2 Section B - Census Income

2.1 Load and explore the data (note your observations)

2.1.1 Data importation and structure

The census income data set was comprised of 32561 observations and 14 variables. These f orm a mixture of numerical and categorical variables. The dependent variable in this study is 'income' which illustrates if an individual earns an annual income of more or less than US D 50,000. An overview of the head of the data can be found in table 5.

```
import pandas as pd # Import pandas
# Data importation
census_data = pd.read_csv("./CensusDB.csv")

# Head of data
census_data.head()
```

Table 5 - Head of census income dataset

	age	workclass	fnlwgt	education	education-num	marital-status	occupation	relationship	sex	capital-gain	capital-loss	hours-per-week	native-country
0	90	?	77053	HS-grad	9	Widowed	?	Not-in-family	Female	0	4356	40	United-States
1	82	Private	132870	HS-grad	9	Widowed	Exec-managerial	Not-in-family	Female	0	4356	18	United-States
2	66	?	186061	Some-college	10	Widowed	?	Unmarried	Female	0	4356	40	United-States
3	54	Private	140359	7th-8th	4	Divorced	Machine-op-inspct	Unmarried	Female	0	3900	40	United-States
4	41	Private	264663	Some-college	10	Separated	Prof-specialty	Own-child	Female	0	3900	40	United-States

```
# Dimensions of dataset
census_data.shape

(32561, 14)
```

```
census_data.dtypes

age                int64
workclass         object
fnlwgt             int64
education         object
education-num      int64
marital-status    object
occupation        object
relationship      object
sex               object
capital-gain       int64
capital-loss       int64
hours-per-week     int64
native-country    object
income            object
dtype: object
```

2.1.2 Handling missing data and imputation

Missing data values were indicated with '?'. These were replaced with Nan indicating to python that such values were missing elements. Computation of the missing values per variable was calculated, with the 'work class', 'occupation' and 'native-country' variables obtaining missing values percentages of approximately 5.63%, 5.66% and 1.79% respectively. Figure 10 shows an overview of the missing data.

```
# Replace ? with NaN
#! pip install numpy
import numpy as np
census_data.replace('?', np.NaN, inplace=True)

# Percentage of missing datapoints per variable
census_data.isna().mean() * 100

age               0.000000
workclass         5.638647
fnlwgt            0.000000
education         0.000000
education-num     0.000000
marital-status    0.000000
occupation        5.660146
relationship      0.000000
sex               0.000000
capital-gain      0.000000
capital-loss      0.000000
hours-per-week    0.000000
native-country    1.790486
income            0.000000
dtype: float64
```

```
# ! pip install matplotlib
# ! pip install seaborn
import matplotlib.pyplot as plt # matplotlib
import seaborn as sns # seaborn

# Visualisation of missing data points per variable
plt.figure(figsize=(13,8)) # Set figure size
sns.heatmap(census_data.isnull(), # Return a boolean same-sized object indica
ting if the values are NA
```

```
            yticklabels=False,  # yticklabels=False disables y axis values
            cbar=False, cmap="viridis") # cbar = False disables the color bar
and cmap="viridis" specifies the colour viridis
```

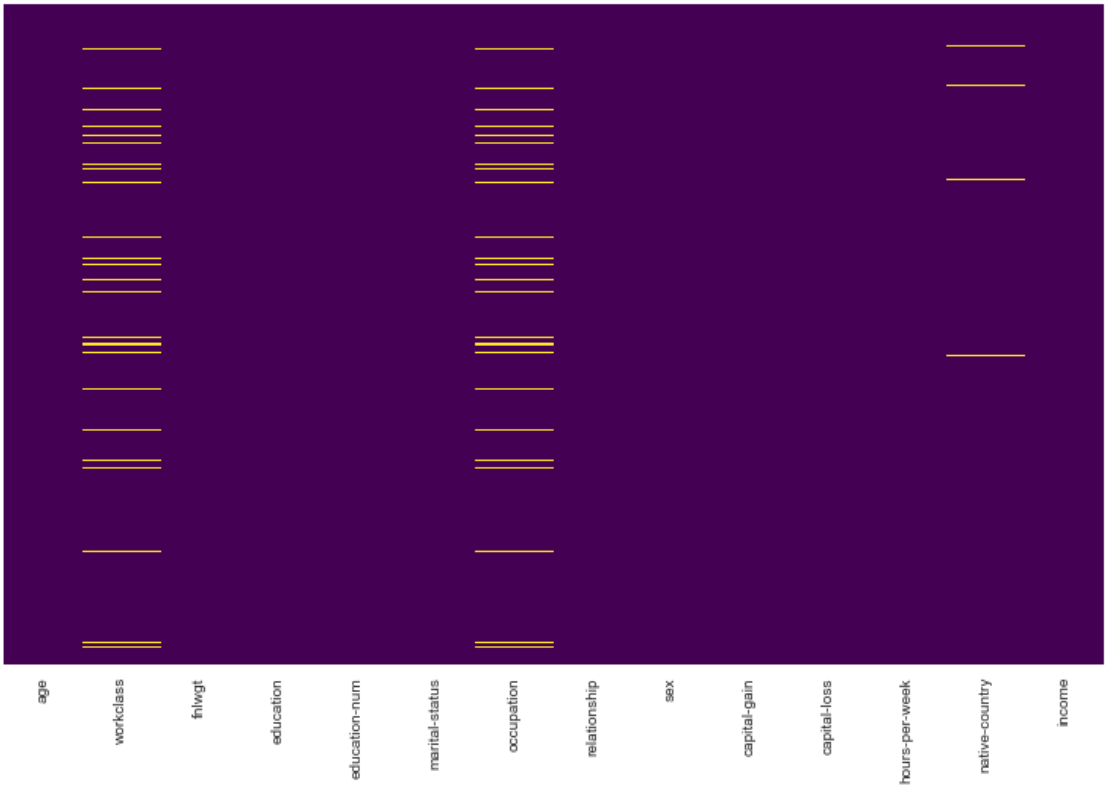

Figure 10-Map of missing data

To replace these missing data values mode imputation was employed. Mode utilized as the variables with missing datapoints were categorical, preventing any form of quantitative imputation such as mean and median.

```
# Find mode of variables with missing data.
census_workclass_mode = census_data["workclass"].mode()[0]


census_occupation_mode = census_data["occupation"].mode()[0]
```

```
census_native_country_mode = census_data["native-country"].mode()[0]

census_data["workclass"].fillna(census_workclass_mode, inplace = True) # fill missing data with mode of workclass variable


census_data["occupation"].fillna(census_occupation_mode, inplace = True) # fill missing data with mode of occupation variable


census_data["native-country"].fillna(census_native_country_mode, inplace = True) # fill missing data with mode with mode of native-country variable
```

After applying the mode imputation all missing data points were removed.

```
# No missing datapoints
census_data.isna().mean() * 100
```

```
age               0.0
workclass         0.0
fnlwgt            0.0
education         0.0
education-num     0.0
marital-status    0.0
occupation        0.0
relationship      0.0
sex               0.0
capital-gain      0.0
capital-loss      0.0
hours-per-week    0.0
native-country    0.0
income            0.0
dtype: float64
```

2.1.3 Exploratory Data Analysis

2.1.3.1 Income distribution from census

From figure 11, most individuals in the dataset earned incomes of less than USD 50,000. About 24000 of the individuals in the dataset earned less than USD 50,000 and only about 7000 individuals earned more than USD 50,000.

```
# countplot for income distribution
sns.countplot(x="income", data = census_data)
```

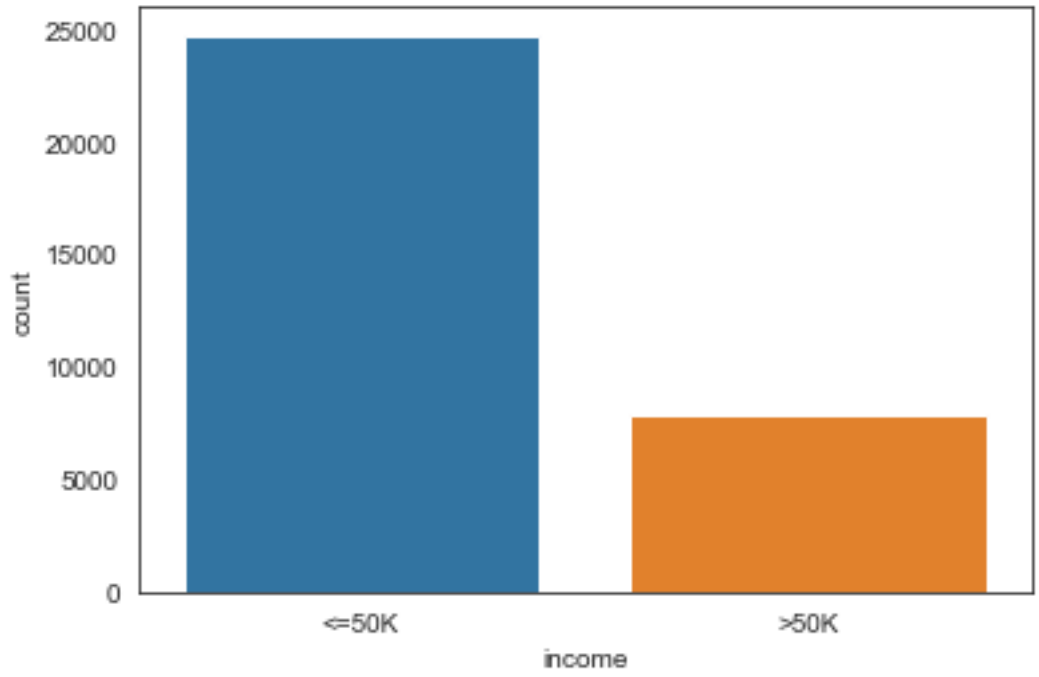

Figure 11-Income distribution from census

2.1.3.2 Age distribution from census

The age variables were segregated into 8 groups. The distribution of age groups in figure 12 shows that the 30 to 40 age group had the highest count of about 8,200. Closely following this group was the 20 to 30 age group with about 8000 individuals. The group with the least count was the 80 to 90 age group with a small count of about 130.

```
# Defining the age bins
bins = [10,20,30,40,50,60,70,80,90] # Define bins
```

```
# Using panadas cut method to create age groups
age_groups = pd.cut(census_data["age"],bins) # Cut with defined bins

age_groups
```

```
0        (80, 90]
1        (80, 90]
2        (60, 70]
3        (50, 60]
4        (40, 50]
           ...
32556    (20, 30]
32557    (20, 30]
32558    (30, 40]
32559    (50, 60]
32560    (20, 30]
Name: age, Length: 32561, dtype: category
```

```
census_data['age_groups'] = age_groups # create variable age_groups

# Countplot with age groups
sns.countplot(x="age_groups", data = census_data)

# Drop age groups variable
census_data.drop('age_groups' , axis =1, inplace = True)
```

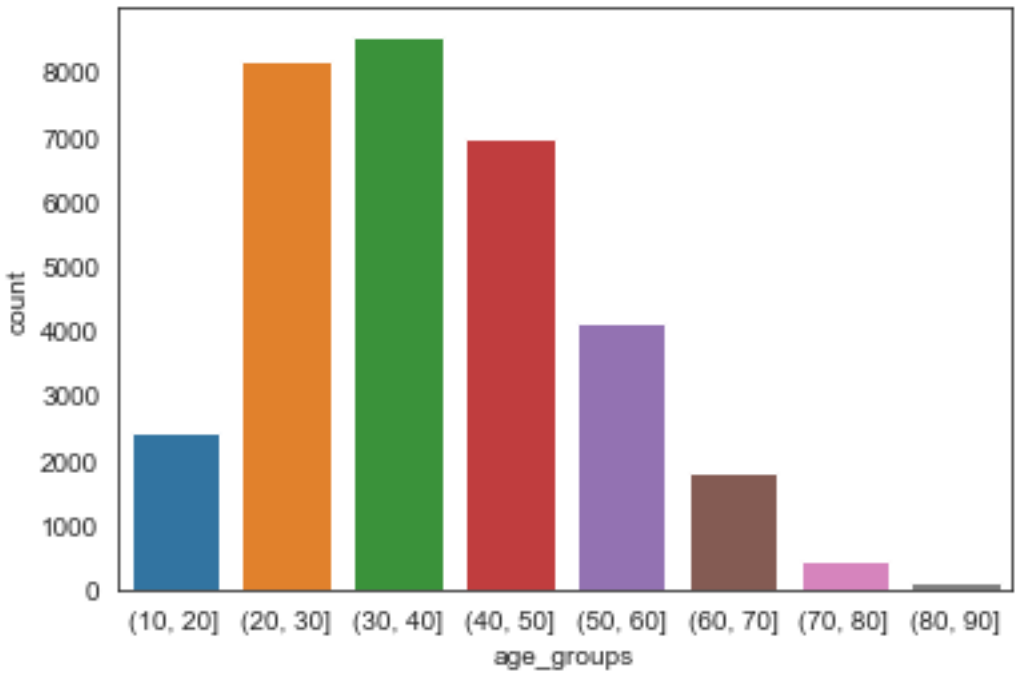

Figure 12 -Age groups in census

2.1.3.3 Income status per education level

Figure 13 illustrates a relationship between education levels and earning potential. 'HS-grades' constituted the bulk of individuals within the dataset, with about 8000 people earning less than USD 50,000 and about 1900 earning more than that income level. A similar excess of lower to higher income earners was witnessed in the 'some-college' and 'Bachelors' group. Individuals with masters on the other hand witnessed the inverse of this with about 1500 people earning more than USD 50,000 and 1200 earning less than this value. A similar relation was witnessed in individuals with doctorates. These income variations may result from the specialized skills and knowledge higher education holder have, earning them a place in the higher income bracket.

```
sns.set_style("white") # Set style to white

# Set education to the y-axis and hue to the income variable
# Kind of plot is count and pallete defines the colour
# Edgecolor defines the boldness of the edges of the catplot.
# Lower values give bolder colours
```

```
cat_plot = sns.catplot(y="education", hue="income",
                        kind="count", palette="pastel",
edgecolor=".5",data=census_data)


plt.title("Income status per education level", fontsize = 20) # Define title
and fontsize


plt.xlabel('Count') # Label x axis


plt.ylabel('education level') # Label y axis


cat_plot.fig.set_size_inches(10,10) # Set width to 10 and height to 10
```

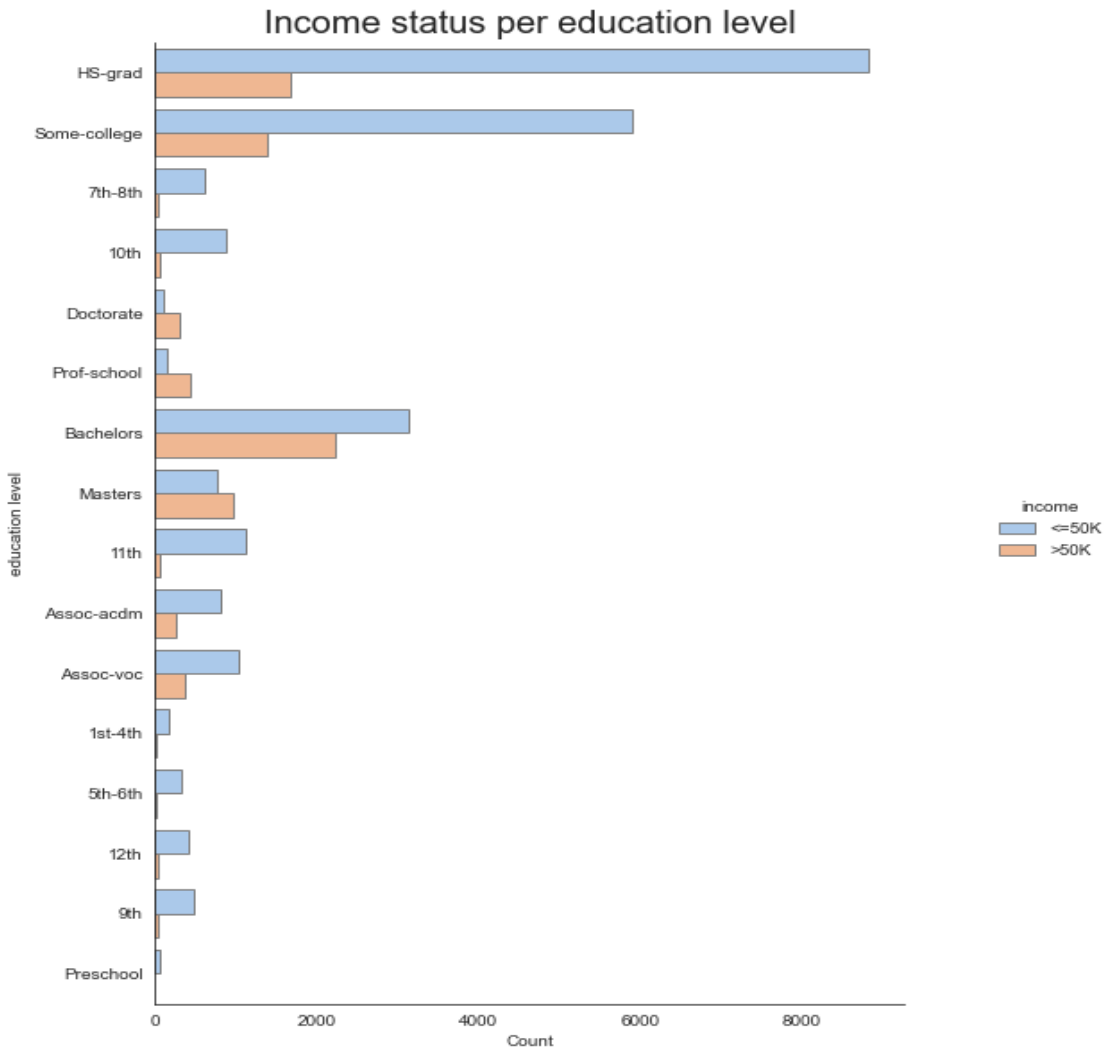

Figure 13 - Income status per education level

2.2 Use appropriate methods to handle categorical data

2.2.1 Dropping variables

The first task in handling the categorical data is dropping the 'education' variable. The education variable does not provide any additional information as it is already represented in numerical form as the 'education-num' variable.

```
# Drop education variable
census_data.drop(['education'], axis = 1, inplace = True)
```

2.2.2 Category reduction in variables

Reducing the number of categories in variables may prove useful in improving model accuracy (Allwein, Schapire and Singer 2000). As such categories that shared a similar relationship were categorized into one group. In the ‘marital-status’ variable, the 'Widowed',' Divorced', 'Separated' and 'Never-married' categories were all encoded as 'Single'. 'Married-civ-spouse', 'Married-spouse-absent' and 'Married-AF-spouse' on the other hand were encoded as 'Married'. Regarding the ‘native-country’ variable, since the dataset was collected in the United States most of the participants hailed from the there. As such all other regions were aggregated and encoded as 'Other-Country'.

```
# Find unique elements in marital-status
census_data['marital-status'].unique()
```

```
array(['Widowed', 'Divorced', 'Separated', 'Never-married',
       'Married-civ-spouse', 'Married-spouse-absent', 'Married-AF-spouse'],
      dtype=object)
```

```
# Reduce marital-status into single and married
census_data["marital-status"].replace(['Widowed','Divorced', 'Separated', 'Ne
ver-married',], 'Single',inplace= True)


census_data["marital-status"].replace(['Married-civ-spouse', 'Married-spouse-
absent', 'Married-AF-spouse',], 'Married' , inplace= True)
```

```
# Find unique elements in marital-status
census_data['native-country'].unique()
```

```
array(['United-States', 'Mexico', 'Greece', 'Vietnam', 'China', 'Taiwan',
       'India', 'Philippines', 'Trinadad&Tobago', 'Canada', 'South',
       'Holand-Netherlands', 'Puerto-Rico', 'Poland', 'Iran', 'England',
       'Germany', 'Italy', 'Japan', 'Hong', 'Honduras', 'Cuba', 'Ireland',
       'Cambodia', 'Peru', 'Nicaragua', 'Dominican-Republic', 'Haiti',
```

```
       'El-Salvador', 'Hungary', 'Columbia', 'Guatemala', 'Jamaica',
       'Ecuador', 'France', 'Yugoslavia', 'Scotland', 'Portugal', 'Laos',
       'Thailand', 'Outlying-US(Guam-USVI-etc)'], dtype=object)
```

```
# Categorize into US and other country
census_data['native-country'] = ["United-States" if i == "United-States" else
"Other-Country" for i in census_data['native-country']]
```

2.2.3 Label encoding

The final step in handling the categorical data in this study is label encoding. In label encoding, categories in variables are encoded into numerical form. This allows for a machine-readable format and facilitates the modelling process. An alternative to label encoding is one hot encoding where dummy variables are created by constructing a new binary feature for each category in a variable and indicating with the value 1 where it occurs in the dataset and 0 where it doesn't.

One hot encoding may not be appropriate when the number of categories is large, since more categories lead to more dimensions created, this may result in the "curse of dimensionality" (Koppen, 2000). The dimensionality curse simply implies that increasing dimensions or variables leads to an exponential increase in the computational resources required to process such dimensions. Since the dataset has high category variables like 'occupation' and 'relationship' the label encoding technique was utilized because it did not create additional dimensions. Table 6 shows the head of the label encoded dataset. The dependent variable 'income' now has 0 representing individuals who earn less than USD 50,000 and 1 for those who earn more than that amount.

```
#! pip install sklearn
from sklearn import preprocessing

# Select categories
categorical = ['workclass', 'marital-status', 'occupation', 'relationship', '
sex','native-country','income']
```

```
# Label encode
for feature in categorical:
        label_encoder = preprocessing.LabelEncoder()
        census_data[feature] = label_encoder.fit_transform(census_data[featur
e])
```

```
census_data.head()
```

Table 6-Head of label encoded data

	age	workclass	fnlwgt	education-num	marital-status	occupation	relationship	sex	capital-gain	capital-loss	hours-per-week	native-country	income
0	90	3	77053	9	1	9	1	0	0	4356	40	1	0
1	82	3	132870	9	1	3	1	0	0	4356	18	1	0
2	66	3	186061	10	1	9	4	0	0	4356	40	1	0
3	54	3	140359	4	1	6	4	0	0	3900	40	1	0
4	41	3	264663	10	1	9	3	0	0	3900	40	1	0

2.3 Machine learning application

2.3.1 Data splitting and class imbalance correction

To undertake the machine learning modelling process, the data was split into a train and test set with a ratio of 70:30 respectively. The data was also scaled with the MinMaxScaler which normalizes the data points to fit between the values of 0 and 1. Scaling is relevant to prevent higher values from dominating when machine learning models make distance computations. This enables models to understand the data better and facilitates faster training times (Zheng and Casari, 2018).

```
# !pip install -U imbalanced-learn
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split

scal = MinMaxScaler()
independent_dat = census_data.drop(["income"],axis=1) #Features
x = scal.fit_transform(independent_dat)
y = census_data['income']# Targe varibales

x_train, x_test, y_train, y_test = train_test_split(x, y, test_size = 0.3, ra
ndom_state = 42) # 30% test set
```

2.3.2 Classification models

Classification in machine learning involves predicting a class or categorical label for a given set of input data. The classification models used in this study are; Random forest, Gradient boost machine and XGboost.

2.3.2.1 Random forest

A member of the supervised learning algorithm family is the Decision tree algorithm. This algorithm's goal is to develop a model that can be used to infer simple decision rules from training data to predict the target variable (Quinlan, 1996).

The random forest classification model is an amalgamation of multiple decision trees. To develop an uncorrelated forest of trees whose forecast together is more accurate than that of any individual tree, random forest employs bagging and variable randomness when generating each individual tree (Rigatti, 2017).

The random forest algorithm at baseline produced an accuracy of about 86%. An analysis of the confusion matrix in figure 14, gives insight into the true negative, false positive, false negative and true positive. Below are the definitions of these metrics.

- **Accuracy** is the number of correctly predicted observations, expressed as a percentage of all the observations.
- **True positives** are positive predictions (1) that were actually positive (1).
- **True negatives** are negative predictions (0) that were actually negative (0).
- **False positives** are positive predictions (1) that were actually negative(0).
- **False negatives** are negative predictions (0) that were actually positive (1).

The random forest algorithm at baseline gave a true negative of 6964, a false positive of 465, a false negative of 897 and a true positive of 1443.

```
# Random Forest
# Import Random forest model
from sklearn.ensemble import RandomForestClassifier
from sklearn import metrics

# RandomForestClassifier
# random_state for repeatability
clf = RandomForestClassifier(random_state=76)
```

```
clf.fit(x_train, y_train) # fit model


# Train the model using the training sets

y_pred = clf.predict(X_test) # predict with model on test set
# Model Accuracy
accuracy_rf = metrics.accuracy_score(y_test, y_pred) # Find model accuracy
print('Accuracy:', accuracy_rf) # print accuracy

Accuracy: 0.8605793837649708
```

```
# classification report
from sklearn.metrics import classification_report, confusion_matrix
matrix = confusion_matrix(y_test,y_pred) # Confusoin matrix
sns.heatmap(matrix, annot = True, fmt = 'd')
plt.title('confustion Matrix') # plot title
plt.xlabel('Predicted') # x axis
plt.ylabel('True') # y axis
print(classification_report(y_test,y_pred)) # print report
```

	precision	recall	f1-score	support
0	0.89	0.94	0.91	7429
1	0.76	0.62	0.68	2340
accuracy			0.86	9769
macro avg	0.82	0.78	0.80	9769
weighted avg	0.85	0.86	0.86	9769

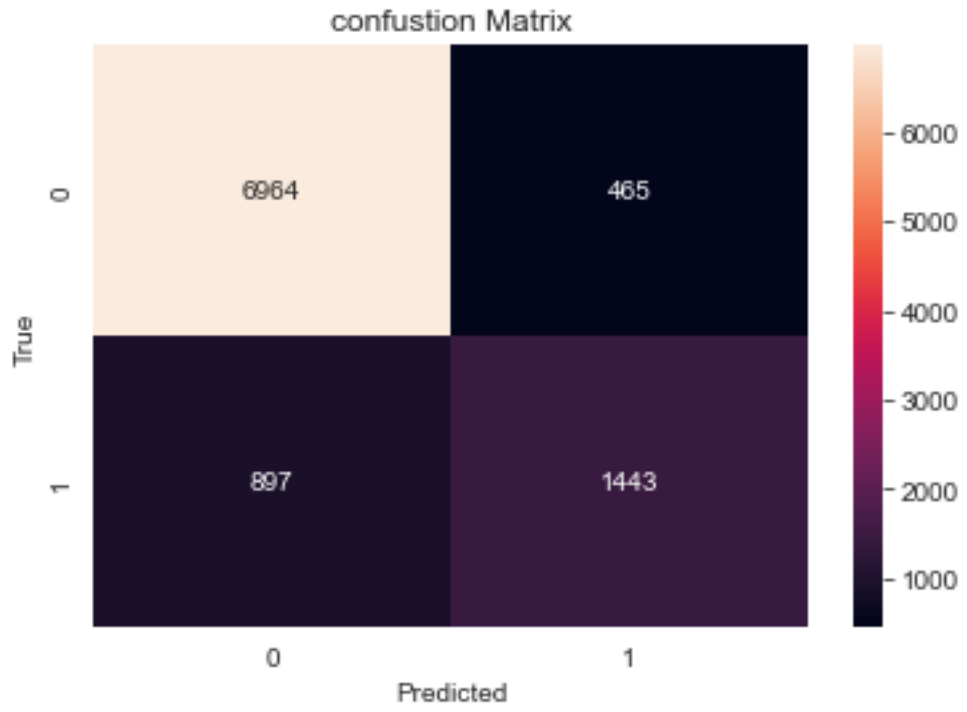

Figure 14-Confusion matrix for random forest

2.3.2.2 Gradient boost machine (GBM)

Gradient boosting is predicated on the notion that the best subsequent model, when used in conjunction with earlier models, minimizes overall prediction error. The main concept is to define the desired results for this subsequent model to reduce mistakes. Depending on how modifying a case's prediction affects the total prediction error, the desired result will vary for each instance in the data:

- The case's next target outcome is a high value if a slight modification in the prediction leads to a significant decrease in inaccuracy. The inaccuracy will be decreased by the new model's predictions that are quite near to its goals.
- The next intended outcome of the case is zero if a modest modification in the prediction for a case has no impact on the error. This forecast cannot be altered to reduce the error.

Because target outcomes are determined for each case based on the gradient of the error concerning the prediction, this method is known as gradient boosting. In the space of

potential predictions for each training case, each new model moves in the direction that minimizes prediction error (Natekin and Knoll, 2013).

GBM at baseline produced an accuracy of about 86.5% From figure 15 GBM gave a true negative of 7082, a false positive of 347, a false negative of 968 and a true positive of 1372.

```
from sklearn.ensemble import GradientBoostingClassifier
gbc_model = GradientBoostingClassifier(random_state=76)
gbc_model.fit(x_train, y_train)
y_pred = gbc_model.predict(x_test)
# Model Accuracy
accuracy_gbm = metrics.accuracy_score(y_test, y_pred)
print('Accuracy:', accuracy_gbm)

Accuracy: 0.86539052103593
```

```
# classification report
from sklearn.metrics import classification_report, confusion_matrix
matrix = confusion_matrix(y_test,y_pred)
sns.heatmap(matrix, annot = True, fmt = 'd')
plt.title('confustion Matrix')
plt.xlabel('Predicted')
plt.ylabel('True')
print(classification_report(y_test,y_pred))
```

	precision	recall	f1-score	support
0	0.88	0.95	0.92	7429
1	0.80	0.59	0.68	2340
accuracy			0.87	9769
macro avg	0.84	0.77	0.80	9769
weighted avg	0.86	0.87	0.86	9769

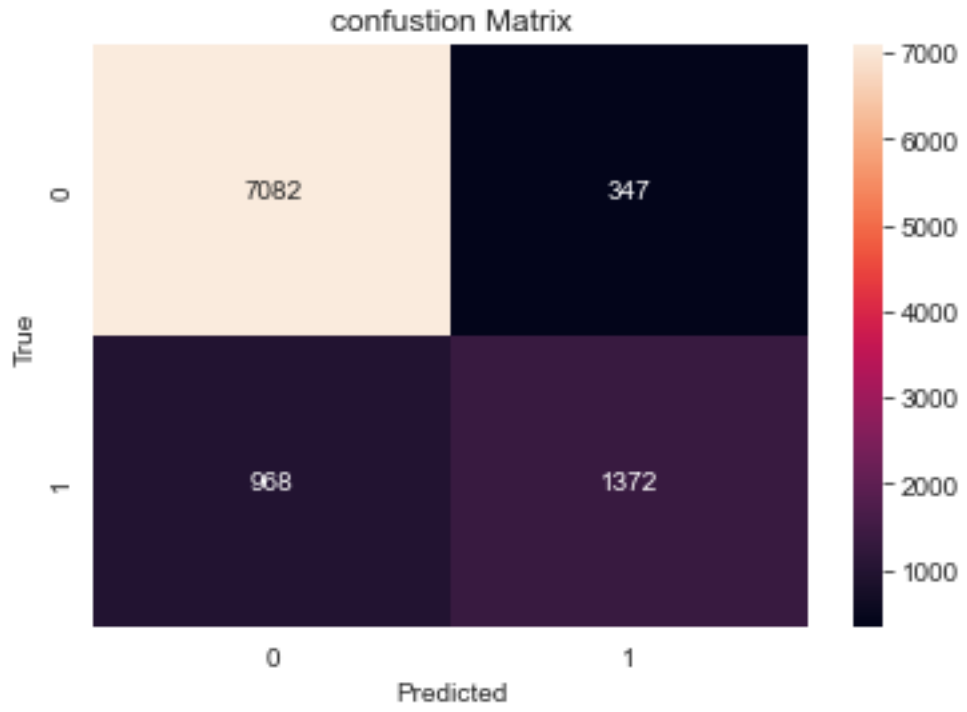

Figure 15-Confusion matrix for gradient boost machine

2.3.3.3 XGboost

Extreme Gradient Boosting (XGBoost) is a distributed, scalable gradient-boosted decision tree (GBDT) machine learning framework introduced by Tianqi Chen and Carlos Guestrin (2016). It functions by fusing gradient bossing and decision trees. With optimization features like Gradient Boosting, Stochastic Gradient Boosting, and Regularized Gradient Boosting, XGBoost focuses on computational speed and model performance.

The XGBoost technique is used to optimize the utilization of memory and computation resources. The approach incorporates sparse aware implementation by addressing missing values from data sets automatically. The technique also permits continuous training of a model that has previously been fitted on fresh data.

```
# ! pip install xgboost
from xgboost import XGBClassifier
xgb_model = XGBClassifier(random_state=76)
xgb_model.fit(x_train, y_train)
y_pred = xgb_model.predict(x_test)
# Model Accuracy
accuracy_xgb = metrics.accuracy_score(y_test, y_pred)
print('Accuracy:', accuracy_xgb)

Accuracy: 0.8714300337803256
```

XGboost at baseline produced an accuracy of about 87.1% From figure 16 XGboost gave a true negative of 6987, a false positive of 442, a false negative of 814 and a true positive of 1526.

```
# classification report
from sklearn.metrics import classification_report, confusion_matrix
matrix = confusion_matrix(y_test,y_pred)
sns.heatmap(matrix, annot = True, fmt = 'd')
plt.title('confustion Matrix')
plt.xlabel('Predicted')
plt.ylabel('True')
print(classification_report(y_test,y_pred))

              precision    recall  f1-score   support

           0       0.90      0.94      0.92      7429
           1       0.78      0.65      0.71      2340

    accuracy                           0.87      9769
   macro avg       0.84      0.80      0.81      9769
weighted avg       0.87      0.87      0.87      9769
```

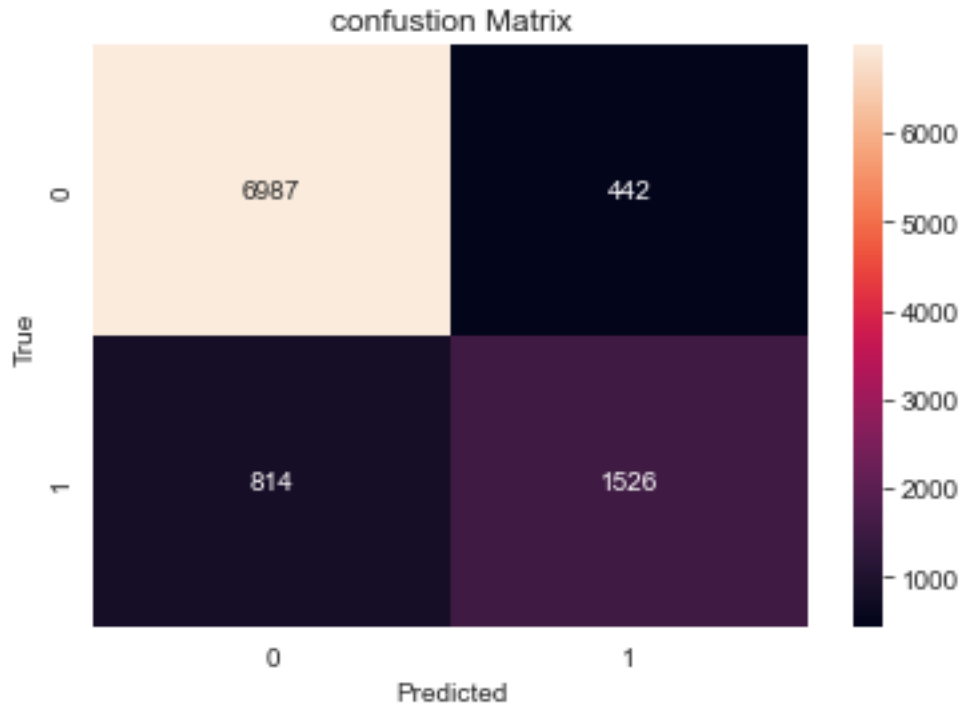

Figure 16-Confusion matrix for XGboost

2.3.3 Clustering

Clustering is a process of grouping data so that data points similar to each other are within the same group. The kinds of clustering algorithms discussed in this study are K-means and the gaussian mixture model.

2.3.3.1 K-means clustering

In the K-means clustering algorithm, the partitioning of the data space makes it feasible for data points from the same cluster to be as similar as possible to one another while making data points from separate clusters as distinct as possible. Each cluster in K-means is represented by its centroid, which is the arithmetic mean of the data points assigned to the cluster. A centroid is a data point that symbolizes the cluster's mean, and it isn't required that it be part of the dataset. In this approach, the algorithm minimizes intra-cluster distance at each step until each data point is closer to its own cluster's centroid than to other clusters' centroid. K-means employs an iterative process to find a final clustering depending on the number of clusters defined by the user by searching for a predetermined

number of clusters within an unlabeled dataset (represented by the variable K). Putting "k" equal to 2, for instance, will divide the dataset into two clusters, whereas putting "k" equal to 4 will divide it into four clusters. To arrive at a final clustering of the data points, K-means iteratively calculates new centroids after starting its process with randomly selected data points as suggested centroids of the groups (Li and Wu, 2012).

To commence with K-means the labelled dependent variable 'income' was dropped, enabling the algorithm to make its own clusters and labels. The data was also scaled with the standard scaler since K-means relies on the calculation of Euclidean Distance to form groups. Without scaling K-means will put more weight on higher values and such higher variances, leading clusters to be separated along such values and becoming isotropic in all directions. (Kang and Cho, 2009).

```
# Feature scaling
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
df = sc.fit_transform(census_data.drop('income', axis = 1))
```

Since the dependent variable had 2 classes the number of clusters was set to 2. The labels produced by the K-means algorithm were compared with the dependent variable and an accuracy of about 69.8% was obtained at baseline. From figure 17 The K-means algorithm produced a true negative of 16502, a false positive of 1584, a false negative of 8218 and a true positive of 6257.

```
y_actual = census_data['income']

kmeans = KMeans(n_clusters = 2, random_state= 3)
y_kmeans = kmeans.fit_predict(df)
kmeans_score = metrics.accuracy_score(y_kmeans, y_actual)
print('Accuracy:', metrics.accuracy_score(y_kmeans, y_actual))

Accuracy: 0.6989650195018581
```

```
# classification report
from sklearn.metrics import classification_report, confusion_matrix
matrix = confusion_matrix(y_kmeans, y_actual)
sns.heatmap(matrix, annot = True, fmt = 'd')
plt.title('confustion Matrix')
plt.xlabel('Predicted')
plt.ylabel('True')
print(classification_report(y_kmeans, y_actual))
```

```
              precision    recall  f1-score   support

           0       0.67      0.91      0.77     18086
           1       0.80      0.43      0.56     14475

    accuracy                           0.70     32561
   macro avg       0.73      0.67      0.67     32561
weighted avg       0.73      0.70      0.68     32561
```

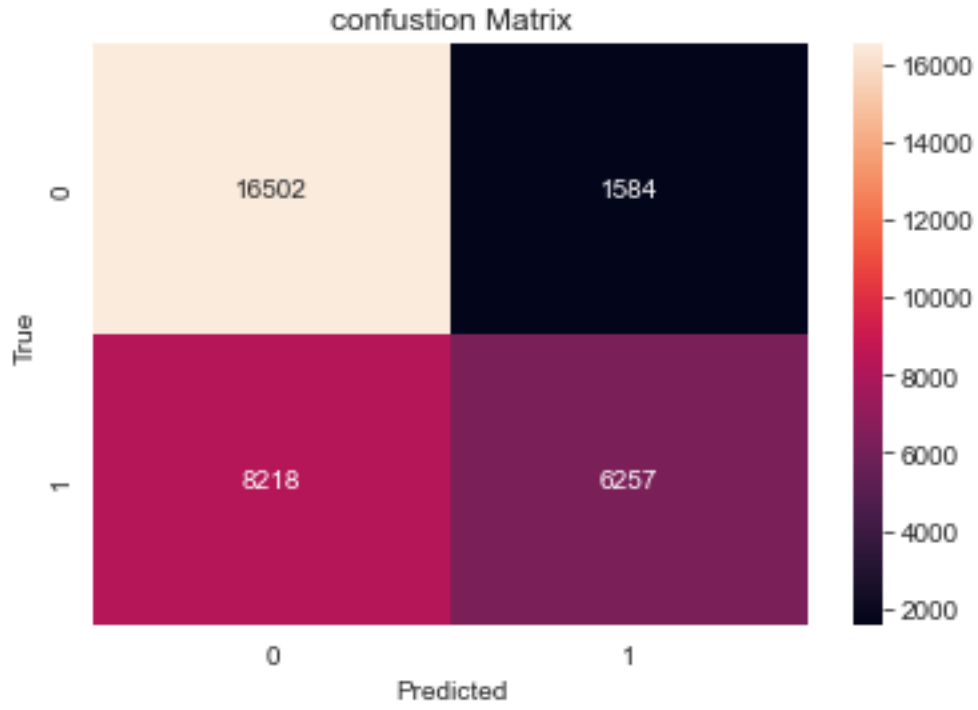

Figure 17-Confusion matrix for K-means

2.3.3.1 Gaussian Mixture Model (GMM)

A probabilistic model called a "Gaussian mixing model" posits that all of the data points are produced by combining a limited number of Gaussian distributions with unknowable parameters. Gaussian mixture models can be seen as a generalization of k-means clustering to include details of the covariance structure of the data as well as the locations of the latent Gaussian centres. Unlike K-means, a set of k gaussian is fitted to the data instead of locating clusters by "nearest" centroids. Additionally, the mean, weight and variance of each cluster are calculated according to the gaussian distribution parameters. The number of probabilities of each data point belonging to each cluster is then determined after learning the parameters for each data point (Yang, Lai and Lin, 2012).

Again, since the dependent variable had 2 classes the number of components was set to 2. The labels produced by the GMM algorithm were compared with the dependent variable and an accuracy of about 71.7% was obtained at baseline. From figure 18 The GMM algorithm produced a true negative of 17446, a false positive of 1923, a false negative of 7274 and a true positive of 5918.

```
from sklearn.mixture import GaussianMixture
gmm = GaussianMixture(n_components=2,random_state=3)
gmm.fit(df)

#predictions from gmm
GM_labels = gmm.predict(df)
gmm_score = metrics.accuracy_score(GM_labels, y_actual)
print('Accuracy:', metrics.accuracy_score(GM_labels, y_actual))

Accuracy: 0.717545529928442
```

```
# classification report
from sklearn.metrics import classification_report, confusion_matrix
matrix = confusion_matrix(GM_labels, y_actual)
sns.heatmap(matrix, annot = True, fmt = 'd')
plt.title('confustion Matrix')
plt.xlabel('Predicted')
plt.ylabel('True')
print(classification_report(y_kmeans, y_actual))
```

```
              precision    recall  f1-score   support

           0       0.67      0.91      0.77     18086
           1       0.80      0.43      0.56     14475

    accuracy                           0.70     32561
   macro avg       0.73      0.67      0.67     32561
weighted avg       0.73      0.70      0.68     32561
```

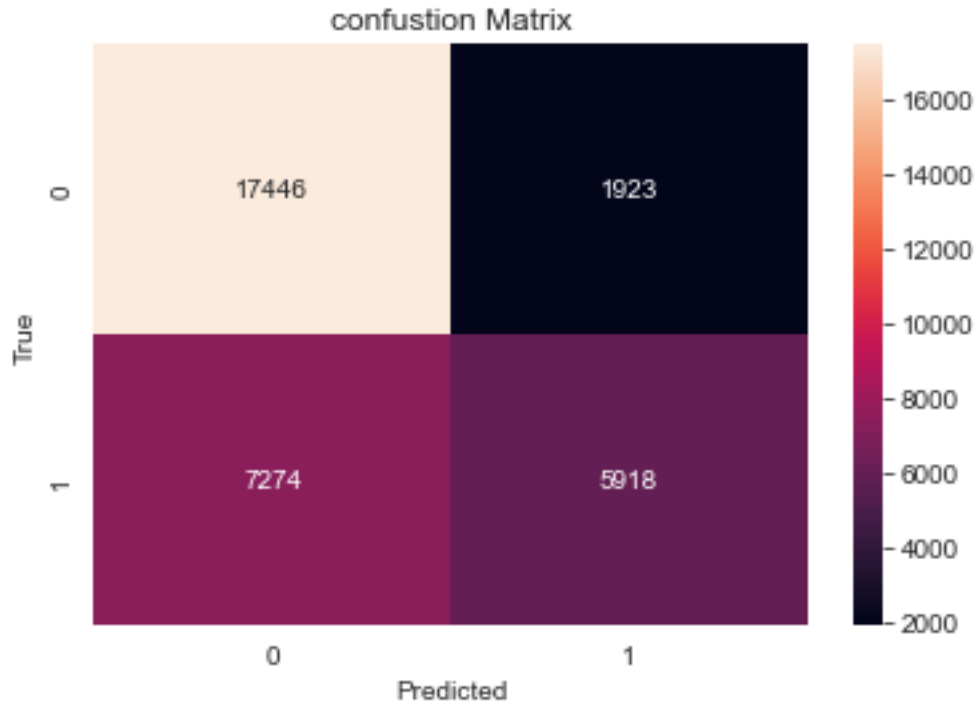

Figure 18-Confusion matrix for Gaussian Mixture Model

2.3.4 Artificial Neural network

The biological neural networks that shape the structure of the human brain are where the phrase "artificial neural network" originates. Artificial neural networks also feature neurons that are interconnected to one another at different levels of the networks, much like the human brain, which has neurons that are interconnected to one another. In these networks are the following layers;

- **Input Layer**: It accepts inputs in the many available formats.
- **The hidden layer**: It is located between the input and output layers. It makes all the computations necessary to uncover patterns and buried features.
- **Output Layer**: This layer is used to communicate the output after the input has undergone several alterations in the hidden layer.

When given input, the artificial neural network computes the weighted total of the inputs and incorporates a bias. A transfer function is used to visualize this computation. To produce the output, it passes the weighted total as an input to an activation function. A node's activation functions determine whether it should fire. The output layer is only accessible to individuals who are fired. Depending on the type of task we are completing, there are many activation functions that can be used (Yegnanarayana,2009).

There are various neural network types, and each has a unique set of applications and degrees of complexity, but for the purpose of this study the feedforward neural network, in which information travels in only one direction from input to output will be focused on.

Two classifiers will also be considered namely the multi-layer perceptron classifier (MLPClassifier) and the KerasClassifer.

2.3.4.1 Multi-Layer Perceptron (MLP) classifier

The MLPClassifier is an implementation of the feedforward neural network obtained from the sklearn package. A maximum iteration of 500 was set to allow room for convergence.

```
%%time # Check time used in training model in jupyter NB


# Importing MLPClassifier
from sklearn.neural_network import MLPClassifier
# Initializing the MLPClassifier
# Random state for repeatability
# Max_iter for 500 maximum iterations facilitating convergence.
mlp_classifier = MLPClassifier(random_state=76, max_iter=500)
mlp_classifier.fit(x_train, y_train)

CPU times: total: 14.8 s
Wall time: 14.7 s
```

MLPClassifer at baseline produced an accuracy of about 85%. From figure 19 the MLPClassifer gave a true negative of 6960, a false positive of 469, a false negative of 996 and a true positive of 1344.

```
from sklearn import metrics
#Predicting y for X_val
y_pred = mlp_classifier.predict(x_test)
accuracy_mlp = metrics.accuracy_score(y_test, y_pred)
print('Accuracy:', accuracy_mlp)

Accuracy: 0.8500358276179752
```

```
# classification report
from sklearn.metrics import classification_report, confusion_matrix
matrix = confusion_matrix(y_test,y_pred)
sns.heatmap(matrix, annot = True, fmt = 'd')
```

```
plt.title('confustion Matrix')
plt.xlabel('Predicted')
plt.ylabel('True')
print(classification_report(y_test,y_pred))
```

```
              precision    recall  f1-score   support

           0       0.87      0.94      0.90      7429
           1       0.74      0.57      0.65      2340

    accuracy                           0.85      9769
   macro avg       0.81      0.76      0.78      9769
weighted avg       0.84      0.85      0.84      9769
```

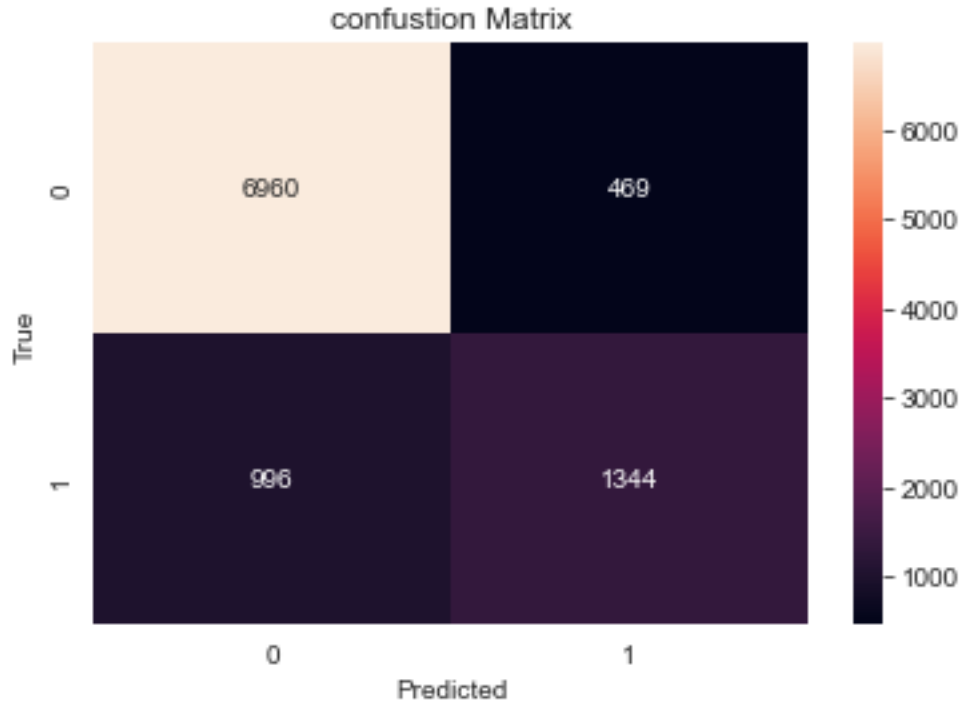

Figure 19-Confusion matrix for MLP Classifier

2.3.4.2 KerasClassifier

Just like the MLPClassifier the KerasClassifier is another implementation of the feedforward neural network but obtained from the tensorflow package. For the base model, a total of 10 neurons was implemented and 1 hidden layer was used.

```
# ! pip install tensorflow
from tensorflow import keras
import tensorflow as tf
from keras.wrappers.scikit_learn import KerasClassifier
```

```
# Initialising ANN
# first we need to define the model
def ANN1(neurons=10, hidden_layers=0, dropout_rate=0, learn_rate= 0.1,init_mo
de='uniform', activation='relu'):
    # model
    model = keras.Sequential()
    model.add(keras.layers.Dense(neurons, input_shape = (X_train.shape[1], ),
activation='relu'))
    for i in range(hidden_layers):
        # Add one hidden layer
        model.add(keras.layers.Dense(neurons, activation='relu'))
        model.add(keras.layers.Dropout(dropout_rate))
    model.add(keras.layers.Dense(1, activation='sigmoid'))
    # Compile model
    optimizer = keras.optimizers.SGD(lr=learn_rate, momentum = 0)
    model.compile(loss='binary_crossentropy', optimizer=optimizer, metrics=['
accuracy'])
    return model

# we will do the grid search with KerasClassifier

ann0 = KerasClassifier(build_fn=ANN1)
```

```
tf.random.set_seed(76) # set randomness for tensorflow function
X_train = x_train
ann0.fit(X_train,y_train)
```

```
from sklearn import metrics
#Predicting y for X_val
y_pred = ann0.predict(x_test)
accuracy_ann = metrics.accuracy_score(y_test, y_pred)
print('Accuracy:', accuracy_ann)
```

```
306/306 [==============================] - 0s 667us/step
Accuracy: 0.8208619101238612
```

KerasClassier at baseline produced an accuracy of about 82%. From figure 20 the KerasClassier gave a true negative of 6950, a false positive of 479, a false negative of 1271 and a true positive of 1069.

```
# classification report
from sklearn.metrics import classification_report, confusion_matrix
matrix = confusion_matrix(y_test,y_pred)
sns.heatmap(matrix, annot = True, fmt = 'd')
plt.title('confustion Matrix')
plt.xlabel('Predicted')
plt.ylabel('True')
print(classification_report(y_test,y_pred))
```

```
              precision    recall  f1-score   support

           0       0.85      0.94      0.89      7429
           1       0.69      0.46      0.55      2340
```

accuracy			0.82	9769
macro avg	0.77	0.70	0.72	9769
weighted avg	0.81	0.82	0.81	9769

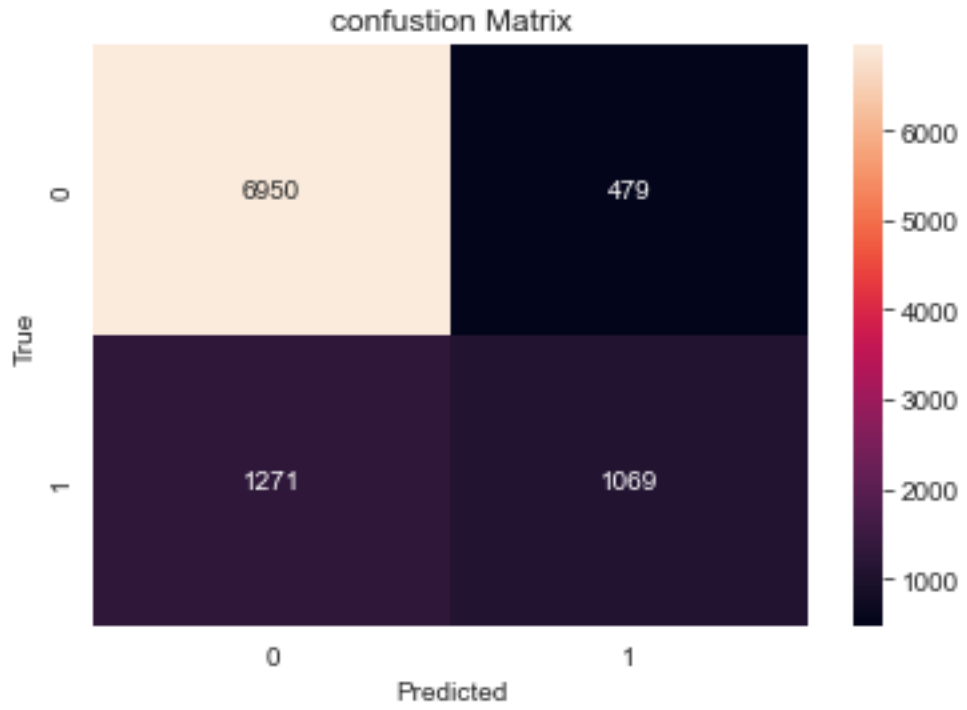

Figure 20 - Confusion matrix for Keras Classifier

2.4 Model optimization and comparison

Optimization in machine learning involves finding the input parameters to a model that result in the minimum or maximum output of that model. For this study, the overall accuracy is the metric of interest, as such maximization would be the objective of the optimization process.

For classification and neural network optimization, several hyperparameters spanning a range of values may be utilized. Finding the optimal combination for all parameters can be a tasking process and may result in extensive time and resource consumption.

To solve this problem, the RadomizedSearchCV algorithm from sklearn was implemented. RadomizedSearchCV. After training N different versions of the model with various randomly chosen hyperparameter combinations, the algorithm selects the best successful version it has seen, leaving a model trained on a nearly ideal set of hyperparameters. An advantage of this is that the number of iterations, or N, can be controlled to manage computation resources and time. Also, adding additional parameters does not reduce the model's efficacy.

2.4.1 Optimization for classification models and comparison

2.4.1.1 Optimization for random forest and comparison

The hyperparameters for random forest optimization were.

- **N_estimators :** This is the number of trees to use in the model. A range of 100 to 500 trees with intervals of 50 was configured for optimization.
- **Criterion:** This is used to gauge how well the split or loss function that determines model results is working. Entropy and Gini were employed. Gini determines the likelihood that a certain feature would be erroneously classified when chosen at random. On the other hand, entropy quantifies the degree of uncertainty in a set of observations from the data.

- **Max_features:** This is the maximum number of features the model will consider when determining a split. Auto and sqrt were used. Auto implies that the maximum number would be determined automatically, and sqrt implies that the number would be determined by taking the square root of the features.
- **Max_depth:** The maximum depth of the tree. This was an integer ranging from None and 10 to 20.
- **Min_samples_split:** This is the minimum number of samples required to split an internal node. The integer values of 2, 5 and 10 were provided.
- **Min_samples_leaf:** This is the minimum number of samples required to be at a leaf node. The integer values of 1, 2 and 4 were provided.
- **Bootstrap:** This indicates whether bootstrap samples are used when building trees. If False, the whole dataset is used to build each tree. Bootstrapping uses random sampling with replacement and assigns measures of accuracy to sample estimates (Probst, Wright and Boulesteix, 2019).

```
from sklearn.model_selection import RandomizedSearchCV
# Number of trees in random forest
n_estimators = [int(x) for x in np.linspace(start = 100, stop = 500, num = 9)
]

# The function to measure the quality of a split.
criterion = ["gini", "entropy"]
# Number of features to consider at every split
max_features = ['auto', 'sqrt']
# Maximum number of levels in tree
max_depth = [int(x) for x in np.linspace(10, 20, num = 10)]
max_depth.append(None)
# Minimum number of samples required to split a node
min_samples_split = [2, 5, 10]
# Minimum number of samples required at each leaf node
min_samples_leaf = [1, 2, 4]
```

```
# Method of selecting samples for training each tree
bootstrap = [True, False]
# Create the random grid
random_grid = {'n_estimators': n_estimators,
               'max_features': max_features,
               'criterion': criterion,
               'max_depth': max_depth,
               'min_samples_split': min_samples_split,
               'min_samples_leaf': min_samples_leaf,
               'bootstrap': bootstrap}


random_grid
```

```
{'n_estimators': [100, 150, 200, 250, 300, 350, 400, 450, 500],
 'max_features': ['auto', 'sqrt'],
 'criterion': ['gini', 'entropy'],
 'max_depth': [10, 11, 12, 13, 14, 15, 16, 17, 18, 20, None],
 'min_samples_split': [2, 5, 10],
 'min_samples_leaf': [1, 2, 4],
 'bootstrap': [True, False]}
```

The RandomizedSearchCV algorithm was set to 100 iterations with three-fold cross validations. Cross-validation involves using different portions of the training dataset to test and train a model on different iterations, in this case,three.

```
# Use the random grid to search for best hyperparameters
# First create the base model to tune
rf = RandomForestClassifier(random_state = 76)
# Random search of parameters, using 3 fold cross validation,
# search across 100 different combinations, and use all available cores
rf_opt_model = RandomizedSearchCV(estimator = rf, param_distributions = rando
m_grid,
                                  n_iter = 100,
                                  cv = 3,
                                  verbose=2, # to show task
```

```
                                    random_state=42, # for repeatability
                                    n_jobs = -1) # to use all CPUs
```

After optimization an accuracy of about 86.6% was obtained with the best parameters for n_estimators, min_samples_split, min_samples_leaf, max_features, max_depth, criterion and bootstrap were 400, 5, 1, auto, 16, entropy and False respectively. From figure 21 the optimized random forest algorithm gave a true negative of 7053, a false positive of 379, a false negative of 934 and a true positive of 1406.

```
# Fit the random search model
rf_opt_model.fit(x_train, y_train)

rf_opt_model.best_params_

{'n_estimators': 400,
 'min_samples_split': 5,
 'min_samples_leaf': 1,
 'max_features': 'auto',
 'max_depth': 16,
 'criterion': 'entropy',
 'bootstrap': False}

# Model Accuracy
y_pred = rf_opt_model.predict(x_test)
accuracy_rf_opt = metrics.accuracy_score(y_test, y_pred)
print('Accuracy:', accuracy_rf_opt)

Accuracy: 0.8659023441498618
```

```
# classification report

from sklearn.metrics import classification_report, confusion_matrix

matrix = confusion_matrix(y_test,y_pred)

sns.heatmap(matrix, annot = True, fmt = 'd')

plt.title('confustion Matrix')

plt.xlabel('Predicted')

plt.ylabel('True')

print(classification_report(y_test,y_pred))
```

```
              precision    recall  f1-score   support

           0       0.88      0.95      0.92      7429
           1       0.79      0.60      0.68      2340

    accuracy                           0.87      9769
   macro avg       0.84      0.78      0.80      9769
weighted avg       0.86      0.87      0.86      9769
```

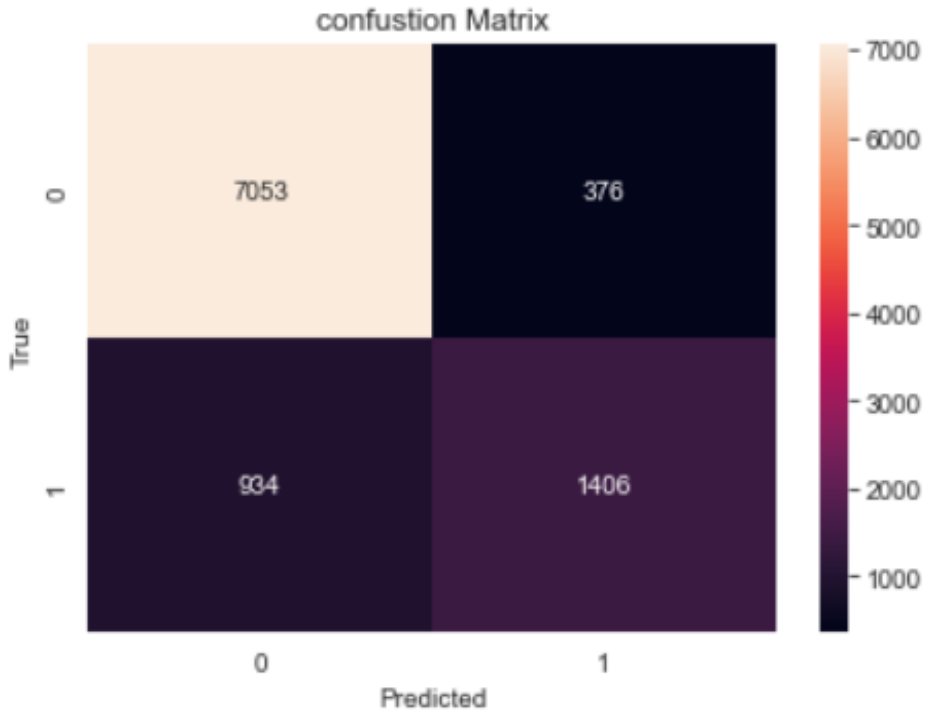

Figure 21-Confusion matrix for optimized random forest

2.4.1.2 Optimization for gradient boost machine and comparison

The hyperparameters selected for gradient boost machine optimization were.

- **N_estimators :** The number of boosting stages to use for the model. 5, 50, 250 and 500 boosting were set of the optimization.
- **Max_depth:** The maximum depth of the tree. Integers of 1, 3, 5, 7 and 9 were configured.
- **Learning_rate:** The learning rate is a hyperparameter that determines how much to alter the model each time the model weights are updated in response to the predicted error. 0.01, 0.1, 1, 10 and 100 were configured.
- **Max_leaf_nodes:** Maximum number of leaf nodes Integers of `2, 5, 10, 20, 50 and 100` were configured (Putatunda and Rama, 2019).

```
from sklearn.ensemble import GradientBoostingClassifier
gbc = GradientBoostingClassifier(random_state=76)
gbc_parameters = {
    "n_estimators":[5,50,250,500],
    "max_depth":[1,3,5,7,9],
    "learning_rate":[0.01,0.1,1,10,100],
    "max_leaf_nodes": [2, 5, 10, 20, 50, 100]
}
```

The gradient boost machine algorithm was set to 100 iterations with three-fold cross-validation.

```
gbm_opt_model = RandomizedSearchCV(estimator = gbc,
                                   param_distributions = gbc_parameters,
                                   n_iter = 100,
                                   cv = 3,
                                   verbose=2, # to show task
                                   random_state=42, , # for repeatability
                                   n_jobs = -1) # to use all CPUs
```

After optimization an accuracy of about 87.2% was obtained with the best parameters for n_estimators, max_leaf_nodes, max_depth and learning_rate was 50, 50, 7 and 0.1 respectively. From figure 22 the optimized gradient boosting algorithm gave a true negative of 7051, a false positive of 378, a false negative of 875 and a true positive of 1465.

```
# Fit the random search model
gbm_opt_model.fit(x_train, y_train)

gbm_opt_model.best_params_
```

```
{'n_estimators': 50,
 'max_leaf_nodes': 50,
 'max_depth': 7,
 'learning_rate': 0.1}

# Model Accuracy
y_pred = gbm_opt_model.predict(x_test)
accuracy_gbm_opt = metrics.accuracy_score(y_test, y_pred)
print('Accuracy:', accuracy_gbm_opt)

Accuracy: 0.8717371276486846
```

```
# classification report
from sklearn.metrics import classification_report, confusion_matrix
matrix = confusion_matrix(y_test,y_pred)
sns.heatmap(matrix, annot = True, fmt = 'd')
plt.title('confustion Matrix')
plt.xlabel('Predicted')
plt.ylabel('True')
print(classification_report(y_test,y_pred))
```

	precision	recall	f1-score	support
0	0.89	0.95	0.92	7429
1	0.79	0.63	0.70	2340
accuracy			0.87	9769
macro avg	0.84	0.79	0.81	9769
weighted avg	0.87	0.87	0.87	9769

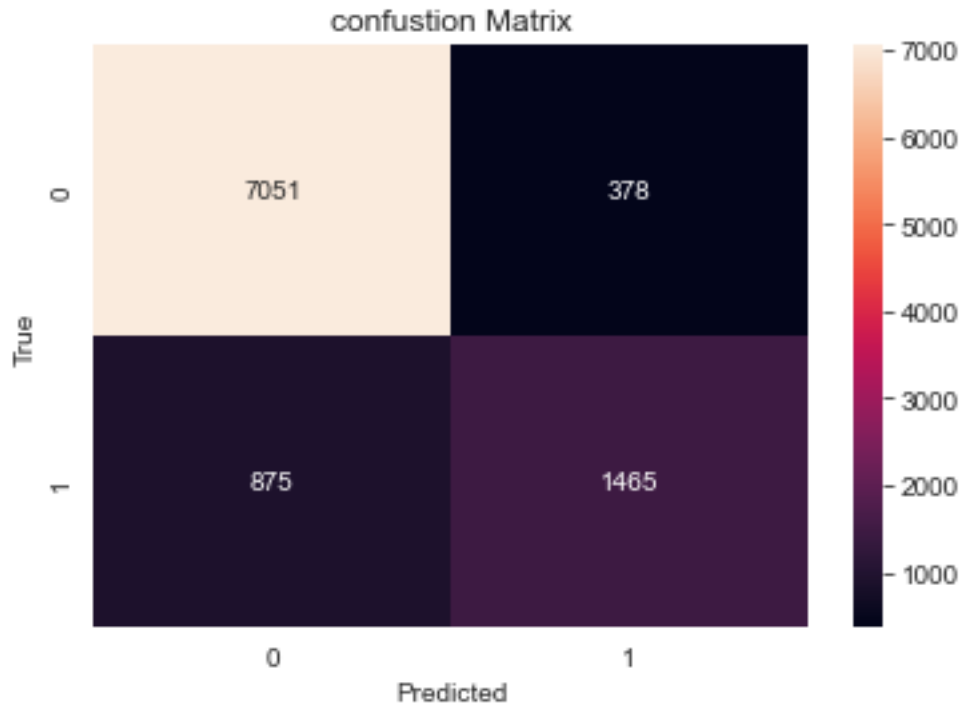

Figure 22 - Confusion matrix for gradient boost machine

2.4.1.3 Optimization for XGboost and comparison

The hyperparameters selected for gradient boost machine optimization were.

- **Max_depth:** The maximum depth of the tree. Integers of 3 to 10 was used with intervals of 2.
- **Min_child_weight:** This is the minimum sum of instance weight needed in a child. Integers of 1 to 6 with intervals of 2 was configured.
- **Gamma:** This is the bare minimal amount of loss reduction needed to partition the tree's leaf node further. The more conservative the algorithm is, the less fuzzy it is and the fewer results it yields, the bigger the gamma. A continuous range of 0 to 10 with intervals of 0.1 was set up.
- **Subsample:** A ratio of the training instances is shown here. If this is set it to 0.5, XGBoost will randomly select 50% of the training data before creating trees. and

doing so will stop overfitting. Every boosting cycle will include one subsampling. A continuous range of 6 to 10 with intervals of 0.1 was set up (Putatunda and Rama, 2018).

```
xgb = XGBClassifier(random_state=76)


xgb_param = {
 'max_depth': list(range(3,10,2)),
 'min_child_weight': list(range(1,6,2)),
 'gamma': [i/10.0 for i in range(0,10)],
 'subsample':[i/10.0 for i in range(6,10)]}


xgb_param
```

```
{'max_depth': [3, 5, 7, 9],
 'min_child_weight': [1, 3, 5],
 'gamma': [0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9],
 'subsample': [0.6, 0.7, 0.8, 0.9]}
```

The XGboost machine algorithm was set to 100 iterations with three-fold cross validations.

```
xgb_opt_model = RandomizedSearchCV(estimator = xgb, param_distributions = xgb
_param,
                                   n_iter = 100,
                                   cv = 3,
                                   verbose=2,
                                   random_state=42,
                                   n_jobs = -1)
```

After optimization an accuracy of about 87.3% was obtained with the best parameters for subsample, min_child_weight, max_depth and gamma were 0.9, 3, 3 and 0.1 respectively.

From figure 23 the optimized XGboost gave a true negative of 7039, a false positive of 390, a false negative of 849 and a true positive of 1491.

```
# Fit the random search model
xgb_opt_model.fit(x_train, y_train)

xgb_opt_model.best_params_

{'subsample': 0.9, 'min_child_weight': 3, 'max_depth': 3, 'gamma': 0.1}

# Model Accuracy
y_pred = xgb_opt_model.predict(x_test)
accuracy_xgb_opt = metrics.accuracy_score(y_test, y_pred)
print('Accuracy:', accuracy_xgb_opt)

Accuracy: 0.8731702323676938

# classification report
from sklearn.metrics import classification_report, confusion_matrix
matrix = confusion_matrix(y_test,y_pred)
sns.heatmap(matrix, annot = True, fmt = 'd')
plt.title('confustion Matrix')
plt.xlabel('Predicted')
plt.ylabel('True')
print(classification_report(y_test,y_pred))

              precision    recall  f1-score   support

           0       0.89      0.95      0.92      7429
           1       0.79      0.64      0.71      2340

    accuracy                           0.87      9769
   macro avg       0.84      0.79      0.81      9769
weighted avg       0.87      0.87      0.87      9769
```

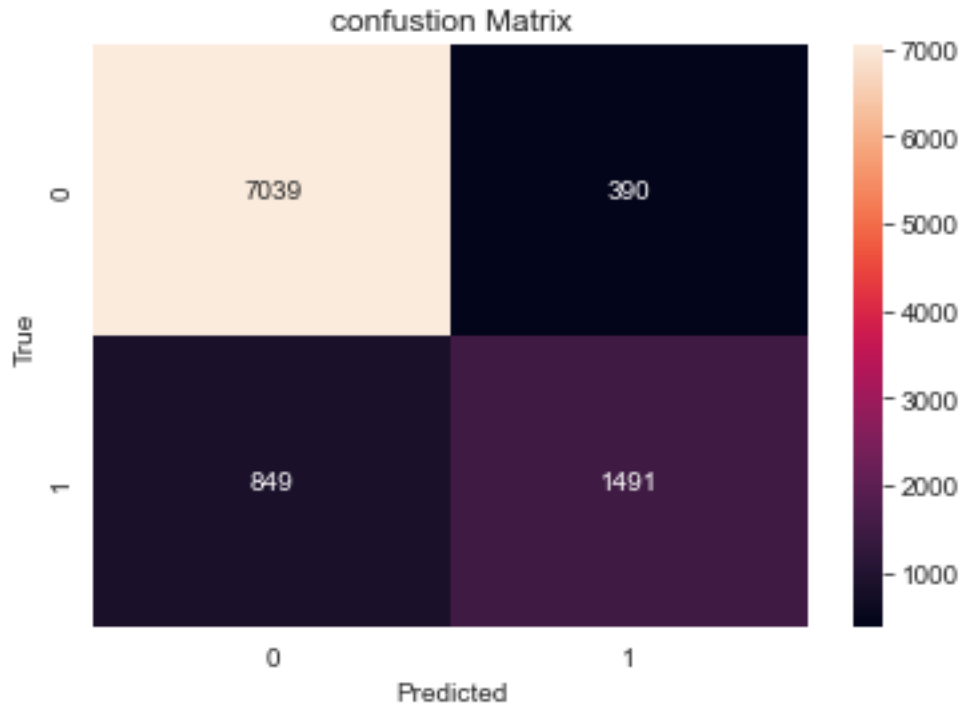

Figure 23 - Confusion matrix for XGboost

2.4.2 Optimization for clustering models

Fränti and Sieranoja (2019) purported that the clustering algorithms may be improved with the use of better initializations and repetitions. As such to optimize the K-means and GMM clustering models, the n_init parameter would be tuned with a range of values from 1 to 40. This parameter is the number of times the algorithms will be initialized and repeated with different centroid seeds.

2.4.2.1 Optimization for K-means clustering models

```
%%time # time of process
init_number = [] # empty list
scores = [] # empty list
for i in range(1,40): # loop from 1 to 40 for n_init
    kmeans = KMeans(n_clusters = 2,
                    random_state= 3,
```

```
                    n_init= i)
    y_kmeans = kmeans.fit_predict(df) # predict clusters with dataframe
    init_number.append(i) # Append in init_number variable
    scr = metrics.accuracy_score(y_kmeans, y_actual) # find accuracy of model
    scores.append(scr) # append score in scores variable

CPU times: total: 55.6 s
Wall time: 7.38 s
```

From table 7 the n_init parameter of 1 seemed to have the best accuracy of about 70%. From figure 24 the accuracy fell slightly after the initialization of 1, but then saw a sharp drop from 17 to 19 initializations, recovering to the previous value afterwards.

```
# Put init_number and scores in datafame kmeans_df

kmeans_df = pd.DataFrame(zip(init_number,scores),

 columns = ['number_of_initializations', 'scores'])

import matplotlib.pyplot as plt # Impot the relevant module
fig, ax = plt.subplots() # Create the figure and axes object
# Plot the first x and y axes:
kmeans_df.plot(x = 'number_of_initializations', y = 'scores', ax = ax)
```

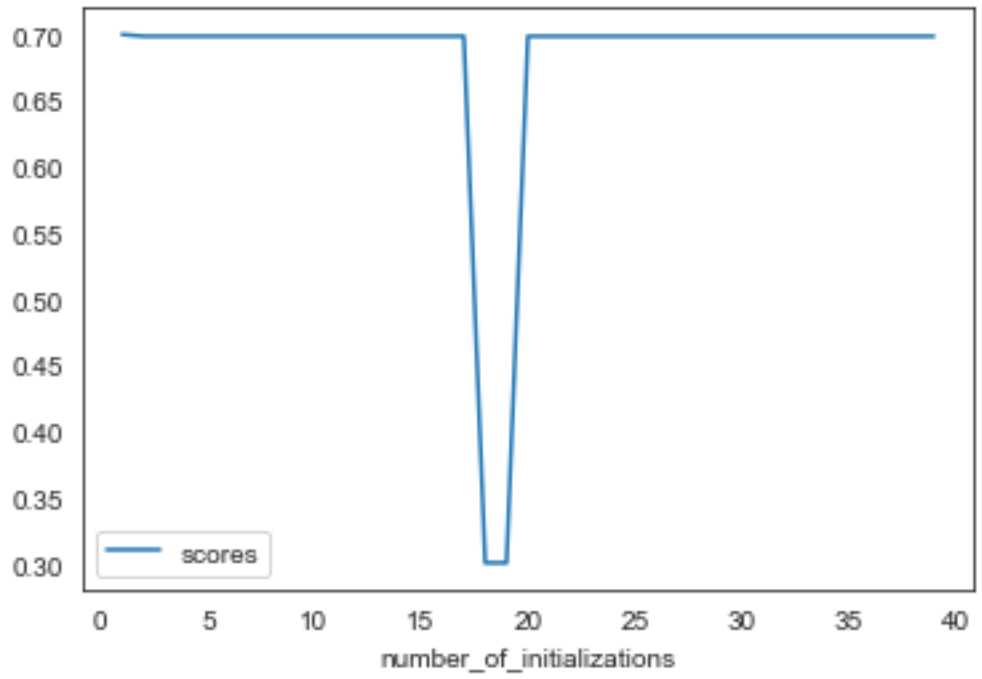

Figure 24 - Scores per number of initializations for K-means

In [144]:

```
max_ind = kmeans_df["scores"].idxmax()
kmeans_df.iloc[[max_ind]]
```

Table 7 - Optimized K-means initialization value and score

	number_of_initializations	scores
0	1	0.700316

```
# Accuracy stored

kmean_opt_score = kmeans_df.iloc[[max_ind]].scores.values[0]
```

2.4.2.1 Optimization for GMM clustering model

```
%%time
init_number = []
scores = []
for i in range(1,40):
    gmm = GaussianMixture(n_components = 2,
                    random_state= 3,
                    n_init= i)
    gmm.fit(df)
    y_gmm = gmm.predict(df)
    init_number.append(i)
    scr = metrics.accuracy_score(y_gmm, y_actual)
    scores.append(scr)

CPU times: total: 16min 20s
Wall time: 4min 50s
```

From table 8 the n_init parameter of 35 seemed to have the best accuracy of about 72.4%. From figure 25 accuracy was constang from 0 to 34 initializations but show a increase from 35 initializations onwards.

```
gmm_df = pd.DataFrame(zip(init_number,scores), columns = ['number_of_initiali
zations', 'scores'])

import matplotlib.pyplot as plt # Impot the relevant module


fig, ax = plt.subplots() # Create the figure and axes object


# Plot the first x and y axes:
gmm_df.plot(x = 'number_of_initializations', y = 'scores', ax = ax)
```

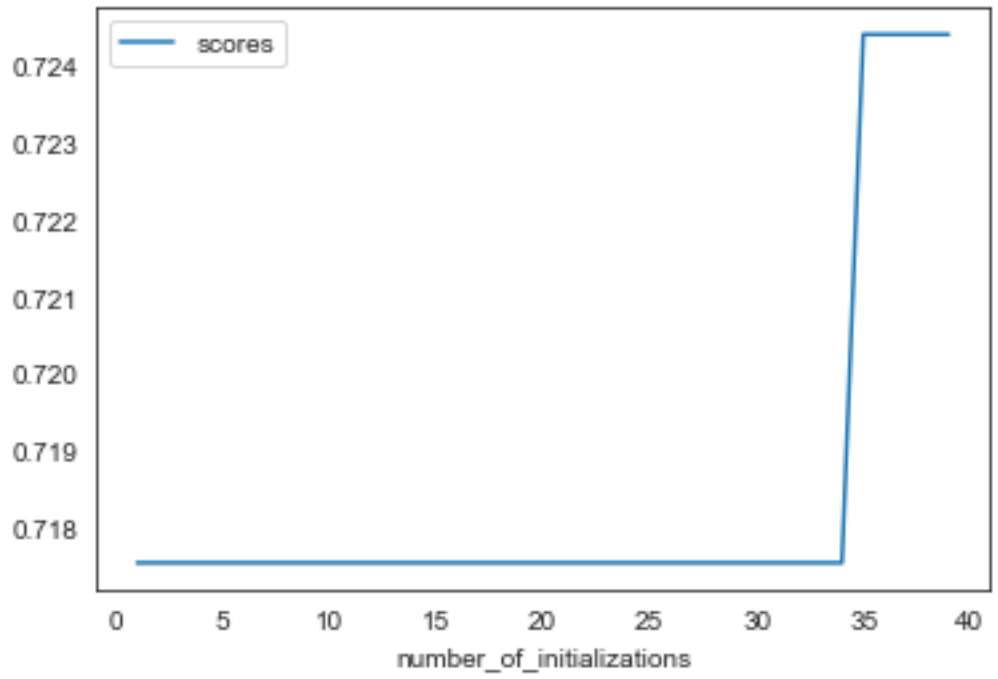

Figure 25 - Scores per number of initializations for GMM

```
max_ind = gmm_df["scores"].idxmax()
gmm_df.iloc[[max_ind]]
```

Table 8 - Optimized GMM initialization value and score

	number_of_initializations	scores
34	35	0.724394

```
# Accuracy stored

gmm_df_score = gmm_df.iloc[[max_ind]].scores.values[0]
```

2.4.3 Optimization for neural networks

2.4.3.1 Optimization Multi-layer Perceptron classifier

The hyperparameters selected for MLP classifier optimization were.

- **Hidden_layer_size:** This allows for the configuration of the number of hidden layers are well as the number of neurons in each layer. (150,) and (150,50) were used, the former implies a hidden layer with 150 neurons and the latter implies 2 hidden layers with 150 neurons in the first and 50 in the second.
- **Activation:** This allows for the configuration of the activation function for the hidden layers. 'tanh' and 'relu' were used. 'tanh' is a hyperbolic tan function which returns f(x) = tanh(x). relu', is a rectified linear unit function which returns f(x) = max (0, x)
- **solver:** This allows for the configuration of the solver for the weight optimization. The firsts solver configured was 'sgd', this refers to the stochastic gradient descent. The second solver configured was 'adam', this referes to a stochastic gradient-based optimizer proposed by a Kingma, Diederik, and Ba (2014).
- **alpha:** This is the strength of the L2 regularization term. The L2 regularization term is divided by the sample size when added to the loss. Continuous values from 0.0001 to 0.1 was utilized.
- **Learning_rate:** This is the Learning rate schedule for weight updates. The first learning rate was constant which as the name suggests is the constant learning rate provided by the default learning_rate_init parameter (0.001). The second is adaptive which keeps the learning rate constant to 'learning_rate_init' if training loss keeps decreasing. Each time two consecutive epochs fail to decrease training loss by at least 0.00001, the current learning rate is divided by 5(Akiba, et. al, 2019).

```
from sklearn.neural_network import MLPClassifier
mlp = MLPClassifier(random_state=76, max_iter=500)

mlp_parameters = {
```

```
    'hidden_layer_sizes' : [(150,), (150,50)],
    'activation': ['tanh', 'relu'],
    'solver': ['sgd', 'adam'],
    'alpha': [0.0001, 0.1],
    'learning_rate': ['constant','adaptive']
}

mlp_parameters
```

```
{'hidden_layer_sizes': [(150,), (150, 50)],
 'activation': ['tanh', 'relu'],
 'solver': ['sgd', 'adam'],
 'alpha': [0.0001, 0.1],
 'learning_rate': ['constant', 'adaptive']}
```

The `MLPClassifier` was set to 20 iterations with 3 cross validations.

```
from sklearn.model_selection import RandomizedSearchCV
mlp_opt_model = RandomizedSearchCV(estimator = mlp, param_distributions = mlp
_parameters,
                                   n_iter = 20,
                                   cv = 3,
                                   verbose=2,
                                   random_state=42,
                                   n_jobs = -1)
```

After optimization an accuracy of about 85.5% was obtained with the best parameters for solver, learning_rate, hidden_layer_sizes, alpha and activate being ‘adam’, constant, (150,50), 0.0001 and ‘tanh’ 50, 7 and 0.1 respectively. From figure 26 the optimized MLPClassifer gave a true negative of 6950, a false positive of 479, a false negative of 939 and a true positive of 1411.

```
# Fit the random search model
mlp_opt_model.fit(x_train, y_train)
```

```
mlp_opt_model.best_params_
```

```
{'solver': 'adam',
 'learning_rate': 'constant',
 'hidden_layer_sizes': (150, 50),
 'alpha': 0.0001,
 'activation': 'tanh'}
```

```
# Model Accuracy
from sklearn import metrics
y_pred = mlp_opt_model.predict(x_test)
accuracy_mlp_opt = metrics.accuracy_score(y_test, y_pred)
print('Accuracy:', accuracy_mlp_opt)
```

```
Accuracy: 0.8558706111167981
```

```
# classification report
from sklearn.metrics import classification_report, confusion_matrix
matrix = confusion_matrix(y_test,y_pred)
sns.heatmap(matrix, annot = True, fmt = 'd')
plt.title('confustion Matrix')
plt.xlabel('Predicted')
plt.ylabel('True')
print(classification_report(y_test,y_pred))
```

```
              precision    recall  f1-score   support

           0       0.88      0.94      0.91      7429
           1       0.75      0.60      0.67      2340
```

```
    accuracy                           0.86      9769
   macro avg       0.81      0.77      0.79      9769
weighted avg       0.85      0.86      0.85      9769
```

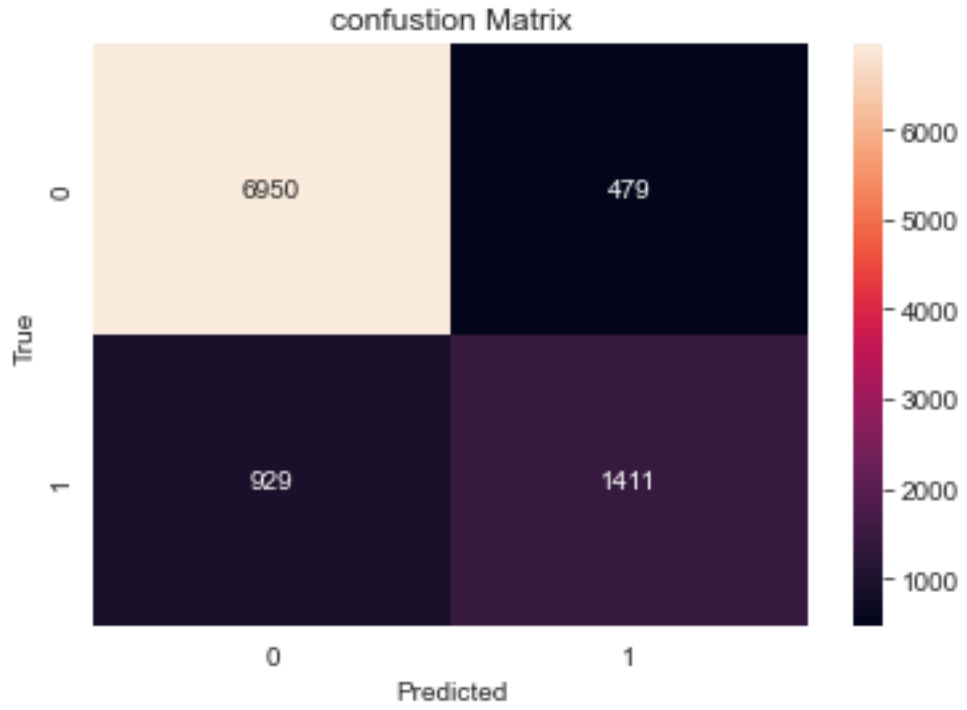

Figure 26 - Confusion matrix for optimized MLPClassifier

2.4.3.2 Optimization KerasClassifier

The hyperparameters selected for MLP classifier optimization were.

- **Neurons:** This is the number of neurons to be considered in each layer. 10, 30, 60, 100 and 200 neurons were configured.
- **Hidden_layers:** This is the number of hidden layers to be used in the model. 0, 1 and 2 were configured indicating no hidden layers, one and two hidden layers.
- **Dropout_rate:** This is the proportion of neurons to drop or zero out during the training phase where neurons are randomly selected. The fractional proportions of 0, 0.1, 0.2 and 0.4 were configured.
- **epochs:** This is the number of times the dataset is passed through the algorithm or trained with the algorithm. This is essential to enable alterations in weight calculations as the model is passed through on each cycle, to enable the model to approach an optimal fit of the data. The epochs used were 6, 7, 15 and 20
- **Learning_rate:** This is the Learning rate schedule for weight updates. This works in the same way as discussed under the MLPClassifer in section 2.4.3.1.
- **Init_mode**: This refers to how the neural network is initiated. The modes configured were 'uniform', 'lecun_uniform', 'normal', 'zero', 'glorot_normal','glorot_uniform', 'he_normal', and 'he_uniform'.
- **Activation:** This refers to the activation function for the hidden layers. 'relu' and 'tanh' were utilized, these were discussed in section 2.4.3.1. "Sigmoid" takes a real value as an input and outputs a value between 0 and 1. "SoftMax" on the other had outputs a multinomial probability distribution (Brownlee, 2016).

```
# we will do the grid search with KerasClassifier for first model
ann = KerasClassifier(build_fn=ANN1)
```

```
param_grid = {
    'neurons': [10, 30, 60, 100, 200],
    'hidden_layers': [0, 1, 2],
    'dropout_rate': [0, 0.1, 0.2, 0.4],
    'epochs': [6, 8, 15, 20],
    'learn_rate': [0.1, 0.03],
    'init_mode': ['uniform', 'lecun_uniform', 'normal', 'zero', 'glorot_norma
l',
                  'glorot_uniform', 'he_normal', 'he_uniform'],
    'activation': ['softmax','relu', 'tanh', 'sigmoid']}
```

The `KerasClassifer was set to 50 iterations with three-fold cross validation.`

```
keras_opt_model = RandomizedSearchCV(estimator = ann, param_distributions = p
aram_grid,
                                n_iter = 50,
                                cv = 3,
                                verbose=2,
                               random_state=76,
                                n_jobs = -1)
```

After optimization an accuracy of about 82.5% was obtained with the best parameters for neurons, learning_rate, init_mode, hidden_layers, epochs, dropout_rate and activation was 60, 0.1, `'lecun_uniform', 2, 15, 0 and ‘relu’.` From figure 27 the optimized KerasClassifer gave a true negative of 6208, a false positive of 1221, a false negative of 489 and a true positive of 1851.

```
# Fit the random search model
tf.random.set_seed(76)
keras_opt_model.fit(x_train, y_train)
```

```
# First parameter extraction
keras_opt_model.best_params_
```

```
{'neurons': 60,
 'learn_rate': 0.1,
 'init_mode': 'lecun_uniform',
 'hidden_layers': 2,
 'epochs': 15,
 'dropout_rate': 0,
 'activation': 'relu'}
```

```
# Model Accuracy
from sklearn import metrics
y_pred = keras_opt_model.predict(x_test)
accuracy_keras_opt1 = metrics.accuracy_score(y_test, y_pred)
print('Accuracy :', accuracy_keras_opt1)
```

```
306/306 [==============================] - 0s 798us/step
Accuracy : 0.8249564950353158
```

```
# classification report
from sklearn.metrics import classification_report, confusion_matrix
matrix = confusion_matrix(y_test,y_pred)
sns.heatmap(matrix, annot = True, fmt = 'd')
plt.title('confustion Matrix')
plt.xlabel('Predicted')
plt.ylabel('True')
print(classification_report(y_test,y_pred))
```

```
              precision    recall  f1-score   support

           0       0.93      0.84      0.88      7429
           1       0.60      0.79      0.68      2340
```

```
              accuracy                           0.82      9769
             macro avg       0.76      0.81      0.78      9769
          weighted avg       0.85      0.82      0.83      9769
```

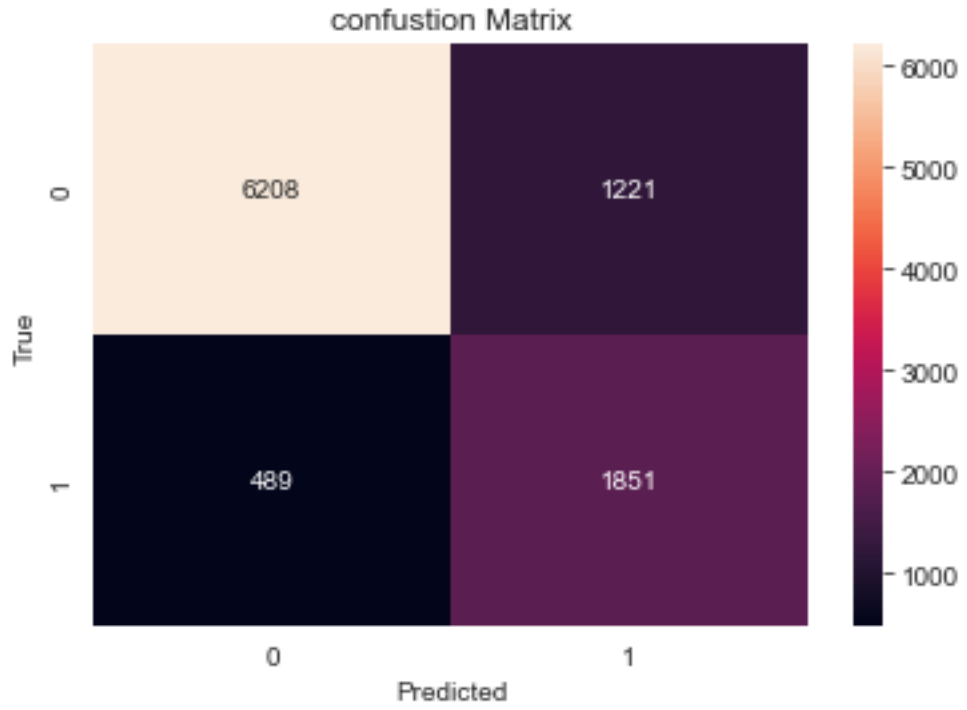

Figure 27 - Confusion matrix for optimized KerasClassifier

2.4.3.3 Optimizing best performing model with Genetic Algorithm.

With an accuracy of about 87.3% the Xgboost algorithm was the best performing model. To improve this accuracy the genetic algorithm was implemented. Unlike the random search cv which selects random parameters on each iteration, genetic algorithms follow and history with the best solutions for optimization passed down from one generation to the others.

A population of unique solutions is repeatedly modified by the genetic algorithm. The genetic algorithm chooses members of the present population to serve as parents at each stage and employs them to produce the offspring that will make up the following generation. The population "evolves" through generations toward an optimal outcome.

To produce the next generation from the existing population, the genetic algorithm applies one of three basic types of rules at each stage:

- **Selection** criteria choose the people, referred to as parents, who will contribute to the population of the following generation. Selection is typically stochastic and is subject to the relative performance of the candidates.
- **Crossover** laws combine two parents to create the next generation's offspring.
- To create children, **mutation** rules subject each parent to random modifications (Mirjalili,2 019).

Due to its ability to build on previous successes, genetic algorithms may result in better accuracy compared to the random search algorithm. For the purposes of classification algorithmic hyperparameter tunning the GASearchCV algorithm from sklearn was utilized to implement genetic algorithmic optimization.

As presented in section 2.4.1.3, the same hyperparameters and values were replicated to ensure overall consistency in measuring model improvement.

```
# !pip install sklearn-genetic-opt
from sklearn_genetic import GASearchCV
from sklearn_genetic.space import Continuous, Categorical, Integer
```

```
from sklearn_genetic.plots import plot_fitness_evolution, plot_search_space


xgb_param = {
 'max_depth': Integer(3, 10),
 'min_child_weight': Integer(1, 6),
 'gamma':  Continuous(0, 1),
 'subsample': Continuous(0.6, 1)}


xgb_param
{'max_depth': <sklearn_genetic.space.space.Integer at 0x1c22f220ac0>,
 'min_child_weight': <sklearn_genetic.space.space.Integer at 0x1c22f2202b0>,
 'gamma': <sklearn_genetic.space.space.Continuous at 0x1c22f2b2a00>,
 'subsample': <sklearn_genetic.space.space.Continuous at 0x1c22f2b2160>}
```

Three-fold cross-validation was configured. The population size was ten with ten generations. Three tournaments were undertaken to select the fittest candidates in the current population. Elitism was activated indicating that only the fitness handful of candidates are guaranteed a place in the next generation. The probability of a cross-over, that is the chance that two chromosomes exchange some of their parts was 80%. Mutation, which is the probability that random elements of the chromosome will be changed into some other value was 10%. The criteria under consideration were ‘max’ since the objective is to maximize accuracy. The simplest form of the evolutionary algorithm was implemented to limit heavy computation requirements. Finally, five of the best solutions were maintained across the generations.

```
xgb = XGBClassifier(random_state=76) # random state for repeatability

evolved_estimator = GASearchCV(estimator=xgb,
                               cv=3, # 3 fold cross validation
                               scoring='accuracy', # metric is accuracy
                               population_size = 10, # population size of 10
                               generations=10, # 10 generations
                               tournament_size=3,# tournament size of 3
                               elitism=True, # elitism is activated
                               crossover_probability = 0.8,
                               mutation_probability = 0.1,
                               param_grid = xgb_param,
                               criteria='max', # maximize
                               algorithm='eaSimple', # simple algorithm
                               n_jobs = -1, # all cpu
                               verbose = True, # show process
                               keep_top_k = 5) # top 5 solutions
```

```
evolved_estimator.fit(x_train, y_train)
```

gen	nevals	fitness	fitness_std	fitness_max	fitness_min
0	10	0.860341	0.00373722	0.867717	0.854686
1	4	0.863334	0.00279048	0.867717	0.859205
2	4	0.865567	0.00204514	0.867717	0.862496
3	4	0.86565	0.00197654	0.867717	0.862496
4	4	0.866282	0.0013788	0.867717	0.86368
5	2	0.866752	0.000854359	0.867717	0.865348
6	4	0.866892	0.000912914	0.867892	0.865348
7	4	0.86605	0.00255797	0.867892	0.859161
8	4	0.867287	0.000789182	0.867892	0.865874
9	4	0.86734	0.000821307	0.867892	0.865874
10	5	0.867208	0.000827609	0.867892	0.865874

Form the evolved estimator the best parameters for max_dept, min_child_weight, gamma and subsample were 3, 4, 0.3314799614609252 and 0.8447434115569709. An accuracy of about 87.5% was also achieved.

```
evolved_estimator.best_params_

{'max_depth': 3,
 'min_child_weight': 4,
 'gamma': 0.3314799614609252
 'subsample': 0.8447434115569709}
```

```
from sklearn import metrics
# Model Accuracy
y_pred = evolved_estimator.predict(x_test)

accuracy_gen_opt = metrics.accuracy_score(y_test, y_pred)

print ('Accuracy:', accuracy_gen_opt)

Accuracy: 0.8746033370867028
```

2.4.4 Comparison of models and improvements from optimization.

2.4.4.1 Comparison of classification models and improvements from optimization

Table 9 illustrates the pre and post-optimization results of the classification models used. Although XGboost had the lowest percentage improvement of 0.2%, it had the best overall accuracy of 87.3%. Gradient boost machine had the best performance improvement from with an increase of 0.7% from 86.5% to 87.2%. The worst performing model was the Random Forest with an accuracy of 86.6% and a performance improvement of 0.6%.

```
dat_clf = {"Classification_model" : ["Random forest", "Gradient Boost Machine
", "XGboost"],
       "Pre_optimization_accuracy" : [accuracy_rf,accuracy_gbm,accuracy_xgb],
       "Post_optimization_accuracy" : [accuracy_rf_opt,accuracy_gbm_opt,accur
acy_xgb_opt],
          "Improvement %" : [((accuracy_rf_opt-accuracy_rf)/accuracy_rf)*100,
                             ((accuracy_gbm_opt-accuracy_gbm)/accuracy_gbm)*100
,
                             ((accuracy_xgb_opt-accuracy_xgb)/accuracy_xgb)*100
]}

df_clf = pd.DataFrame(dat_clf)
df_clf.round(3)
```

Table 9 - Comparison of classification models and improvements from optimization

	Classification_model	Pre_optimization_accuracy	Post_optimization_accuracy	Improvement %
0	Random forest	0.861	0.866	0.619
1	Gradient Boost Machine	0.865	0.872	0.733
2	XGboost	0.871	0.873	0.200

2.4.4.2 Comparison of clustering models and improvements from optimization

Table 10 illustrates the pre and post-optimization results of the clustering models used. The K-means algorithm has an improvement in accuracy and optimized accuracy value of about 0.2% and 70% respectively. GMM on the other hand had the best optimized accuracy of 72.4% and a larger improvement in accuracy of 0.95%

```
dat_clus = {"Clustering_model" : ["Kmeans", "GMM"],
       "Pre_optimization_accuracy" : [kmeans_score,gmm_score],
       "Post_optimization_accuracy" : [kmean_opt_score,gmm_df_score],
          "Improvement %" : [((kmean_opt_score-kmeans_score)/kmeans_score)*10
0,
                             ((gmm_df_score-gmm_score)/gmm_score)*100 ]}

df_clus = pd.DataFrame(dat_clus)
df_clus.round(3)
```

Table 10 - Comparison of clustering models and improvements from optimization

	Clustering_model	Pre_optimization_accuracy	Post_optimization_accuracy	Improvement %
0	Kmeans	0.699	0.700	0.193
1	GMM	0.718	0.724	0.954

2.4.4.3 Comparison of neural network models and improvements from optimization

Table 11 illustrates the pre and post-optimization results of the neural network models used. The MLPClassifer by sklearn was the best performing model with an optimized accuracy of 85.6% and an improvement of about 0.69%. The KerasClassifer on the other hand had the lower optimization accuracy of 82.5% with an improvement of about 0.5%.

```
dat_nn = {"Neural_network_model" : ["MLP", "KerasClassifer"],
       "Pre_optimization_accuracy" : [accuracy_mlp, accuracy_ann],
       "Post_optimization_accuracy" : [accuracy_mlp_opt, accuracy_keras_opt1]
,
          "Improvement %" : [((accuracy_mlp_opt-accuracy_mlp)/accuracy_mlp)*1
00,
                             ((accuracy_keras_opt1-accuracy_ann)/accuracy_ann)*
100 ]}

df_nn = pd.DataFrame(dat_nn)
df_nn.round(3)
```

Table 11 - Comparison of neural network models and improvements from optimization

	Neural_network_model	Pre_optimization_accuracy	Post_optimization_accuracy	Imporvement %
0	MLP	0.850	0.856	0.686
1	KerasClassifer	0.821	0.825	0.499

2.4.4.4 Comparison of optimization algorithms for best performing model.

Table 12 illustrates the improved performance of the best performing model (XGBoost) by applying genetic algorithmic optimization. Optimization with the random search algorithm produced an accuracy of 87.3%. Through optimization with genetic algorithm a higher accuracy of 87.5% was achieved showcasing an improvement of about 0.16%.

```
dat_gen = {"Best_model" : "XGboost",

       "RandomSearchCV" : [accuracy_mlp_opt],

       "Genetic_Algorithm" : [accuracy_gen_opt],

        "Improvement %" : [((accuracy_gen_opt-accuracy_mlp_opt)/accuracy_mlp_
opt)*100]}



df_gen = pd.DataFrame(dat_gen)

df_gen.round(3)
```

Table 12 - Comparison of optimization algorithms for best performing model.

	Best_model	Random Search Algorithm	Genetic Algorithm	Improvement %
0	XGboost	0.873	0.875	0.164

2.5 Conclusion

To undertake the classification modelling process an appropriate method must be employed to handle categorical variables and convert categories into a machine-readable format. In the presence of high categories, label encoding may prove more feasible to reduce the overall dimensionality of the dataset and prevent the 'curse of dimensionality'.

Hyperparameter tuning was demonstrated to improve the overall accuracy of clustering, classification, and artificial neural network models. This tuning may be achieved with resource-efficient optimization algorithms without the need of calculating all possible combinations of parameters.

Finally, through the application of genetic algorithmic optimization for the XGboost classification model, the study was able to predict whether an adult makes more USD 50,000 in the dataset with an accuracy of 87.5%

Reference List

Akiba, T., Sano, S., Yanase, T., Ohta, T. and Koyama, M., 2019, July. Optuna: A next-generation hyperparameter optimization framework. In Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining (pp. 2623-2631).

Allwein, E.L., Schapire, R.E. and Singer, Y., 2000. Reducing multiclass to binary: A unifying approach for margin classifiers. Journal of machine learning research, 1(Dec), pp.113-141.

Barford, A., Dorling, D., Smith, G.D. and Shaw, M., 2006. Life expectancy: women now on top everywhere. Bmj, 332(7545), p.808.

Brownlee, J., 2016. How to grid search hyperparameters for deep learning models in python with keras. línea]. Disponible en: https://machinelearningmastery. com/grid-search-hyperparameters-deep-learning-models-python-keras.

Chen, T. and Guestrin, C., 2016, August. Xgboost: A scalable tree boosting system. In Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining (pp. 785-794).

Freund, R.J., Wilson, W.J. and Sa, P., 2006. Regression analysis. Elsevier.

Fränti, P. and Sieranoja, S., 2019. How much can k-means be improved by using better initialization and repeats?. Pattern Recognition, 93, pp.95-112.

Gentleman, R. and Carey, V.J., 2008. Unsupervised machine learning. In Bioconductor case studies (pp. 137-157). Springer, New York, NY.

Hall, M.E., do Carmo, J.M., da Silva, A.A., Juncos, L.A., Wang, Z. and Hall, J.E., 2014. Obesity, hypertension, and chronic kidney disease. International journal of nephrology and renovascular disease, 7, p.75.

Hays, J.T., Dale, L.C., Hurt, R.D. and Croghan, I.T., 1998. Trends in smoking-related diseases: why smoking cessation is still the best medicine. Postgraduate Medicine, 104(6), pp.56-71.

Jiang, T., Gradus, J.L. and Rosellini, A.J., 2020. Supervised machine learning: a brief primer. Behavior Therapy, 51(5), pp.675-687.

Kang, P. and Cho, S., 2009, September. K-means clustering seeds initialization based on centrality, sparsity, and isotropy. In International Conference on Intelligent Data Engineering and Automated Learning (pp. 109-117). Springer, Berlin, Heidelberg.

Kingma, D.P. and Ba, J., 2014. Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980.

Köppen, M., 2000, September. The curse of dimensionality. In 5th online world conference on soft computing in industrial applications (WSC5) (Vol. 1, pp. 4-8).

Li, Y. and Wu, H., 2012. A clustering method based on K-means algorithm. Physics Procedia, 25, pp.1104-1109.

Martins, P.N., Pratschke, J., Pascher, A., Fritsche, L., Frei, U., Neuhaus, P. and Tullius, S.G., 2005. Age and immune response in organ transplantation. Transplantation, 79(2), pp.127-132.

Miles, J., 2005. R-squared, adjusted R-squared. Encyclopedia of statistics in behavioral science.

Mirjalili, S., 2019. Genetic algorithm. In Evolutionary algorithms and neural networks (pp. 43-55). Springer, Cham.

Morgan, J.A. and Tatar, J.F., 1972. Calculation of the residual sum of squares for all possible regressions. Technometrics, 14(2), pp.317-325.

Natekin, A. and Knoll, A., 2013. Gradient boosting machines, a tutorial. Frontiers in neurorobotics, 7, p.21.

Probst, P., Wright, M.N. and Boulesteix, A.L., 2019. Hyperparameters and tuning strategies for random forest. Wiley Interdisciplinary Reviews: data mining and knowledge discovery, 9(3), p.e1301.

Putatunda, S. and Rama, K., 2018, November. A comparative analysis of hyperopt as against other approaches for hyper-parameter optimization of XGBoost. In Proceedings of the 2018 International Conference on Signal Processing and Machine Learning (pp. 6-10).

Putatunda, S. and Rama, K., 2019, December. A modified bayesian optimization based hyper-parameter tuning approach for extreme gradient boosting. In 2019 Fifteenth International Conference on Information Processing (ICINPRO) (pp. 1-6). IEEE.

Quinlan, J.R., 1996. Learning decision tree classifiers. ACM Computing Surveys (CSUR), 28(1), pp.71-72.

Rigatti, S.J., 2017. Random forest. Journal of Insurance Medicine, 47(1), pp.31-39.

Vlassoff, C., 2007. Gender differences in determinants and consequences of health and illness. Journal of health, population, and nutrition, 25(1), p.47.

Yang, M.S., Lai, C.Y. and Lin, C.Y., 2012. A robust EM clustering algorithm for Gaussian mixture models. Pattern Recognition, 45(11), pp.3950-3961.

Yegnanarayana, B., 2009. Artificial neural networks. PHI Learning Pvt. Ltd..

Zheng, A. and Casari, A., 2018. Feature engineering for machine learning: principles and techniques for data scientists. " O'Reilly Media, Inc.".

Zhu, X. and Goldberg, A.B., 2009. Introduction to semi-supervised learning. Synthesis lectures on artificial intelligence and machine learning, 3(1), pp.1-130.