_sam...	0.000487	0.019473	0.012414	0.000010	0.000000	0.407871	0.248525	0.007
2	0.002031	0.000284	0.001802	0.000082	gini	1	2	10	{criterion: 'gini', max_depth: 1, min_sam...	0.000487	0.019473	0.012414	0.000010	0.000000	0.407871	0.248525	0.007
3	0.001887	0.000211	0.001821	0.000124	gini	1	2	25	{criterion: 'gini', max_depth: 1, min_sam...	0.000487	0.019473	0.012414	0.000010	0.000000	0.407871	0.248525	0.007
4	0.001173	0.000195	0.001711	0.000084	gini	1	2	50	{criterion: 'gini', max_depth: 1, min_sam...	0.000487	0.019473	0.012414	0.000010	0.000000	0.407871	0.248525	0.007
...
779	0.211782	0.018377	0.001868	0.000085	entropy	8	100	10	{criterion: 'entropy', max_depth: 8, min...	0.772105	0.759001	0.788148	0.788382	0.000074	0.752282	0.038102	155
780	0.133887	0.019455	0.001745	0.000040	entropy	8	100	25	{criterion: 'entropy', max_depth: 8, min...	0.772105	0.759001	0.788148	0.788382	0.000074	0.752282	0.038102	155
781	0.123873	0.018889	0.001738	0.000083	entropy	8	100	50	{criterion: 'entropy', max_depth: 8, min...	0.772105	0.759001	0.788148	0.788382	0.000074	0.752282	0.038102	155
782	0.118584	0.018008	0.001721	0.000048	entropy	8	100	75	{criterion: 'entropy', max_depth: 8, min...	0.772105	0.759001	0.788148	0.788382	0.000074	0.752282	0.038102	155
783	0.201682	0.037855	0.001835	0.000231	entropy	8	100	100	{criterion: 'entropy', max_depth: 8, min...	0.772105	0.759001	0.788148	0.788382	0.000074	0.752282	0.038102	155

Figure 100 - displaying all the possible results

len = score_df.sort_values("rank_test_score")																
len.head(5)																
mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_criterion	param_max_depth	param_min_samples_leaf	param_min_samples_split	params								
361	0.155513	0.019028	0.001729	0.000088	gini	8	10	50	{criterion: 'gini', max_depth: 8, min_sam...	0.818386	0.800595	0.795056	0.818555	0.754627	0.780264	0.029140
359	0.161005	0.006762	0.002063	0.000159	gini	8	10	10	{criterion: 'gini', max_depth: 8, min_sam...	0.817974	0.800010	0.789151	0.814915	0.754610	0.780274	0.029035
358	0.158255	0.011985	0.002020	0.000182	gini	8	10	5	{criterion: 'gini', max_depth: 8, min_sam...	0.817974	0.800010	0.789151	0.814915	0.754610	0.780274	0.029035
363	0.154013	0.011241	0.001717	0.000082	gini	8	10	100	{criterion: 'gini', max_depth: 8, min_sam...	0.818055	0.807794	0.771019	0.814200	0.752751	0.780274	0.029032
357	0.158058	0.007910	0.001764	0.000081	gini	8	10	2	{criterion: 'gini', max_depth: 8, min_sam...	0.817550	0.800010	0.789151	0.814915	0.754610	0.780269	0.029068

Figure 101 - displaying top 5 estimates

```
1 grid.best_params_ #best parameters for the model

{'criterion': 'gini',
 'max_depth': 8,
 'min_samples_leaf': 10,
 'min_samples_split': 50}
```

Figure 102 - best parameters selected for decision tree

The best parameters selected for the decision tree classifier can be seen enough and now will be used to tune the model and evaluate the results.

Model Tuning & Evaluation

For model tuning, we can simply apply the grid parameters straight to the evaluation metrics however we have applied the parameters and ran the decision model again for easier evaluation.

```
1 new_dc = DecisionTreeClassifier(criterion = 'gini',max_depth=8,min_samples_leaf=10,min_samples_split=50)
2 new_dc.fit(X_train_oversample,y_train_oversample)

DecisionTreeClassifier(ccp_alpha=0.0, class_weight=None, criterion='gini',
                      max_depth=8, max_features=None, max_leaf_nodes=None,
                      min_impurity_decrease=0.0, min_impurity_split=None,
                      min_samples_leaf=10, min_samples_split=50,
                      min_weight_fraction_leaf=0.0, presort='deprecated',
                      random_state=None, splitter='best')

1 new_dc_pred = new_dc.predict(X_test)
```

Figure 103 - tuned decision tree model

```
1 #rechecking accuracy scores
2 print(f'Train Accuracy: {new_dc.score(X_train_oversample,y_train_oversample):.6f}')
3 print(f'Test Accuracy: {new_dc.score(X_test,y_test):.6f}')#test value after tuning

Train Accuracy: 0.835174
Test Accuracy: 0.851956
```

Figure 104 - improved train and test data accuracy

The decision tree model has been evaluated with various metrics to test it in many ways and allow the readers to see the improvements or declines in the models. The metrics chosen are the classification table, confusion matrix, sensitivity and specificity, misclassification rate and AUC-ROC chart.

Classification Chart

```
[97] 1 #confusion Matrix calcualtor
      2 cfm =confusion_matrix(y_test,new_dc_pred)
      3 TP = cfm[0,0] # true positive
      4 TN = cfm[1,1] # true negatives
      5 FP = cfm[0,1] # false positives
      6 FN = cfm[1,0] # false negatives
      7 Total = TP+TN+FP+FN
      8
      9 #all metrics after tuning and oversampling
     10
     11 Specificity = (TN)/(TN+FP)
     12 Sensitivity = (TP)/(TP+FN)
     13 Misclassification = 1-(accuracy_score(y_test,new_dc_pred))
     14 print('Specificity          : ',Specificity)
     15 print('Sensitivity           : ',Sensitivity)
     16 print('Misclassification Rate:',Misclassification)
     17 print('Model Results After Tuning')
     18 print('Accuracy              : ', accuracy_score(y_test,new_dc_pred))
     19 print('Precision              : ', precision_score(y_test,new_dc_pred))
     20 print('Recall                 : ', recall_score(y_test,new_dc_pred))
     21 print('F1-Score               : ', f1_score(y_test,new_dc_pred))
     22 print(classification_report(y_test,new_dc_pred))
```

```
Specificity          : 0.7285129604365621
Sensitivity           : 0.8820012586532411
Misclassification Rate: 0.14676553311173612
Model Results After Tuning
Accuracy              : 0.8532344668882639
Precision              : 0.7285129604365621
Recall                 : 0.5874587458745875
F1-Score              : 0.6504263093788064
```

	precision	recall	f1-score	support
0	0.88	0.93	0.91	3002
1	0.73	0.59	0.65	909
accuracy			0.85	3911
macro avg	0.81	0.76	0.78	3911
weighted avg	0.85	0.85	0.85	3911

Figure 105 - classification table

The results of the classification table show improvement in the accuracy at 85%, precision at 73% and f1-score at 65%. The recall value seems to have dropped a bit at 59%. Many parameters were tried and tested however the recall value remained the same throughout and only the computation time increased hence the parameters selected above were the ones chosen to carry out grid search.

The values for Misclassification dropped to 14% which means the model has become more accurate while Specificity and Sensitivity increased to 72% and 88% respectively. The increase is seen as a good thing as higher values for both determine the correct classification of false values and correct classification of true values, respectively.

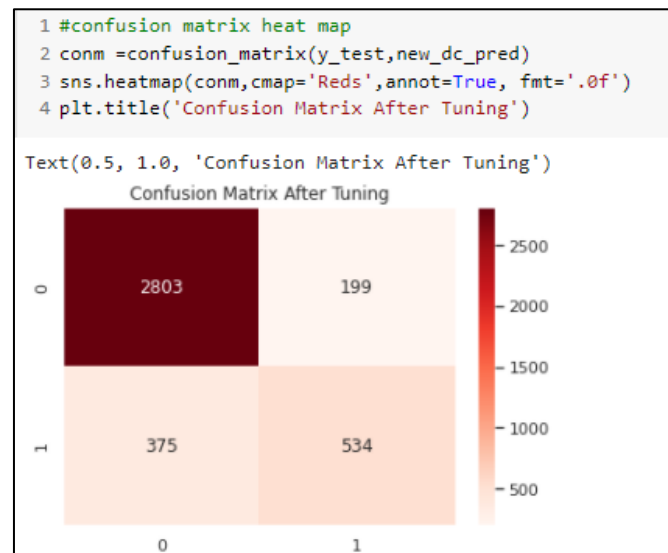


Figure 106 - confusion matrix after tuning

The AUC-Roc curve helps display how good the model is performing. It is usually used to interpret binary classification problems such as ours. ROC plots the TPR vs the FPR. TPR is a true positive rate or sensitivity while FPR is a false positive rate and is 1-specificity. AUC determines how well a classifier can distinguish between inputs and the higher it is, the better the model is performing. (Bhandari, 2020)

```
1 #prediction probabilities
2 x_probs = [0 for _ in range(len(y_test))]
3 new_dc_probs = new_dc.predict_proba(X_test)
4 dc_probs = dc.predict_proba(X_test)
```

Figure 107 - prediction probabilities

X_probs is the baseline and will consist of the 0 probabilities.

New_dc_probs is the probability found by the tuned decision tree.

```
1 new_dc_probs = new_dc_probs[:,1] #Keeping positive prob
2 dc_probs = dc_probs[:,1]
```

Here the positive outcome is kept.

```

1 #computing auROC and roc curve value
2 x_auc = roc_auc_score(y_test,x_probs)
3 new_dc_auc = roc_auc_score(y_test,new_dc_probs)
4 dc_auc = roc_auc_score(y_test,dc_probs)

```

Figure 108 – AUC scores

This computes the area under the ROC curve.

```

1 #AUC-ROC value
2 print ('Random prediction AUROC = %.3f' %(x_auc))
3 print ('Decision Tree prediction AUROC = %.3f'%(new_dc_auc))
4 print ('Decision Tree prediction AUROC - no tuning = %.3f'%(dc_auc))

Random prediction AUROC = 0.500
Decision Tree prediction AUROC = 0.845
Decision Tree prediction AUROC - no tuning = 0.739

```

Figure 109 & 110 – AUC-ROC Values

Random prediction informs us of the “wrong” predictions. When we have assigned the probability to be 0 and AUROC will equal to 0.5. The decision tree which was tuned is 0.845 while the decision tree without any tuning is at 0.739.

```

1 #roc curve values
2 x_fpr,x_tpr,_ = roc_curve (y_test,x_probs)
3 new_dc_fpr,new_dc_tpr,_ = roc_curve(y_test,new_dc_probs)
4 dc_fpr,dc_tpr,_ = roc_curve(y_test,dc_probs)

```

Figure 111 - computing values to model the curve

The FPR and TPR are used to plot the ROC Curve.

```

1 #chart for roc curve
2 import matplotlib.pyplot as plt
3
4 #plotting roc curve
5 plt.plot(x_fpr,x_tpr, linestyle = ':', label = 'Random Prediction (AUROC = %0.3f)' %x_auc)
6 plt.plot(new_dc_fpr,new_dc_tpr, linestyle = 'solid', label = 'Decision Tree Prediction (AUROC = %0.3f)' %new_dc_auc)
7 plt.plot(dc_fpr,dc_tpr, linestyle = 'solid', label = 'DT prediction - no tuning (AUROC = %0.3f)' %dc_auc)
8
9 #title
10 plt.title ('ROC PLOT')
11 #axis labels
12 plt.xlabel('FALSE POSITIVE RATE')
13 plt.ylabel('TRUE POSITIVE RATE')
14 #show legend
15 plt.legend()
16 #show plot
17 plt.show()
18
19
20 #compares performamnce of different learning models however it can be used to

```

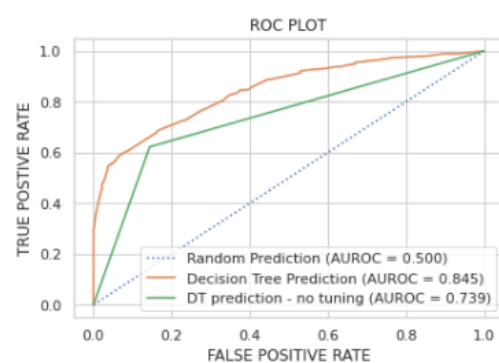


Figure 112 - ROC Plot showing AUROC values

This chart is usually used to compare the performances of different models. So, for this, we have compared the untuned and tuned model and can see that the tuned model represented by the orange line is performing much better. (Data Professor, 2020)

```

1 print('Decision Tree (Tuned) %.3f' % new_dc_auc)
Decision Tree (Tuned) 0.845

```

Figure 113 - AUC value of tuned decision tree model

	Before Tuning and Over Sampling	Before Tuning and After Over Sampling	After Tuning and Over Sampling
Precision	0.60	0.57	0.73
Accuracy	0.81	0.80	0.85
Recall	0.61	0.62	0.59
F1-Score	0.60	0.59	0.65
Misclassification Rate	-	0.20	0.14

Sensitivity	-	0.88	0.88
Specificity	0.58	0.57	0.73
TN	2628	2570	2803
TP	555	565	534
FP	374	432	199
FN	354	344	375
AUC-ROC	-	0.74	0.85

Table 2 - The table shows comparison of all the metrics used in a cohesive format.

2.2.4 Support Vector Machine

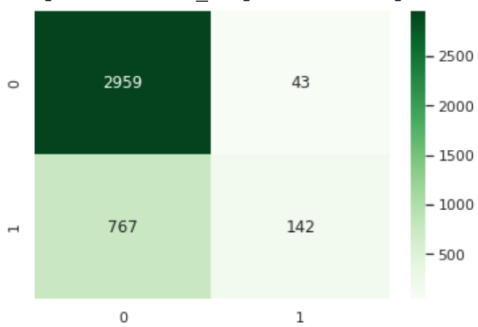
SVM Meta Parameter Selection

```
# initially apply support vector classifier with rbf kernel
from sklearn.svm import SVC
svc=SVC(kernel='rbf')
svc.fit(X_train,y_train)
yhat=svc.predict(X_test)

# initial SVM accuracy score
from sklearn.metrics import accuracy_score,confusion_matrix,classification_report,f1_score
accuracySVC = accuracy_score(y_test,yhat)
accuracySVC

0.7928918435182818

# confusion matrix heat map
sns.heatmap(confusion_matrix(y_test,yhat),cmap='Greens',annot=True, fmt='.0f')

<matplotlib.axes._subplots.AxesSubplot at 0x7fbb1195ef50>


```
classification report
print(classification_report(y_test,yhat))
```



|              | precision | recall | f1-score | support |
|--------------|-----------|--------|----------|---------|
| 0            | 0.79      | 0.99   | 0.88     | 3002    |
| 1            | 0.77      | 0.16   | 0.26     | 909     |
| accuracy     |           |        | 0.79     | 3911    |
| macro avg    | 0.78      | 0.57   | 0.57     | 3911    |
| weighted avg | 0.79      | 0.79   | 0.74     | 3911    |


```

Figure 114 - SVM Model

The figure above shown the initial information on the Support Vector Machine classifier applied before the model tuning. As can be seen, the model accuracy score is around 79.3% with the classification report and confusion matrix displayed above.

According to the research by (Soares and Brazdil, 2006), the choices of parameters in Support Vector Machine (SVM) classifiers are highly correlated and sensitive with the algorithm. There is evidence that parameter selection is used to support machine learning. The meta-learning approach of selecting a set of parameters to be used can allow users to face the least expected errors and getting optimal results (Soares and Brazdil, 2006).

To get the best accuracy results, Support Vector Machine (SVM) has several types of parameters to be tuned some of the commonly used includes specific kernels that can be used in the algorithm, regularisation C parameters, and Gamma coefficient. The implementation of the model tuning can be done by using Python language code and commonly use Scikit-learn class.

Types of kernels are used to find the lower dimensional input space and transform the data into higher dimensional space. In SVM, the kernel that is mainly used is rbf which stands for Radial Basis Function using Gaussian model. Other kernels commonly used in the algorithm can be linear and polynomial kernels. Kernels are mostly used in solving non-linear separation problems. In this project, those kernels will be compared and selected based on their results that would be discussed later.

```
# SVM kernels
kernels = ['Polynomial', 'RBF', 'Sigmoid', 'Linear']
def getClassfier(ktype):
    if ktype == 0:
        # Polynomial kernel
        return SVC(kernel='poly', degree=8, gamma="auto")
    elif ktype == 1:
        # Radial Basis Function kernel
        return SVC(kernel='rbf', gamma="auto")
    elif ktype == 2:
        # Sigmoid kernel
        return SVC(kernel='sigmoid', gamma="auto")
    elif ktype == 3:
        # Linear kernel
        return SVC(kernel='linear', gamma="auto")
```

Figure 115 - SVM Kernels

The parameters that are going to be used and compared during model tuning are shown above, the kernels used are polynomial, RBF, sigmoid, and linear kernel. The machine will select the best parameters to be applied for the dataset in the model hyperparameter tuning chapter later.

The SVM regularisation parameter used is C which acted as a penalty parameter and is used to report to the machine if the SVM model optimisation error can be bearable. That is why C regularisation represented the error term or misclassification so that the machine and user can control the boundaries between decision and misclassification trade-off. However, the regularisation parameters might affect the data to be overfitted. In the project, the C regularisation parameters also will be tested by using different inputs of C to check the best accuracy score to be able to train the best models with suitable inputs.

While Gamma used to be the coefficient for kernels and can be used to define the distance between data points influenced by the plausible line of separation. The higher Gamma coefficient will also find near the decision boundary to be considered. In this project, the Gamma coefficient numbers would be automatically generated by the machine without manually inputting the coefficient range. Therefore, the $1/n_features$ are going to be used for the coefficient.

SVM Model Tuning

The parameters can be tuned by using a different method such as Grid Search or Random Search. Both parameter settings will help to build and evaluate the model based on the different inputs selected for each combination of the algorithm. The most common hyperparameter tuning method is using Grid Search method which will also be used in this project's Support Vector Machine (SVM) tuning. Grid Search combined the estimated results based on the model evaluation with grid search preamble tuning that is under the scikit-learn library in Python.

Model hyperparameters are the parameters that the models are unable to estimate the parameters. Hyperparameter tuning is used to determine the right combination of hyperparameters that are used to evaluate and enhance model performance to get maximum results. Finding the correct hyperparameter may result in the maximum performance of the models constructed.

```

for i in range(4):
    # Train a SVC model using different kernel
    svcclassifier = getClassifier(i)
    # Making prediction
    svcclassifier.fit(X_train, y_train)
    y_pred = svcclassifier.predict(X_test)
    # Print Model Evaluation Classification Report
    print("Evaluation:", kernels[i], "kernel")
    print(classification_report(y_test, y_pred))

```

Evaluation: Polynomial kernel					
	precision	recall	f1-score	support	
0	0.78	0.97	0.86	3002	
1	0.44	0.08	0.14	909	
accuracy			0.76	3911	
macro avg	0.61	0.53	0.50	3911	
weighted avg	0.70	0.76	0.69	3911	

Evaluation: RBF kernel					
	precision	recall	f1-score	support	
0	0.79	0.99	0.88	3002	
1	0.77	0.16	0.26	909	
accuracy			0.79	3911	
macro avg	0.78	0.57	0.57	3911	
weighted avg	0.79	0.79	0.74	3911	

Evaluation: Sigmoid kernel					
	precision	recall	f1-score	support	
0	0.78	0.80	0.79	3002	
1	0.29	0.27	0.28	909	
accuracy			0.68	3911	
macro avg	0.54	0.53	0.54	3911	
weighted avg	0.67	0.68	0.67	3911	

Evaluation: Linear kernel					
	precision	recall	f1-score	support	
0	0.77	1.00	0.87	3002	
1	0.00	0.00	0.00	909	
accuracy			0.77	3911	
macro avg	0.38	0.50	0.43	3911	
weighted avg	0.59	0.77	0.67	3911	

Figure 116 - Model Tuning

Based on the classification report above, we can see how different kernels affect the accuracy score as well as other report scores. We can see the best kernel are RBF (Radial Based Function) with an accuracy rate of 79%, while Polynomial kernel has 76% accuracy score, Linear kernel with 77% accuracy score, and Sigmoid kernel come in the last at 68%.

```

# import grid search to choose meta parameter
from sklearn.model_selection import GridSearchCV

# grid search parameters
param_grid = {'C': [0.1, 1, 10, 100], 'gamma': ['auto'], 'kernel': ['rbf', 'poly', 'sigmoid']}

# hyperparameter tuning with Grid Search
grid = GridSearchCV(SVC(), param_grid, refit=True, verbose=2, n_jobs=4)
grid.fit(X_train, y_train)

Fitting 5 folds for each of 12 candidates, totalling 60 fits
[Parallel(n_jobs=4)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=4)]: Done 33 tasks | elapsed: 3.4min
[Parallel(n_jobs=4)]: Done 60 out of 60 | elapsed: 16.6min finished
GridSearchCV(cv=None, error_score=nan,
              estimator=SVC(C=1.0, break_ties=False, cache_size=200,
                             class_weight=None, coef0=0.0,
                             decision_function_shape='ovr', degree=3,
                             gamma='scale', kernel='rbf', max_iter=-1,
                             probability=False, random_state=None, shrinking=True,
                             tol=0.001, verbose=False),
              iid='deprecated', n_jobs=4,
              param_grid={'C': [0.1, 1, 10, 100], 'gamma': ['auto'],
                           'kernel': ['rbf', 'poly', 'sigmoid']},
              pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
              scoring=None, verbose=2)

# best class and parameters of the model
print(grid.best_estimator_)

SVC(C=10, break_ties=False, cache_size=200, class_weight=None, coef0=0.0,
    decision_function_shape='ovr', degree=3, gamma='auto', kernel='rbf',
    max_iter=-1, probability=False, random_state=None, shrinking=True,
    tol=0.001, verbose=False)

```

Figure 117 - Grid Search

The figure above shown the Grid Search hyperparameter tuning are being used to find the best combination of SVM model parameters by firstly input the selection of C regularisation, automatic gamma number, and 3 kernels. Then the GridSearchCV from Sklearn model selection is being called to fit the training data and find the optimal parameters. The best parameter that the grid search found was the RBF kernel with a C of 10.

SVM Model Evaluation

To evaluate the model results, the output to see how the performance of the Support Vector Machine (SVM) classifier algorithm is using metrics such as classification report and confusion matrix, which give the detailed information about the model's performance by its True Positive, True Negative, False Positive, and False Negative value.

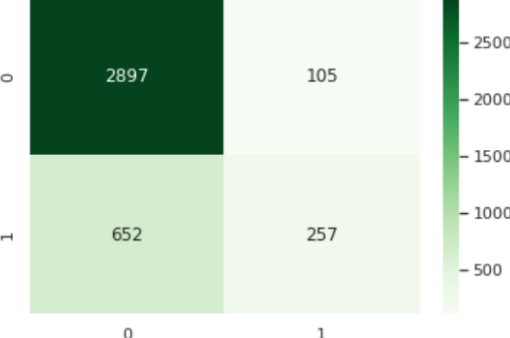
```
# model evaluation classification report
grid_predictions = grid.predict(X_test)
print(confusion_matrix(y_test,grid_predictions))
print(classification_report(y_test,grid_predictions))
```

```
[[2897  105]
 [ 652  257]]
```

	precision	recall	f1-score	support
0	0.82	0.97	0.88	3002
1	0.71	0.28	0.40	909
accuracy			0.81	3911
macro avg	0.76	0.62	0.64	3911
weighted avg	0.79	0.81	0.77	3911

```
# confusion matrix heat map
sns.heatmap(confusion_matrix(y_test,grid_predictions),cmap='Greens',annot=True, fmt='.0f')
```

<matplotlib.axes._subplots.AxesSubplot at 0x7febe5135290>



```
# model accuracy score
accuracySVC2 = accuracy_score(y_test,grid_predictions)
accuracySVC2
```

```
0.8064433648683201
```

Figure 118 - Evaluation

The model evaluation is based on the classification report and confusion matrix, by calling the test dataset to get the accuracy rate and matrices. The best accuracy that could be found with the GridSearch algorithm was around 80.6% which is higher than the initial accuracy before the hyperparameter tuning with the same kernel at 79.3%. The matrix score before and after the tuning can be seen and compared from the table below.

Confusion Matrix	Value Before Tuning	Value After Tuning
True Positive	2959	2897
True Negative	142	257
False Positive	43	105
False Negative	767	652

```
# confusion matrix calculator
cm = confusion_matrix (y_test,grid_predictions)

TN=cm[1][1]
TP=cm[0][0]
FN=cm[1][0]
FP=cm[0][1]

# rates and scores evaluation
print ('Accuracy Score = ', (TP+TN)/(TP+TN+FP+FN))
print ('Misclassification Score = ', 1-(TP+TN)/(TP+TN+FP+FN))
print ('Sensitivity Rate = ', TP/(TP+FN))
print ('Specificity Rate = ', TN/(TN+FP))
print ('Positive Predicted Value = ', TP/(TP+FP))
print ('Negative Predicted Value = ', TN/(TN+FN))
print ('Positive Likelihood Ratio = ', ((TP/(TP+FN))/(1-(TN/(TN+FP))))))
print ('Negative Likelihood Ratio = ', ((1-(TP/(TP+FN)))/(TN/(TN+FP))))

Accuracy Score = 0.8064433648683201
Misclassification Score = 0.19355663513167987
Sensitivity Rate = 0.8162862778247394
Specificity Rate = 0.7099447513812155
Positive Predicted Value = 0.9650233177881412
Negative Predicted Value = 0.28272827282728275
Positive Likelihood Ratio = 2.8142441197386256
Negative Likelihood Ratio = 0.25877185769433597
```

Figure 119 - Evaluation Metrics

To calculate the evaluation scores and rates, the classifier scores are being displayed in the figure above. The accuracy score of 80.6%, misclassification score represented the error rate which is the remaining score from the accuracy of around 0.193.

Sensitivity rates are true positive rates to measure the proportion of correctly identified as having a positives rate. On the other hand, specificity rate measures the correctly identified negative rate of not having the specific condition. They are also being used to find the positive

and negative likelihood ratios. Based on the calculation in the above figure, the sensitivity rate is more than 81.6% with less than 71% specificity rate.

Positive and Negative predicted values represented the probability of the positive or negative subject being classified as positive is truly positive in reality and reverse. In the project, the positive predicted value is shown as 96.5% which is good and represented the accuracy of the positive being positive while the negative predicted value stands at around 28.3% showing the possibility of getting a false negative is low.

The positive likelihood ratio (+LR) represented the change of odds of someone having a condition with a positive result. The SVM positive likelihood ratio of more than 2.8 indicated that there is a 2-to-3-fold increase of someone having a condition with a positive result. Therefore, the higher the +LR indicated there are more informative to the test. While negative likelihood ratio (-LR) represented the odds of someone with the negative result getting the condition and the smaller number indicated more informative test received. The odd of -LR is around 0.26 which is less than 1-fold indicated quite a good result

Part 2: Group Component

2.3 Critical Analysis of Models

The hyperparameter tuning can be referred to as the machine algorithm to find the best set of optimal hyperparameters to construct the logic with methods that have been input before the process is started. Two tuning strategies that can be applied to find hyperparameter optimisation is by using Grid Search and Random Search. It is an important step to build a model and is the way to improve the model performance (Boyle, 2019).

The models' hyperparameter tuning results are being compared in the accuracy below. The model before and after tuning that has been done based on the model in the previous chapter has been concluded in the table below.

		i

The essential classification and evaluation metric that is very commonly used is the accuracy score. It is based on the proportion of the total true results being received out of the total number of cases. The accuracy score can be counted by the total number of True Positive added with True Negative and divided by the total amount of true and false positive and negative. Therefore, the higher the accuracy rate represented a suitable model to be used for the dataset (Agarwal,2019).

According to the model processing and construction, Random Forest Model had an accuracy of 85.3% before tuning however the accuracy after tuning was slightly lower at 82.6%. However, the train accuracy went from 98.2% to 83.1% after tuning which means that there was no overfitting. Compared to other models Random Forest's performance after tuning was still of high accuracy but not the best after tuning. Therefore, the random forest model was the model with the highest accuracy by default before tuning as compared to other models.

Logistic regression was one of the lowest accuracy scores which before hyperparameter tuning has 56.2% accuracy while after the model tuning, logistic regression has very high improvement as compared to any other model. The logistic regression model tuning improves the accuracy score by almost 20%, making the accuracy score after hyperparameter tuning to be 73.8%.

Compared to logistic regression, the support vector machine accuracy score improves slightly after the model tuning. Earlier, the SVM accuracy score shown the accuracy score to be 79.3%. however, after the implementation of the hyperparameter tuning, it has seen the improvement of accuracy score to 80.6% which is slightly improving compared to the accuracy score previously.

Lastly, the decision tree classifiers optimisation finders have also improved the initial highest accuracy score among other models. The accuracy score from the decision tree before hyperparameter tuning was 80.1%. With the hyperparameter tuning implemented, the accuracy of the model improved more than 5 per cent to 85.3% making it among the highest and can be considered as the most suitable model to be used with the dataset due to its high accuracy.

For Logistic regression and decision tree models, the data needed to be oversampled as the logistic regression was not predicting values for (1) Arrested while the decision trees were not producing accurate results when the tree was being visualized. Therefore, both models' data was resampled and model tuning significantly increased the accuracy of both models. Logistic

regression had the most increase from 56.2% to 73.8% however when compared to all four models, the decision tree was the best model with an accuracy of 80.1% to 85.3%.

2.4 Conclusions

In conclusion, four models were built on the Chicago crime dataset that was selected. The 4 models were Random Forest Classification model, Logistic Regression, Support Vector Machine and Decision Tree model. All of the models were tuned using GridSearchCV. Comparison and evaluation of the models were done using the common accuracy metric between all the models to find which model is the most suitable for the prediction and analysis.

Based on the accuracy score after the optimisation from the hyperparameter tuning results, it is found that the decision tree was the best model for predicting whether criminal arrests were made or not as compared to the three other models. The decision tree is a good model to select for criminal arrest predictions as it is easier to be interpreted. While other models might also be used but might give lower accuracy results compared to the decision tree.

Future Improvements related to this dataset may include making use of a larger data set so there is no issue of data imbalances which caused an issue for the Random Forest and Logistic Regression models and it would also help to improve the consistency of the dataset that might improve the accuracy results. More models can also be applied for the ability to compare and find the suitable model for the dataset.

2.5 References

1. Aditya .P (2018). *L1 and L2 Regularization*. - Aditya .P - Medium. [online] Medium. Available at: <https://medium.com/@aditya97p/l1-and-l2-regularization-237438a9caa6> [Accessed 20 Jul. 2021].
2. Agarwal, R., 2019. *The 5 Classification Evaluation metrics every Data Scientist must know*. [online] Medium. Available at: <https://towardsdatascience.com/the-5-classification-evaluation-metrics-you-must-know-aa97784ff226> [Accessed 9 July 2021].
3. Bhandari, A. (2020). *AUC-ROC Curve in Machine Learning Clearly Explained - Analytics Vidhya*. [online] Analytics Vidhya. Available at: <https://www.analyticsvidhya.com/blog/2020/06/auc-roc-curve-machine-learning/> [Accessed 21 Jul. 2021]
4. Brownlee, J. (2020). *SMOTE for Imbalanced Classification with Python*. [online] Machine Learning Mastery. Available at: <https://machinelearningmastery.com/smote-oversampling-for-imbalanced-classification/> [Accessed 20 Jul. 2021].
5. Brownlee, J. (2020). *Cost-Sensitive Logistic Regression for Imbalanced Classification*. [online] Machine Learning Mastery. Available at: <https://machinelearningmastery.com/cost-sensitive-logistic-regression/> [Accessed 20 Jul. 2021].
6. Boyle, T., 2019. *Hyperparameter Tuning*. [online] Medium. Available at: <https://towardsdatascience.com/hyperparameter-tuning-c5619e7e6624> [Accessed 14 July 2021].
7. Data Professor (2020). *How to Plot an ROC Curve in Python | Machine Learning in Python. YouTube*. Available at: https://www.youtube.com/watch?v=uVJXPPrWRJ0&ab_channel=DataProfessor [Accessed 21 Jul. 2021].
8. Fraj, M.B. (2017). *InDepth: Parameter tuning for Decision Tree - Mohtadi Ben Fraj - Medium*. [online] Medium. Available at: <https://medium.com/@mohtedibf/indepth-parameter-tuning-for-decision-tree-6753118a03c3> [Accessed 21 Jul. 2021].
9. Gandhi, R., 2018. *Support Vector Machine — Introduction to Machine Learning Algorithms*. [online] Medium. Available at: <https://towardsdatascience.com/support->

vector-machine-introduction-to-machine-learning-algorithms-934a444fca47>

[Accessed 10 July 2021].

10. Grace-Martin, K. (2015). *What is a Logit Function and Why Use Logistic Regression? - The Analysis Factor*. [online] The Analysis Factor. Available at: <https://www.theanalysisfactor.com/what-is-logit-function/> [Accessed 20 Jul. 2021].
11. Great Learning Team (2020). *Decision Tree Algorithm Explained with Examples*. [online] GreatLearning Blog: Free Resources what Matters to shape your Career! Available at: <https://www.mygreatlearning.com/blog/decision-tree-algorithm/> [Accessed 9 Jul. 2021].
12. Gupta, P. (2017). *Decision Trees in Machine Learning - Towards Data Science*. [online] Medium. Available at: <https://towardsdatascience.com/decision-trees-in-machine-learning-641b9c4e8052> [Accessed 9 Jul. 2021].
13. Hale, J. (2019). *Don't Sweat the Solver Stuff - Towards Data Science*. [online] Medium. Available at: <https://towardsdatascience.com/dont-sweat-the-solver-stuff-aea7cddc3451> [Accessed 21 Jul. 2021].
14. Jain, R., 2017. *Simple Tutorial on SVM and Parameter Tuning in Python and R / HackerEarth Blog*. [online] HackerEarth Blog. Available at: <https://www.hackerearth.com/blog/developers/simple-tutorial-svm-parameter-tuning-python-r/> [Accessed 1 July 2021].
15. Liu, C., 2020. *SVM Hyperparameter Tuning using GridSearchCV - Velocity Business Solutions Limited*. [online] Velocity Business Solutions Limited. Available at: <https://www.vebuso.com/2020/03/svm-hyperparameter-tuning-using-gridsearchcv/> [Accessed 7 July 2021].
16. Mitrani, A. (2019). *Evaluating Categorical Models II: Sensitivity and Specificity*. [online] Medium. Available at: <https://towardsdatascience.com/evaluating-categorical-models-ii-sensitivity-and-specificity-e181e573cff8#:~:text=Sensitivity%20is%20the%20metric%20that,negatives%20of%20each%20available%20category.> [Accessed 21 Jul. 2021].
17. Navlani, A. (2018). *Random Forests Classifiers in Python*. [online] DataCamp Community. Available at: <https://www.datacamp.com/community/tutorials/random-forests-classifier-python#how> [Accessed 22 Jul. 2021].

18. Patil, P., 2018. *What is Exploratory Data Analysis?*. [online] Medium. Available at: <https://towardsdatascience.com/exploratory-data-analysis-8fc1cb20fd15> [Accessed 8 July 2021].
19. Quantstart (2013). Beginner's Guide to Decision Trees for Supervised Machine Learning | QuantStart. [online] Quantstart.com. Available at: <https://www.quantstart.com/articles/Beginners-Guide-to-Decision-Trees-for-Supervised-Machine-Learning/> [Accessed 9 Jul. 2021].
20. Ray, S., 2017. *Commonly Used Machine Learning Algorithms / Data Science*. [online] Analytics Vidhya. Available at: <https://www.analyticsvidhya.com/blog/2017/09/common-machine-learning-algorithms/> [Accessed 3 July 2021].
21. Saini, B. (2020). Hyperparameter Tuning of Decision Tree Classifier Using GridSearchCV. [online] Medium. Available at: <https://ai.plainenglish.io/hyperparameter-tuning-of-decision-tree-classifier-using-gridsearchcv-2a6ebcaffeda> [Accessed 20 Jul. 2021].
22. Saxena, S. (2020). *Random Forest Hyperparameter Tuning in Python / Machine learning*. [online] Analytics Vidhya. Available at: <https://www.analyticsvidhya.com/blog/2020/03/beginners-guide-random-forest-hyperparameter-tuning/> [Accessed 23 Jul. 2021].
23. SckitLearn (2020). `sklearn.tree.DecisionTreeClassifier` — scikit-learn 0.24.2 documentation. [online] Scikit-learn.org. Available at: <https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html#sklearn.tree.DecisionTreeClassifier> [Accessed 20 Jul. 2021].
24. ScikitLearn (2020). `sklearn.model_selection.KFold` — scikit-learn 0.24.2 documentation. [online] Scikit-learn.org. Available at: https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.KFold.html [Accessed 22 Jul. 2021].
25. Soares, C. and Brazdil, P., 2006. Selecting parameters of SVM using meta-learning and kernel matrix-based meta-features. *Proceedings of the 2006 ACM symposium on Applied computing - SAC '06*, pp.564-568.
26. StackExchange (2018). *What happens when bootstrapping isn't used in sklearn.RandomForestClassifier?* [online] Cross Validated. Available at:

<https://stats.stackexchange.com/questions/354336/what-happens-when-bootstrapping-isnt-used-in-sklearn-randomforestclassifier> [Accessed 23 Jul. 2021].

27. Srivastava, T. (2015). *Random Forest Parameter Tuning / Tuning Random Forest*. [online] Analytics Vidhya. Available at: <https://www.analyticsvidhya.com/blog/2015/06/tuning-random-forest-model/> [Accessed 23 Jul. 2021].
28. Thanda, A. (2020). *What is Logistic Regression? A Beginner's Guide [2021]*. [online] CareerFoundry. Available at: <https://careerfoundry.com/en/blog/data-analytics/what-is-logistic-regression/#what-is-logistic-regression> [Accessed 19 Jul. 2021].
29. Wijaya, C. Y. (2020). *5 SMOTE Techniques for Oversampling your Imbalance Data*. [online] Medium. Available at: <https://towardsdatascience.com/5-smote-techniques-for-oversampling-your-imbalance-data-b8155bdbe2b5> [Accessed 20 Jul. 2021].

