



# Term Deposit Prediction

Rofiah Adeshina

## ***Outline***

- Introduction
- Exploratory Data Analysis
- Data Preprocessing
- Building the model and Results
- Conclusion
- References

## Introduction

Predictive models are now used in all industries to identify and predict important metrics. In this scenario, a bank is being considered. The bank collected a huge amount of data that includes profiles of those customers who have to subscribe to term deposits and those who did not subscribe to a term deposit. The task is to come up with a robust predictive model that would help them identify customers who would or would not subscribe to their term deposits in the future.

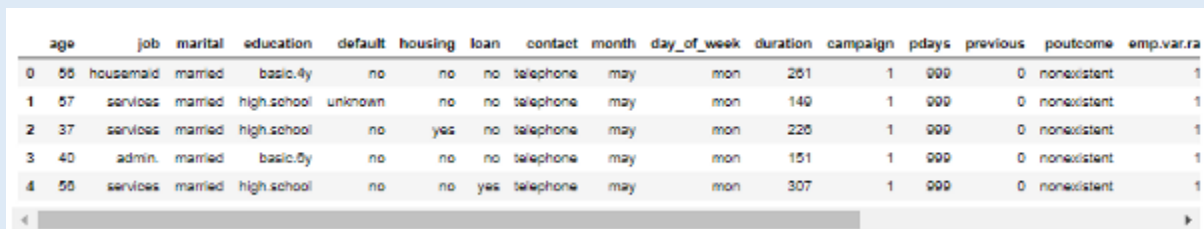
The goal is to carry out data exploration, data cleaning, feature extraction, and developing robust machine learning algorithms that would aid them in the department.

## Exploratory Data Analysis

The data is loaded as shown below

```
df = pd.read_csv(r'C:\Users\HP\Desktop\CV, P.Statement and others\10 Academy\week 6\bank-additional\bank-additional\bank-additional-full.csv', sep=';')
df.head()
```

The data highlights profiles of customers who have or haven't subscribed to the bank's term deposit. The data used is the [bank additional full CSV file](#), with 41188 rows and 21 columns. Full details including column description can be gotten from the link above. An overview of the loaded data is shown below:



	age	job	marital	education	default	housing	loan	contact	month	day_of_week	duration	campaign	pdays	previous	poutcome	emp.var.ra
0	55	housemaid	married	basic.4y	no	no	no	telephone	may	mon	261	1	999	0	nonexistent	1
1	57	services	married	high.school	unknown	no	no	telephone	may	mon	140	1	999	0	nonexistent	1
2	37	services	married	high.school	no	yes	no	telephone	may	mon	226	1	999	0	nonexistent	1
3	40	admin.	married	basic.5y	no	no	no	telephone	may	mon	151	1	999	0	nonexistent	1
4	55	services	married	high.school	no	no	yes	telephone	may	mon	307	1	999	0	nonexistent	1

fig 1: Data Overview

The normal practice of exploratory data analysis is adopted here. By using the .info() function on the data frame, it is seen that the data has no missing values with 11 categorical columns. Also, a statistical description of the data is shown below.

```
df.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 41188 entries, 0 to 41187
Data columns (total 21 columns):
age                41188 non-null int64
job                41188 non-null object
marital            41188 non-null object
education          41188 non-null object
default            41188 non-null object
housing            41188 non-null object
loan               41188 non-null object
contact            41188 non-null object
month              41188 non-null object
day_of_week        41188 non-null object
duration           41188 non-null int64
campaign           41188 non-null int64
pdays             41188 non-null int64
previous           41188 non-null int64
poutcome           41188 non-null object
emp.var.rate       41188 non-null float64
cons.price.idx     41188 non-null float64
cons.conf.idx      41188 non-null float64
euribor3m          41188 non-null float64
nr.employed        41188 non-null float64
y                  41188 non-null object
dtypes: float64(5), int64(5), object(11)
memory usage: 6.6+ MB
```

fig 2: Data Info

```
df.describe()
```

	age	duration	campaign	pdays	previous	emp.var.rate	cons.price.idx	cons.conf.idx	euribor3m	nr.employed
count	41188.00000	41188.00000	41188.00000	41188.00000	41188.00000	41188.00000	41188.00000	41188.00000	41188.00000	41188.00000
mean	40.02408	258.285010	2.587593	982.476454	0.172983	0.081886	93.575884	-40.502600	3.821291	5187.035911
std	10.42125	259.279249	2.770014	189.910907	0.494901	1.570980	0.578840	4.628198	1.734447	72.251528
min	17.00000	0.000000	1.000000	0.000000	0.000000	-3.400000	92.201000	-50.800000	0.834000	4083.800000
25%	32.00000	102.000000	1.000000	999.000000	0.000000	-1.600000	93.075000	-42.700000	1.344000	5089.100000
50%	38.00000	180.000000	2.000000	999.000000	0.000000	1.100000	93.749000	-41.800000	4.857000	5191.000000
75%	47.00000	319.000000	3.000000	999.000000	0.000000	1.400000	93.994000	-38.400000	4.981000	5228.100000
max	98.00000	4918.000000	56.000000	999.000000	7.000000	1.400000	94.787000	-28.900000	5.045000	5228.100000

fig 3: Statistical Description of Data

## Univariate analysis

```
#Univariate Analysis
def univariate_plot(column, data):
    plot = sns.catplot(x=column, data=data, kind='count')
    plot.set_xticklabels(rotation=45)
    plt.show()
    return plot
univariate_plot('y', df)
univariate_plot('job', df)
univariate_plot('education', df)
univariate_plot('marital', df)
univariate_plot('loan', df)
univariate_plot('housing', df)
```

Univariate analysis of all categorical columns shows the diverse group of customers the bank has. The categorical column “y” represents the client’s response (yes/no) to the term subscription. It is the target variable for the predictive analysis and shows a class imbalance.

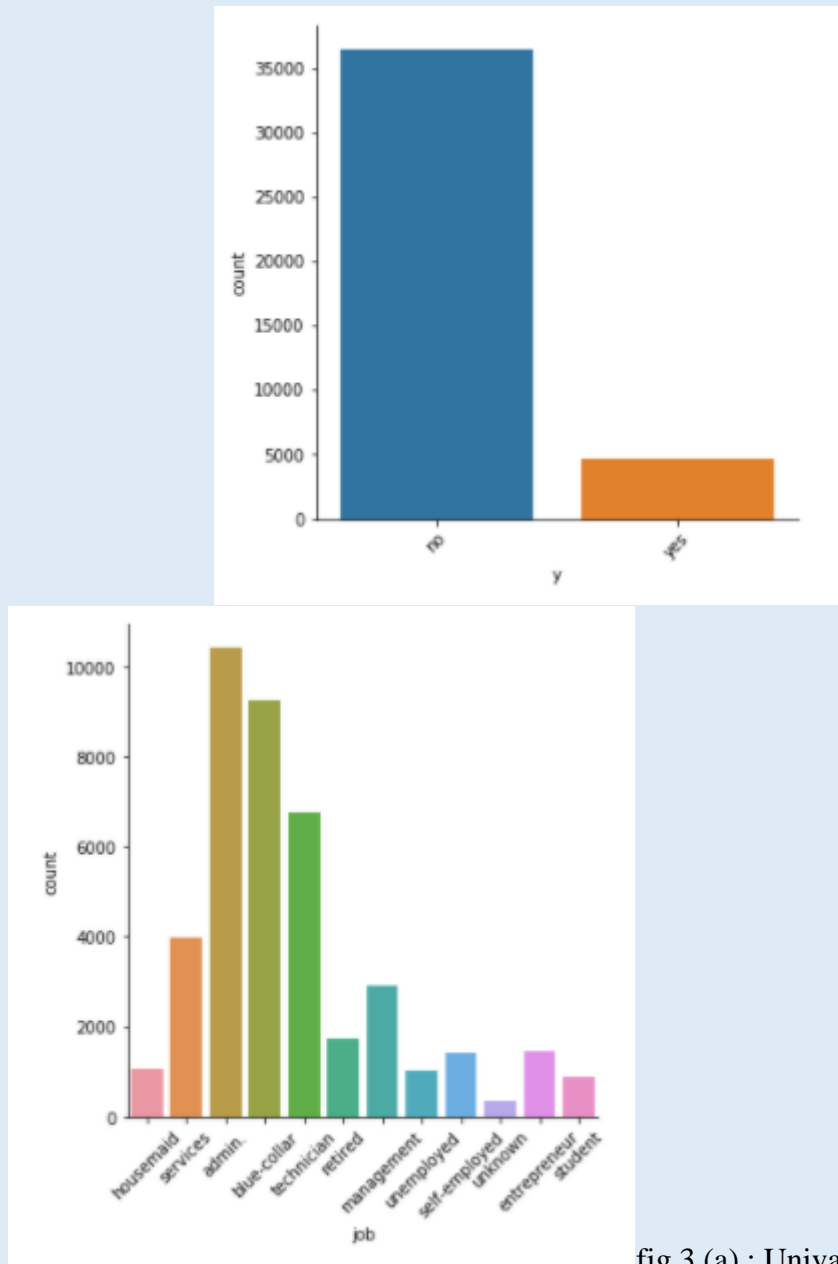


fig 3 (a) : Univariate Count Plots

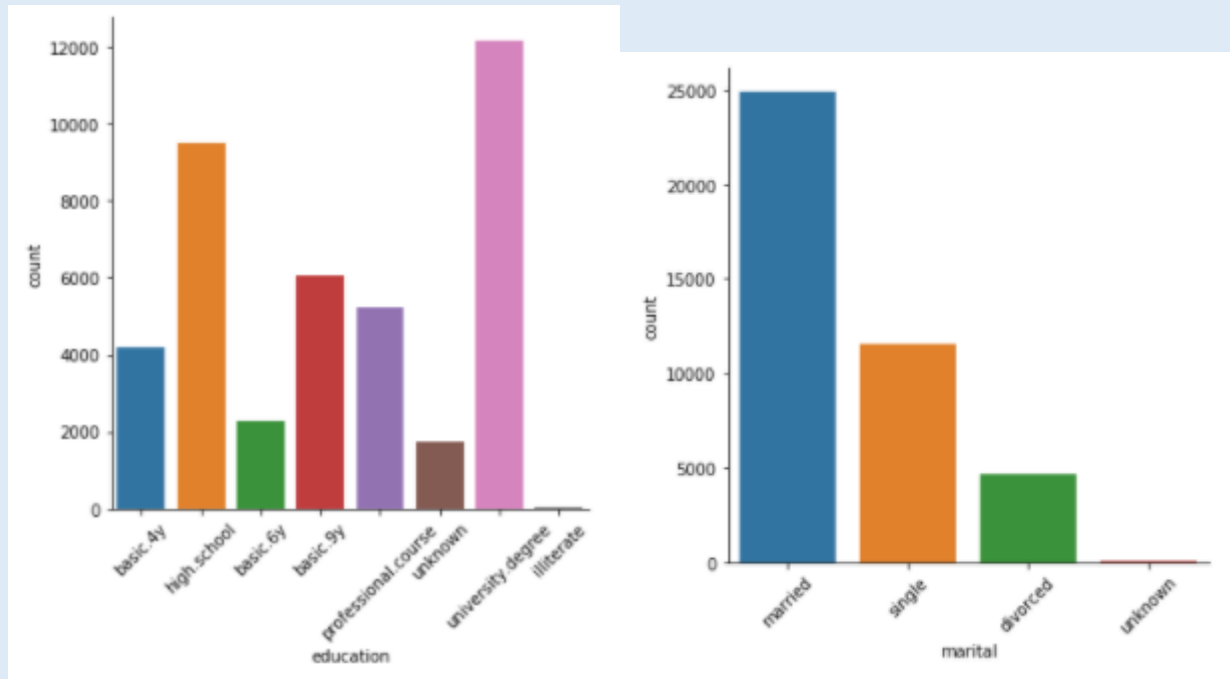


fig 3 (b) : Univariate Count Plots

```
# Distribution plots
def distribution_plot(col, col_distribution):
    plot = sns.distplot(df[col], kde=False, color='red', bins=10)
    plt.title(col_distribution, fontsize=18)
    plt.xlabel(col, fontsize=16)
    plt.ylabel('Frequency', fontsize=16)
    plt.show()
    return plot
distribution_plot('age', 'Age distribution')
distribution_plot('cons.price.idx', 'Price distribution')
```

As part of the univariate analysis, the distribution of the numerical columns is also shown

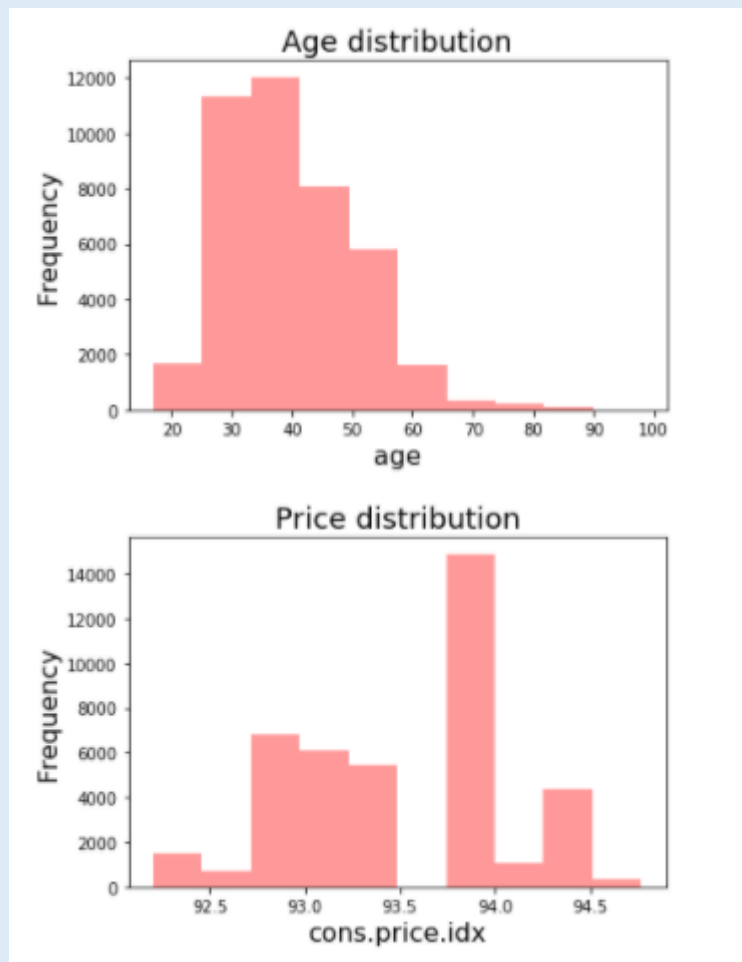


fig 4: Numerical Column Distribution

### ***Bivariate Analysis***

This shows how the features relate to one another

```
#bivariate Analysis
def bivariate_plot(cat, num, data, hue):
    plot = sns.catplot(x=cat, y=num, data=data, kind='box', hue=hue)
    plot.set_xticklabels(rotation=45)
    plt.show()
    return plot
bivariate_plot('marital', 'age', df, 'y')
bivariate_plot('job', 'age', df, 'y')
```

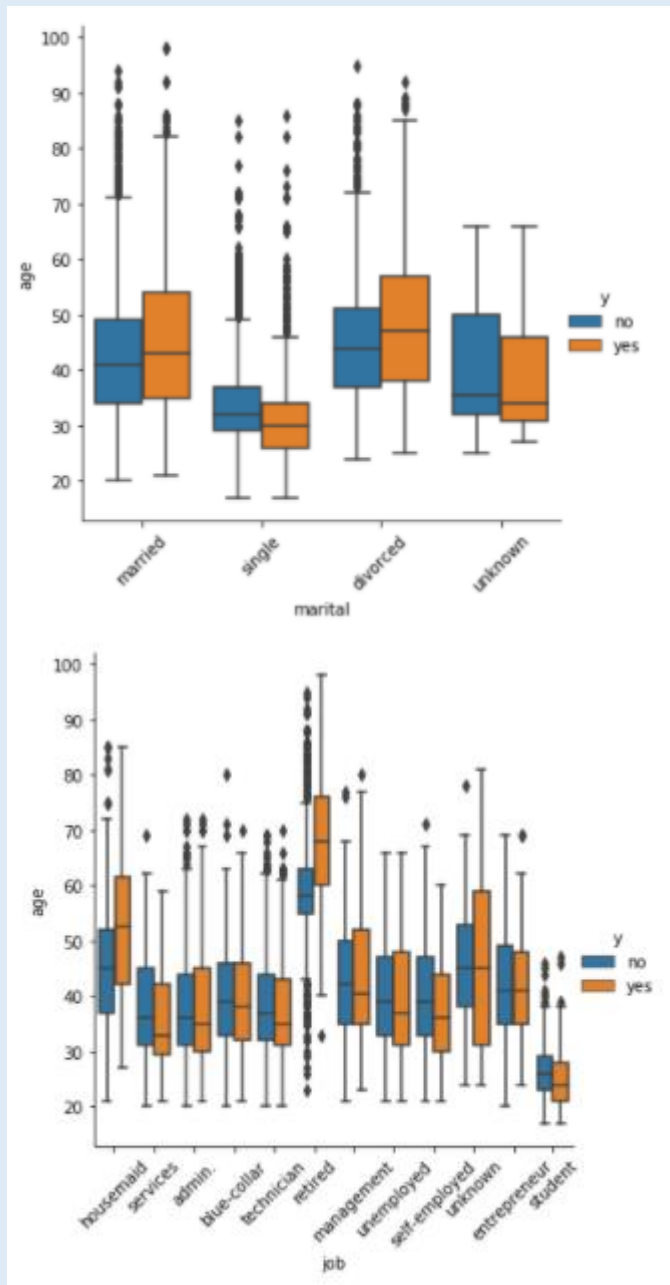


fig 5: Bi-variate Box Plots

From the univariate analysis, it is seen that there is a class imbalance in the target variable “y” with our main target “yes” being the minority class. Feeding this imbalanced data to the classifier model can make it biased in favor of the majority class “no”, simply because it did not have enough data to learn about the minority. It is therefore essential to balance the classes before feeding it to the model.

### ***Data Preprocessing***

This is data preparation for modeling and machine learning. The feature to be predicted (“target/dependent” feature) is the response to the term deposit subscription (“y” column). The



other features that we use for the prediction are called the “descriptive/independent” features. There is a certain format for the data before we can perform modeling using `Scikit-Learn`. The following steps are followed for data preparation:

1. Outliers and unusual values (such as a negative age) are taken care of: they are replaced with mean.
2. Any categorical descriptive feature is encoded to be numeric as follows: one-hot-encoding, as this is a classification problem, the target feature is label-encoded. (in case of a binary problem, the positive class is encoded as 1).
3. All descriptive features (which are all numeric at this point) are scaled.
4. Splitting the data into test and train data set (90% train and 10% validation)
5. Dealing with the class imbalance in the target feature

### *1. Taking Care of Outliers*

To deal with outliers the automatic outliers detection Isolate forest was used, however, it dropped over 4000 rows of the data set which is way too much. As an alternative, feature importance can be done and outliers of the most important features are dealt with. The function below generates box plots that help visualize these features that contain outliers.

```
#boxplots for outlier detection
def outlier(col, col_distribution):
    chart = sns.boxplot(x=df[col])
    plt.title(col_distribution, fontsize=18)
    plt.xlabel(col, fontsize=16)
    plt.show()
```

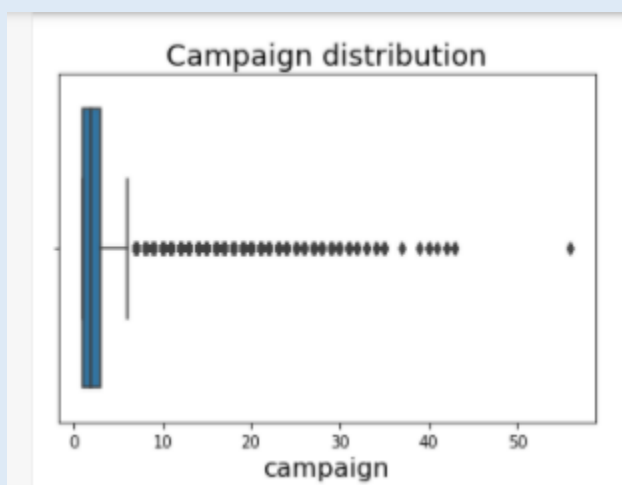


fig 7(a): Campaign Outlier Distribution

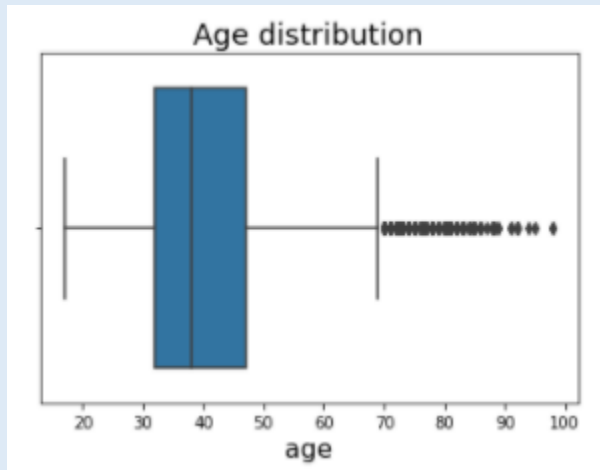


fig 7(b): Age Outlier Distribution

The function below detects the outliers and replaces them with the median.

```
#Treating the outliers
def replace_outlier_with_median(dataFrame, feature):
    """ a function for replacing outliers with the median, used when there's
    too many outliers in a feature"""
    Q1 = dataFrame[feature].quantile(0.25)
    Q3 = dataFrame[feature].quantile(0.75)
    median = dataFrame[feature].quantile(0.50)
    IQR = Q3 - Q1
    upper_whisker = Q3 + (1.5 * IQR)
    lower_whisker = Q1 - (1.5 * IQR)
    dataFrame[feature] = np.where(dataFrame[feature] > upper_whisker, median,
    dataFrame[feature])
    dataFrame[feature] = np.where(dataFrame[feature] < lower_whisker,
    median, dataFrame[feature])
    # replace in "age" and "campaign" with median
    replace_outlier_with_median(df, 'age')
    replace_outlier_with_median(df, 'campaign')
```

## 2. Encoding Categorical features

The categorical columns in that are descriptive features are one hot encoded, while that of the target feature is label encoded.

```
# create an object of the OneHotEncoder
OHE =
ce.OneHotEncoder(cols=['job', 'marital', 'education', 'default', 'housing', '
loan', 'contact', 'month', 'day_of_week', 'poutcome'], use_cat_names=True)
# encode the categorical variables
df = OHE.fit_transform(df)
df.head()
```

```
#label encoding target
# creating instance of labelencoder
lbe = LabelEncoder()
# Assigning numerical values and storing in another column
target = lbe.fit_transform(df.y)
target = pd.DataFrame(target, columns = ['y'])
target.head()
```

### ***3. Scaling data***

Once all categorical descriptive features are encoded, all features in this transformed dataset will be numerical. It is always a good idea to scale these numerical descriptive features before fitting the model, as scaling is mandatory for some models especially those that consider Euclidean distance. The main idea why this is done is that some variables are often measured at different scales and would not contribute equally to model fitting and this may lead the trained model to create some bias. Here the numerical columns from the data frame are scaled

```
# data standardization with sklearn
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
num_cols = ['emp.var.rate', 'cons.price.idx', 'cons.conf.idx',
            'euribor3m', 'nr.employed', 'age', 'pdays', 'campaign', 'previous']
Feature[num_cols].head()
Feature[num_cols] = scaler.fit_transform(Feature[num_cols])
Feature.head()
```

### ***4. Split data into test and train***

The data is split in the ratio 9:1 using 90% for training and 10% for validation.

```
#splitting into test and train
X_train, X_test, y_train, y_test = train_test_split(Feature, target,
                                                    test_size=0.10, random_state=15)
print ("Training and testing split was successful.")
X_train.head()
```

### ***5. Class Imbalance***

In classification problems, balanced data is very important. Data is said to be imbalanced when instances of one class outnumber the other(s) by a large proportion as it is seen in the target feature “y”.

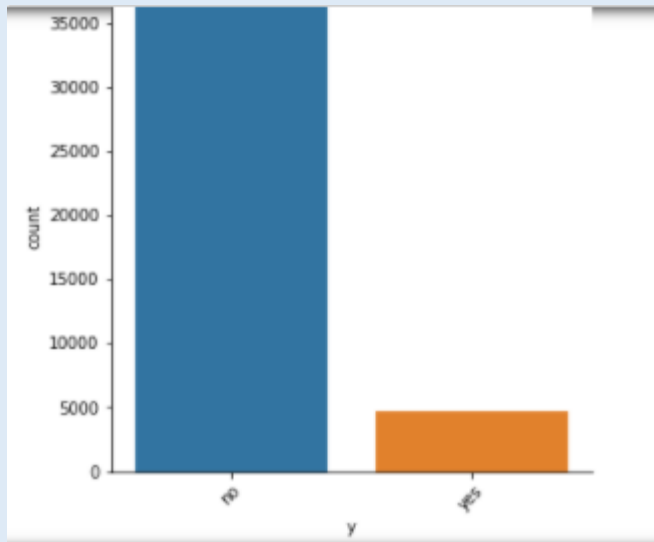


fig 8: Term Deposit Class Imbalance

Feeding imbalanced data to your classifier can make it biased in favor of the majority class, simply because it did not have enough data to learn about the minority. To deal with imbalance the oversampling method SMOTE (**S**ynthetic **M**inority **O**ver-Sampling **T**echnique) is used to generate synthetic data for the minority class.

```
#dealing with class imbalance
from imblearn.over_sampling import SMOTE
sm = SMOTE(random_state = 33)
X_train_new, y_train_new = sm.fit_sample(X_train,
y_train.values.ravel())
y_train_n = pd.DataFrame(y_train_new, columns = ["Y"])
pd.Series(y_train_new).value_counts().plot.bar()
```

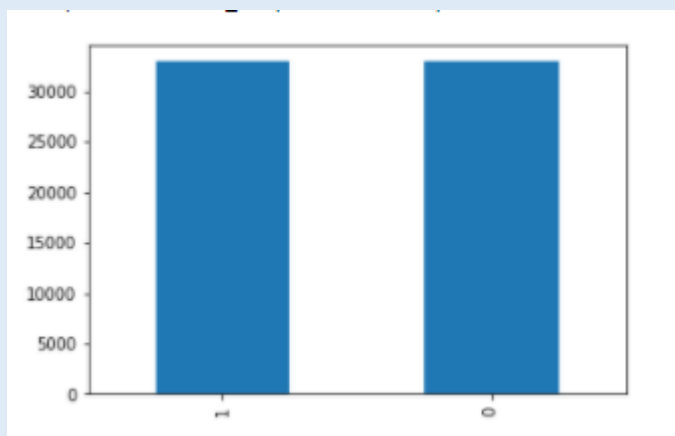


fig 9: Output after SMOTE

***Building the model***

1. Fitting the model with k-fold cross-validation using various machine learning classifier algorithms. Determine model accuracy.
2. Prediction with validation set.

### ***1. Fitting the model with k-fold cross-validation using various machine learning classifier algorithms***

Having split the data into test and train to fit and validate the model, K-fold cross-validation is adopted to further improve model evaluation. However, the stratified version is used as the k-fold cross-validation is not appropriate for evaluating imbalanced classifiers.

The reason being that the data is split into  $k$ -folds with a uniform probability distribution. This might work fine for data with a balanced class distribution, however, when the distribution is severely skewed (class imbalance), it is likely that one or more folds will have few or no examples from the minority class. This means that some or perhaps many of the model evaluations will be misleading, as the model need only predict the majority class correctly. The function below is used to implement stratified k-fold cross validation

```
from sklearn.model_selection import StratifiedKFold
def model_predictor(model, x, y):
    scores = []
    kf = StratifiedKFold(n_splits=10, shuffle=True, random_state=1)
    for train_index, test_index in kf.split(X_train, y_train):
        KX_train, KX_test = X_train.iloc[train_index], X_train.iloc[test_index]
        Ky_train, Ky_test = y_train.iloc[train_index], y_train.iloc[test_index]
        trained_model = model.fit(KX_train, Ky_train)
        scores.append(model.score(X = KX_test ,y = Ky_test))
    return trained_model, scores
```

Three different machine learning algorithms were used to build predictive models and the accuracy for each considered. The logistic regression, random forest, and XGBoost machine learning algorithms were used.

- **Logistic Regression**

```
# Initialize logistic regression model
from sklearn.linear_model import LogisticRegression
log_model = LogisticRegression()
#Initialize decision tree
from sklearn.ensemble import RandomForestClassifier
forest_model = RandomForestClassifier(n_estimators=10,
    criterion='entropy',random_state=0)
#initialize XGBoost
import xgboost as xgb
XGB_model = xgb.XGBClassifier()
```

```
#fit models, predict and determine scores
log_model_trained_model, log_model_scores, = model_predictor(log_model,
X_train_new, y_train_n)
print("Accuracy of the model is" + ":" +str(np.mean(log_model_scores)))
print("std of scores computed" + ":" +str(np.std(log_model_scores)))
#make predictions with validation set
y_pred_log = pd.DataFrame(log_model.predict(X_test), columns=["Term Deposit
Predictions"])
df_log = pd.concat([y_test,y_pred_log], axis=1)
df_log.head()
#confusion matrix
from sklearn.metrics import confusion_matrix
from sklearn.metrics import classification_report
confusion_matrix = confusion_matrix(y_test, y_pred_log)
confusion_matrix
```

The Logistic model accuracy was 0.9008874052983294 (90% accuracy) accurate with a confusion matrix

```
array([[3559,    65],
       [ 379,   116]])
precision    recall  f1-score   support

           0       0.90      0.98      0.94      3624
           1       0.64      0.24      0.35       495

 accuracy                   0.89      4119
 macro avg              0.77      0.61      0.64      4119
weighted avg              0.87      0.89      0.87      4119
```

Though the accuracy of the model is pretty high, the confusion matrix precision shows that the major target “yes” is only predicted correctly 64% of the time.

- **Random Forest Classifier**

This algorithm is used to train the model trying to determine if model performance improves.

```
#fit models, predict and determine scores
forest_model_trained_model, forest_model_scores =
model_predictor(forest_model, X_train_new, y_train_n.values.ravel())
print("Accuracy of the model is" + ":"
+str(np.mean(forest_model_scores)))
print("std of scores computed" + ":" +str(np.std(forest_model_scores)))
#make predictions with validation set
y_pred_forest = pd.DataFrame(forest_model.predict(X_test),
columns=["Term Deposit Predictions"])
print(y_pred_forest.head())
y_test.head()
```

The Random forest model accuracy was 0.8898539627847784 (89 % accuracy) accurate with a confusion matrix

```
[[3558   66]
 [ 378  117]]
```

	precision	recall	f1-score	support
0	0.90	0.98	0.94	3624
1	0.64	0.24	0.35	495
accuracy			0.89	4119
macro avg	0.77	0.61	0.64	4119
weighted avg	0.87	0.89	0.87	4119

as was the case in logistic regression.

### ***Feature Importance***

With the random forest algorithm, the features that contribute the most to the prediction can be known. Below shows the first 8 features that contributed the most to term deposit prediction.

```
#plot the 7 most important features
plt.figure(figsize=(10,7))
feat_importances = pd.Series(forest_model.feature_importances_,
index = Feature.columns)
feat_importances.nlargest(10).plot(kind='barh');
```

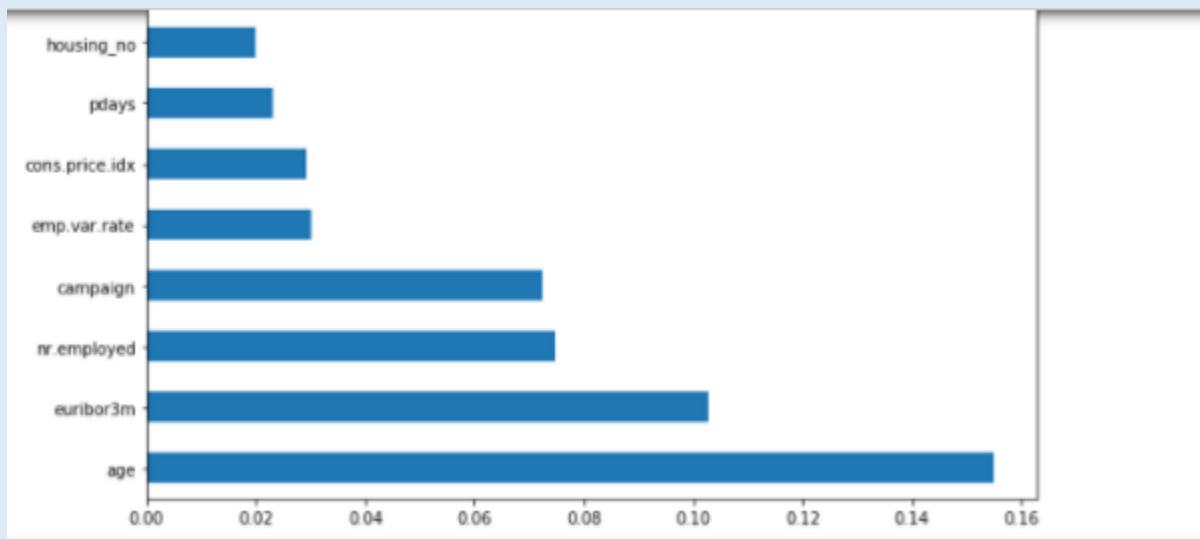


fig 10: Feature Importance

- **XGBoost**

The final machine learning algorithm considered was the XGBoost

The XGBoost model accuracy was 0.8979199297838093 (89 % accuracy) accurate with a confusion matrix

```
[[3517  107]
 [ 355  140]]
```

	precision	recall	f1-score	support
0	0.91	0.97	0.94	3624
1	0.57	0.28	0.38	495
accuracy			0.89	4119
macro avg	0.74	0.63	0.66	4119
weighted avg	0.87	0.89	0.87	4119

Though the precision is relatively same as that of random forest, the precision is lower.

### ***Conclusion***

Other machine learning classification algorithms can be used to train the model and their performances determined. So far the models though have good accuracy don't possess a good enough precision in terms of predicting a 'yes' for a term deposit. Further analysis can be done to improve the model such as hyperparameter tuning of the different machine learning algorithms employed, training the model with the important features from feature engineering.

### ***References***

- [How to Deal with Imbalanced Data using SMOTE](#)
- [How to Fix k-Fold Cross-Validation for Imbalanced Classification](#)
- [4 Automatic Outlier Detection Algorithms in Python](#)