|  |
| --- |
| Photo displaying partial image of two pie charts on a canvas-textured page |
| Term Deposit Prediction  Rofiah Adeshina |
| |  |  |  | | --- | --- | --- | | ROFEE | **8/29/20** | [Course title] | |

***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](http://archive.ics.uci.edu/ml/datasets/Bank+Marketing), 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:

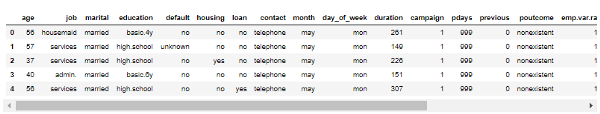


fig 1: Data Overview

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.

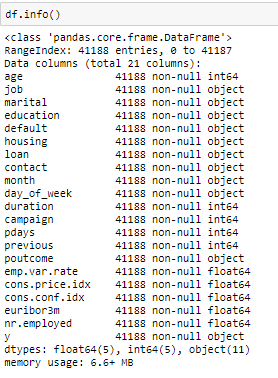


fig 2: Data Info

df.describe()

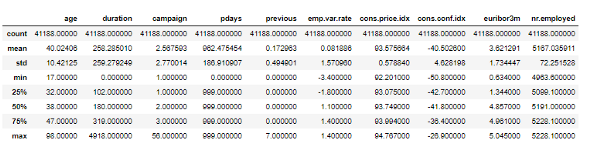


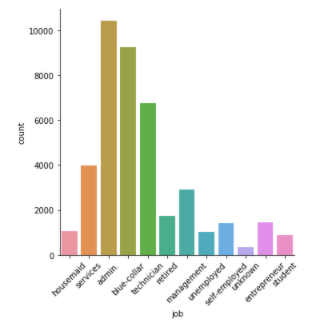
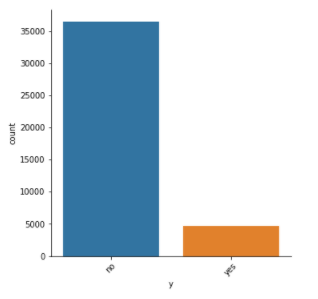
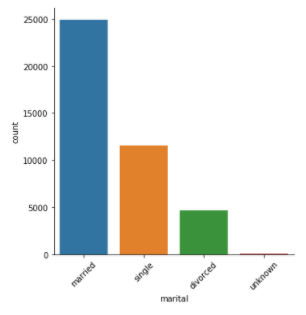
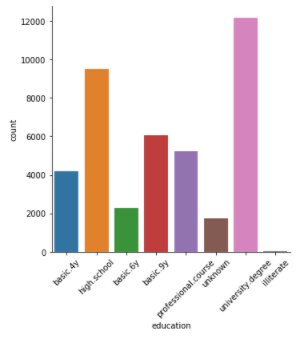
fig 3: Statistical Description of Data

***Univariate analysis***

#Univariante Analysis  
def univariante\_plot(column, data):  
 plot = sns.catplot(x=column, data=data, kind=’count’)  
 plot.set\_xticklabels(rotation=45)  
 plt.show()  
 return plot

univariante\_plot('y', df)  
univariante\_plot('job', df)  
univariante\_plot('education', df)  
univariante\_plot('marital', df)  
univariante\_plot('loan', df)  
univariante\_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 Plotsfig 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

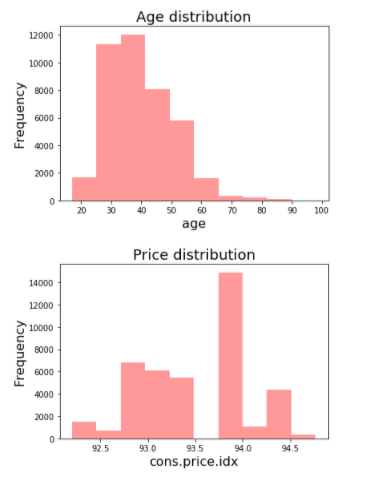


fig 4: Numerical Column Distribution

***Bivariate Analysis***

This shows how the features relate to one another

#bivariante Analysis  
def bivariante\_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

bivariante\_plot('marital', 'age', df, 'y')  
bivariante\_plot('job','age', df, 'y')

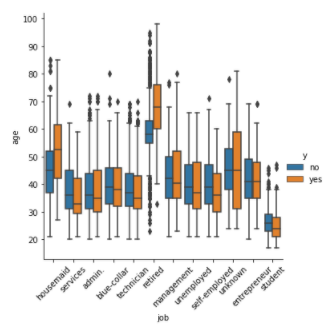
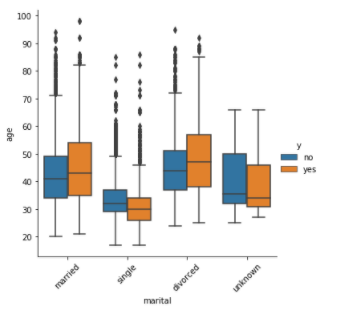


fig 5: Bi-variate Box Plots

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()  
 return chart

outlier('age', 'Age distribution')  
outlier('campaign', 'Campaign distribution')

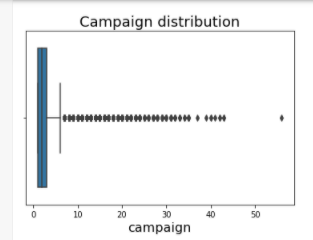


fig 7(a): Campaign Outlier Distribution

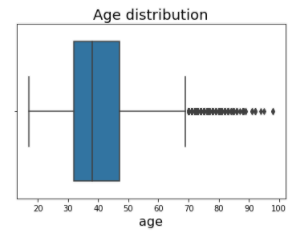


fig 7(b): Age Outlier Distribution

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”.

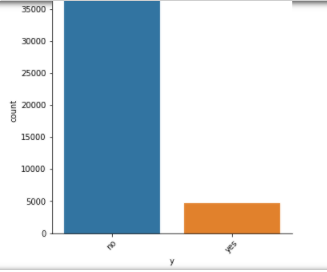


fig 8: Term Deposit Class Imbalance

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 **Te**chnique 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()

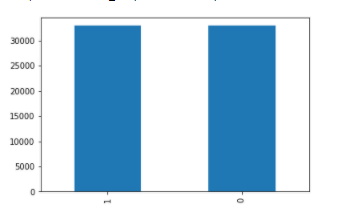


fig 9: Output after SMOTE

***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’);

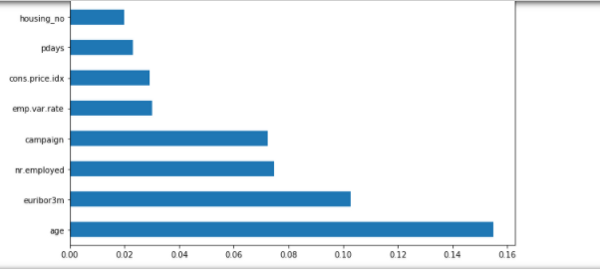


fig 10: Feature Importance

* **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](https://medium.com/analytics-vidhya/balance-your-data-using-smote-98e4d79fcddb)
* [How to Fix k-Fold Cross-Validation for Imbalanced Classification](https://machinelearningmastery.com/cross-validation-for-imbalanced-classification/)
* [4 Automatic Outlier Detection Algorithms in Python](https://machinelearningmastery.com/model-based-outlier-detection-and-removal-in-python/)