WEEK 2 – PART 1 & 2 TREE BASED MODELS

By: **GROUP 6**

HARKIRAT SINGH, HARSHIT GAUR, PUNEET MADAN, AKASH RAJ

MASTER OF PROFESSIONAL STUDIES IN ANALYTICS

ALY6040: DATA MINING

JULY 27, 2022

To: **PROF. KASUN SAMARSINGHE**

PART 1

INTRODUCTION

This dataset includes descriptions of hypothetical samples corresponding to 23 species of gilled mushrooms in the Agaricus and Lepiota Family Mushroom drawn from The Audubon Society Field Guide to North American Mushrooms (1981). Each species is identified as definitely edible, definitely poisonous, or of unknown edibility and not recommended. This latter class was combined with the poisonous one. The Guide clearly states that there is no simple rule for determining the edibility of a mushroom; no rule like "leaflets three, let it be'' for Poisonous Oak and Ivy.

Understanding the Dataset

```
mushrooms <- read_excel("mushrooms.xlsx")

# structure of the data
str(mushrooms)

## tibble [8,124 x 23] (S3: tbl_df/tbl/data.frame)
##  $ class                    : chr [1:8124] "p" "e" "e" "p" ...
##  $ cap-shape                : chr [1:8124] "x" "x" "b" "x" ...
##  $ cap-surface              : chr [1:8124] "s" "s" "s" "y" ...
##  $ cap-color                : chr [1:8124] "n" "y" "w" "w" ...
##  $ bruises                  : chr [1:8124] "t" "t" "t" "t" ...
##  $ odor                     : chr [1:8124] "p" "a" "l" "p" ...
##  $ gill-attachment          : chr [1:8124] "f" "f" "f" "f" ...
##  $ gill-spacing             : chr [1:8124] "c" "c" "c" "c" ...
##  $ gill-size                : chr [1:8124] "n" "b" "b" "n" ...
##  $ gill-color               : chr [1:8124] "k" "k" "n" "n" ...
##  $ stalk-shape              : chr [1:8124] "e" "e" "e" "e" ...
##  $ stalk-root               : chr [1:8124] "e" "c" "c" "e" ...
##  $ stalk-surface-above-ring: chr [1:8124] "s" "s" "s" "s" ...
##  $ stalk-surface-below-ring: chr [1:8124] "s" "s" "s" "s" ...
##  $ stalk-color-above-ring  : chr [1:8124] "w" "w" "w" "w" ...
##  $ stalk-color-below-ring  : chr [1:8124] "w" "w" "w" "w" ...
##  $ veil-type                : chr [1:8124] "p" "p" "p" "p" ...
##  $ veil-color               : chr [1:8124] "w" "w" "w" "w" ...
##  $ ring-number              : chr [1:8124] "o" "o" "o" "o" ...
##  $ ring-type                : chr [1:8124] "p" "p" "p" "p" ...
##  $ spore-print-color        : chr [1:8124] "k" "n" "n" "k" ...
##  $ population               : chr [1:8124] "s" "n" "n" "s" ...
##  $ habitat                  : chr [1:8124] "u" "g" "m" "u" ...
```

There are a total of 8124 observations in the Dataset along with 23 variables. The "Class" feature in the dataset is what we trying to classify using the various attributes available.

Missing Value

```
nrow(mushrooms) - sum(complete.cases(mushrooms))
```

```
## [1] 0
```

This command shows the number of missing values in the data set that will be required to be replaced for the analysis since Classification techniques like Decision Tree cannot operate on missing values.

Removing Redundant Variable

```
table(mushrooms$`veil-type`)
```

```
##
##    p
## 8124
```

```
mushrooms$veil.type <- NULL
```

Since all the values for this variable are the same i.e “p” hence we have removed the variable from the Analysis since it won’t add any value or contribute to Information Gain.

Analyzing the odor variable

```
table(mushrooms$class,mushrooms$odor)
```

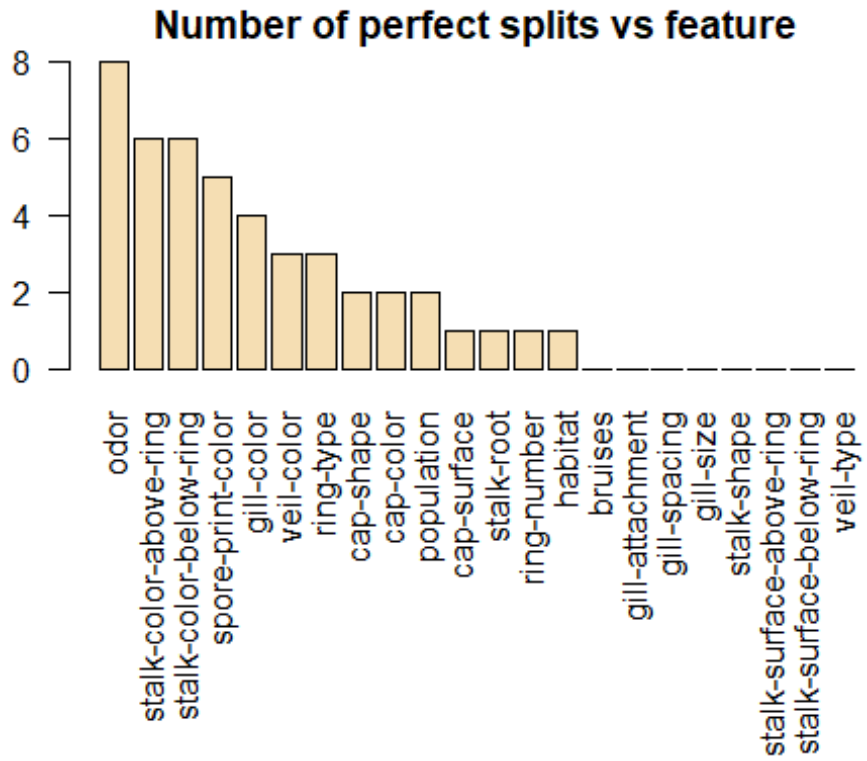
```
##
##        a    c    f    l    m    n    p    s    y
##   e  400    0    0  400    0 3408    0    0    0
##   p    0  192 2160    0   36  120  256  576  576
```

The above output shows that the mushrooms with odor values ‘c’, ‘f’, ‘m’, ‘p’, ‘s’ and ‘y’ are clearly poisonous. And the mushrooms having almond (a) odor (400) are edible. Such observations will help us to predict the output class more accurately.

Feature Importance

```
number.perfect.splits <- apply(X=mushrooms[-1], MARGIN = 2, FUN = function(co
l){
t <- table(mushrooms$class,col)
sum(t == 0)
})

# Descending order of perfect splits
order <- order(number.perfect.splits,decreasing = TRUE)
number.perfect.splits <- number.perfect.splits[order]
```

```
# Plot graph
par(mar=c(10,2,2,2))
barplot(number.perfect.splits,
main="Number of perfect splits vs feature",
xlab="",ylab="Feature",las=2,col="wheat")
```



To get a good understanding of the 21 predictor variables, I've created a table for each predictor variable vs class type (response/ outcome variable) in order to understand whether that particular predictor variable is significant for detecting the output or not.

Our next step in the data exploration stage is to predict which variable would be the best one for splitting the Decision Tree. For this reason, we have plotted a graph that represents the split for each of the 21 variables.

Data Splicing

```
set.seed(12345)
train <- sample(1:nrow(mushrooms),size = ceiling(0.80*nrow(mushrooms)),replac
e = FALSE)
# training set
mushrooms_train <- mushrooms[train,]
# test set
mushrooms_test <- mushrooms[-train,]
```

Data Splicing is the process of splitting the data into a training set and a testing set. The training set is used to build the Decision Tree model and the testing set is used to validate the efficiency of the model. The splitting is performed in the above code snippet

Adding Penalty

```
penalty.matrix <- matrix(c(0,1,10,0), byrow=TRUE, nrow=2)
```

In order to minimize the number of poisonous mushrooms misclassified as edible we will assign a penalty 10x bigger, than the penalty for classifying an edible mushroom as poisonous.

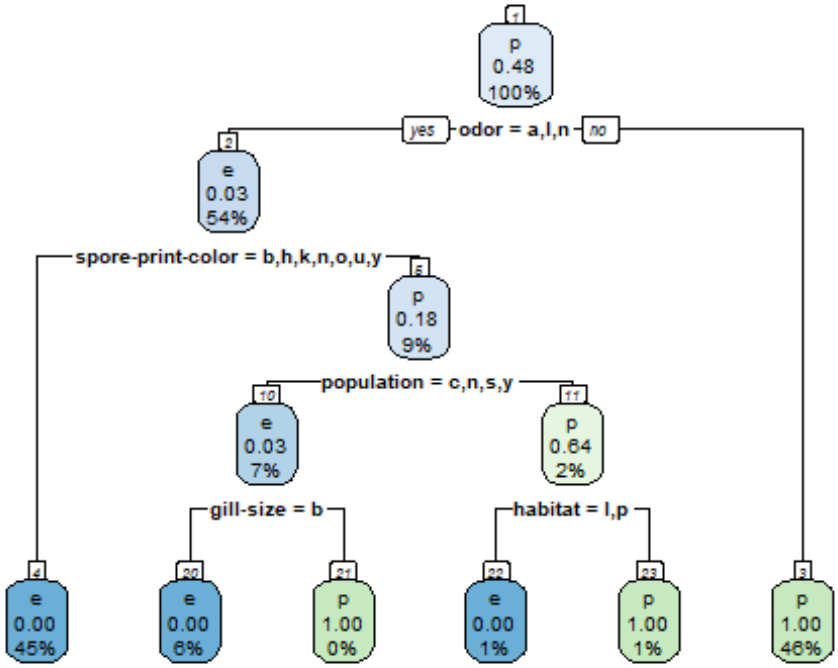
Building a model

```
tree <- rpart(class~.,
data=mushrooms_train,
parms = list(loss = penalty.matrix),
method = "class")
```

In this stage, we're going to build a Decision Tree by using the rpart (Recursive Partitioning And Regression Trees) algorithm. In the paramters we have added the loss function to inclusive of the penalty matrix.

Visualising the tree

```
rpart.plot(tree, nn=TRUE)
```



In this step, we'll be using the rpart.plot library to plot our final Decision Tree.

Pruning the tree

```
cp.optim <- tree$cptable[which.min(tree$cptable[,"xerror"]),"CP"]
tree <- prune(tree, cp=cp.optim)
```

To validate the model we use the printcp and plotcp functions. 'CP' stands for Complexity Parameter of the tree.We prune the tree to avoid any overfitting of the data. The convention is to have a small tree and the one with least cross validated error given by printcp() function i.e. 'xerror'.

Calculating Accuracy

```
pred <- predict(object=tree,mushrooms_test[-1],type="class")
t <- table(mushrooms_test$class,pred)
confusionMatrix(t)

## Confusion Matrix and Statistics
##
##    pred
##       e   p
##   e 829   0
##   p   0 795
```

```
##
##                Accuracy : 1
##                  95% CI : (0.9977, 1)
##     No Information Rate : 0.5105
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 1
##
##  Mcnemar's Test P-Value : NA
##
##             Sensitivity : 1.0000
##             Specificity : 1.0000
##          Pos Pred Value : 1.0000
##          Neg Pred Value : 1.0000
##              Prevalence : 0.5105
##          Detection Rate : 0.5105
##    Detection Prevalence : 0.5105
##       Balanced Accuracy : 1.0000
##
##        'Positive' Class : e
##
```

The output shows that all the samples in the test dataset have been correctly classified and we've attained an accuracy of 100% on the test data set with a 95% confidence interval (0.9977, 1). Thus we can correctly classify a mushroom as either poisonous or edible using this Decision Tree model.

Brief Code Walk-Through

In order to create a system that can forecast consumers' next purchases based on information from prior transactions as well as from customer meta data, we specifically targeted the dataset of the H&M Group (in-person shop and online store). For our use, we have used the Kaggle json credential file and script to get and import this dataset from Kaggle.

To confirm that the distribution of data points matched the values available in the dataset's features, we first did an exploratory data analysis (EDA). We next took the necessary steps for any missing values and unimportant features.

We established a calendar of dates spanning from 22nd March 2020 to 22nd June 2020 in order to achieve our goal of forecasting the purchases made by clients in the following 90 days from their individual prior transactions.

The feature engineering technique was used to shape our data points into the necessary data kinds, as well as to merge different data sets together to produce better dimensionality and information in a single final data set to be used for predictive model construction.

We also built our desired (needed) dependent variable using the cumulative output of different features in the data set, as well as some independent variables. In our data set, we took into account customers' historical purchasing behaviour, and the dependent variable 'next 90 days purchase' was serviced in accordance with the models' specifications.

Following all of these stages, we used three distinct models: Decision Tree Classifier, Random Forest Classifier, and Gradient Boosting Model to predict whether or not a consumer will make a purchase within 90 days of their previous transaction. We also looked for crucial features and trained our models in each scenario after applying hyper-parameter tuning to each one to acquire the optimal parameters set.

PART 2

Introduction

The H&M Group is a collection of brands and companies with about 4,850 physical locations and 53 online marketplaces. Customers can browse a wide assortment of products in our online store. However, if there are too many options, clients could not find what they are looking for or what intrigues them right away, which could prevent them from making a purchase. Product recommendations are essential for improving the buying experience. More importantly, assisting consumers in making sound decisions benefits sustainability since it lowers returns and, as a result, lowers transportation-related emissions.

We are going to develop a system to predict next purchases of customers based on data from previous transactions, as well as from customer meta data.

About the dataset

The dataset contains the following files.

- images/ - a folder of images corresponding to each article_id. The images are placed in subfolders starting with the first three digits of the article_id.
- articles.csv - CSV file containing the detailed metadata for each article_id available for purchase.
- customers.csv - CSV file containing the metadata for each customer_id in dataset.
- transactions_train.csv - CSV file containing the transactions data. It consists of the purchases of each customer for each date, as well as additional information. Duplicate rows correspond to multiple purchases of the same item.

We are going to start off our analysis by exploring the data, understanding the meaning and significance of the attributes involved properly, and perform necessary actions to streamline our process for the road to our goal.

Importing data from kaggle

```
import pandas as pd
import numpy as np
import seaborn as sns
import json
import datetime
import matplotlib.pyplot as plt
pd.set_option('display.max_columns', 40)
```

We are importing some necessary and important libraries/packages for our analysis.

- Pandas, Numpy, DateTime libraries have been imported to be utilized for exploring and molding the data set according to our needs.
- Seaborn, Matplotlib libraries have been imported to visually represent the data and insights from it.
- Json library has been imported to download and import dataset from Kaggle.

```
# Label Encoding and One-Hot Encoding Libraries
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import OneHotEncoder
from sklearn.compose import ColumnTransformer

# Standardization and Normalization Libraries
from sklearn.preprocessing import MinMaxScaler
from sklearn.preprocessing import StandardScaler

# Train Test Split
from sklearn.model_selection import train_test_split

# Missing Value Imputation
from sklearn.impute import SimpleImputer

# Decision Tree Classifier
from sklearn.tree import DecisionTreeClassifier

# Random Forest Classifier and Gradient Boosting Classifier
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier
from sklearn.datasets import make_classification

# SciPy.Stats for Plotting
import scipy.stats as stats
import pylab

# Metrics for understanding the model's performance
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, confusi
```

These libraries/packages have been imported to perform techniques of Label and One-Hot Encoding, Scaling (Standardization and Normalization), and splitting the data set into train-test sets randomly.

We also imported libraries for imputing missing vaues and Classifiers.

```
!mkdir ~/.kaggle
```

```
json_string = {"username":"mrnerd","key":"e90faefa3b2a5f6183c87004e6f7dd56"}
with open('/root/.kaggle/kaggle.json', 'w', encoding='utf-8') as f:
    json.dump(json_string, f, ensure_ascii=False, indent=4)
```

With credentials of Kaggle's account, we are creating a connection to kaggle's storage from where we will be downloading and importing the required data set into this google colab notebook.

```
!chmod 600 /root/.kaggle/kaggle.json
```

Giving '600' permission to the json file used for making connection to kaggle's storage for further usages.

Downloading and Unzipping specific files from Kaggle Dataset

```
! kaggle competitions download -c h-and-m-personalized-fashion-recommendations -f articles.c
! kaggle competitions download -c h-and-m-personalized-fashion-recommendations -f customers.
! kaggle competitions download -c h-and-m-personalized-fashion-recommendations -f sample_sub
! kaggle competitions download -c h-and-m-personalized-fashion-recommendations -f transactio
```

```
Downloading articles.csv.zip to /content
 94% 4.00M/4.26M [00:00<00:00, 35.8MB/s]
100% 4.26M/4.26M [00:00<00:00, 37.4MB/s]
Downloading customers.csv.zip to /content
 86% 84.0M/97.9M [00:01<00:00, 106MB/s]
100% 97.9M/97.9M [00:01<00:00, 88.8MB/s]
Downloading sample_submission.csv.zip to /content
 93% 47.0M/50.3M [00:01<00:00, 48.7MB/s]
100% 50.3M/50.3M [00:01<00:00, 43.5MB/s]
Downloading transactions_train.csv.zip to /content
100% 583M/584M [00:06<00:00, 127MB/s]
100% 584M/584M [00:06<00:00, 97.2MB/s]
```

In this step, we are downloading the 4 files from H&M's data set named '*h-and-m-personalized-fashion-recommendations*', stored in kaggle's data storage, into this google colab notebook.

```
! unzip /content/articles.csv.zip
! unzip /content/customers.csv.zip
! unzip /content/sample_submission.csv.zip
! unzip /content/transactions_train.csv.zip
```

```
Archive:  /content/articles.csv.zip
  inflating: articles.csv
Archive:  /content/customers.csv.zip
  inflating: customers.csv
Archive:  /content/sample_submission.csv.zip
  inflating: sample_submission.csv
Archive:  /content/transactions_train.csv.zip
  inflating: transactions_train.csv
```

When a dataset is downloaded or imported from kaggle's data storage, it is stored as a zip file in the google colab's environment.

In this step, we are unzipping the files to be used further for our analysis.

Initial Exploration on the data

We are reading the '*articles.csv*' file and storing the dataframe into a variable.

```
articles = pd.read_csv('/content/articles.csv')
```

This step will check the first 3 records of the dataframe to get a gist of the records present in it.

```
articles.head(3)
```

We are reading the '*customers.csv*' file and storing the dataframe into a variable.

```
customers = pd.read_csv('/content/customers.csv')
```

This step will check the first 3 records of the dataframe to get a gist of the records present in it.

```
customers.head(3)
```

	customer_id	FN	Active	club_member_status	fashic
0	00000dbacae5abe5e23885899a1fa44253a17956c6d1c3...	NaN	NaN	ACTIVE	
1	0000423b00ade91418cceaf3b26c6af3dd342b51fd051e...	NaN	NaN	ACTIVE	
2	000058a12d5b43e67d225668fa1f8d618c13dc232df0ca...	NaN	NaN	ACTIVE	

We are reading the '*transaction.csv*' file and storing the dataframe into a variable.

```
transactions = pd.read_csv('/content/transactions_train.csv')
```

This step checks the first 5 (default) records of the dataframe to get a gist of the records present in it.

```
transactions.head(3)
```

	t_dat	customer_id	article_id	price	sales_chann
0	2018-09-20	000058a12d5b43e67d225668fa1f8d618c13dc232df0ca...	663713001	0.050831	
1	2018-09-20	000058a12d5b43e67d225668fa1f8d618c13dc232df0ca...	541518023	0.030492	

```
# Model to predict if a customer will make a purchase within next 90 days of the last purcha
```

Exploratory Data Analysis

To check the columns of the dataset '*Customers*'

```
customers.columns
```

```
Index(['customer_id', 'FN', 'Active', 'club_member_status',
       'fashion_news_frequency', 'age', 'postal_code'],
      dtype='object')
```

To check the distribution of data points in accordance to the values present in dataset's features.

```
customers[['FN', 'Active', 'club_member_status']].value_counts()
```

```
FN   Active  club_member_status
1.0  1.0     ACTIVE                458452
             PRE-CREATE              5631
             LEFT CLUB                  3
dtype: int64
```

We found that '*customers*' data set has 2 features with a single value across the data points. They would deem to be insignificant features to our model for predictions and therefore, we will not use them further in our process.

```
## Filtering customers' demographics
customers = customers[['customer_id','club_member_status','fashion_news_frequency','age']]
```

In this step, we are checking the first transaction's date and last transaction's date in the *transactions* dataset along with respective number of transactions made on those dates.

```
## Checking Transaction dates and number of transactions
print(transactions['t_dat'].min(), ",", transactions[transactions['t_dat'] == transactions['

print(transactions['t_dat'].max(), ",", transactions[transactions['t_dat'] == transactions['
```

```
2018-09-20 , 48399
2020-09-22 , 32866
```

In the *transactions* dataset, a single feature related to date-time was present.

In order to treat this feature as a dateTime data type, we have converted it using pandas' function named 'to_datetime'.

```
## Converting date to datetime
transactions['t_dat']= pd.to_datetime(transactions['t_dat'])

transactions.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 31788324 entries, 0 to 31788323
Data columns (total 5 columns):
 #   Column            Dtype
---  ------            -----
 0   t_dat             datetime64[ns]
 1   customer_id       object
 2   article_id        int64
 3   price             float64
 4   sales_channel_id  int64
dtypes: datetime64[ns](1), float64(1), int64(2), object(1)
memory usage: 1.2+ GB
```

For the *articles* data set, we have already checked how the data points are stored using head function in one of above steps. We are now exploring the features and their data types to check if some engineering is required to be performed or not on our required features set.

```
articles.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 105542 entries, 0 to 105541
Data columns (total 25 columns):
 #   Column                        Non-Null Count   Dtype 
---  ------                        --------------   ----- 
 0   article_id                    105542 non-null  int64 
 1   product_code                  105542 non-null  int64 
 2   prod_name                     105542 non-null  object
 3   product_type_no               105542 non-null  int64 
 4   product_type_name             105542 non-null  object
 5   product_group_name            105542 non-null  object
 6   graphical_appearance_no       105542 non-null  int64 
 7   graphical_appearance_name     105542 non-null  object
 8   colour_group_code             105542 non-null  int64 
 9   colour_group_name             105542 non-null  object
 10  perceived_colour_value_id     105542 non-null  int64 
 11  perceived_colour_value_name   105542 non-null  object
 12  perceived_colour_master_id    105542 non-null  int64 
 13  perceived_colour_master_name  105542 non-null  object
 14  department_no                 105542 non-null  int64 
 15  department_name               105542 non-null  object
 16  index_code                    105542 non-null  object
 17  index_name                    105542 non-null  object
 18  index_group_no                105542 non-null  int64 
 19  index_group_name              105542 non-null  object
 20  section_no                    105542 non-null  int64 
 21  section_name                  105542 non-null  object
 22  garment_group_no              105542 non-null  int64 
 23  garment_group_name            105542 non-null  object
 24  detail_desc                   105126 non-null  object
dtypes: int64(11), object(14)
memory usage: 20.1+ MB
```

For our model building process and it's desired dataset, we have taken out the article IDs and the group names they belong to from *articles* dataset.

We have further merged this dataset with the *transactions* data set.

```
## Checking articles group
articles = articles[['article_id','index_group_name']]
```

```
## Merging transactions to articles
transactions = transactions.merge(articles,how='left',on = 'article_id')
```

To build a model where predictions will be made whether a customer will make a purchase within the next 90 days of their last purchase or not, we have taken a timeline of 22nd March 2020 to 22nd June 2020.

In below step, data is extracted within this timeline from *transactions* table.

```
## Taking 3 months range of dates to make the model
transactions_1 = transactions[(transactions['t_dat'] <= '2020-06-22') & (transactions['t_dat
```

Feature Engineering

We have 2 channels as a store (in-person) and online marketplace. To distinguish customers who use both the channels and make purchases from both of them, we have engineered a feature 'Channel 3' which

comprises of both the channels 1 and 2.

```
## Sales channel Feature engineering
## customer with both channels 1 and 2 marked as channel 3
lst1 = transactions_1[transactions_1['sales_channel_id'] == 1]['customer_id'].values
lst2 = transactions_1[transactions_1['sales_channel_id'] == 2]['customer_id'].values

lst3 = list(set(lst1).intersection(lst2))

lst2 =list(set(lst2)-set(lst3))
lst1 =list(set(lst1)-set(lst3))

channels = pd.DataFrame(zip([2]*len(lst2)+[1]*len(lst1)+[3]*len(lst3),lst2+lst1+lst3),column

transactions_1 = transactions_1.merge(channels,how='left', on = 'customer_id').drop('sales_c
```

In the below step, we are feature engineering the column *index_group_name* to add the feature of *shopping category group* in the above transactions data set.

```
## Adding shopping category group features
purchase_group = transactions_1.pivot_table(index='customer_id',columns= 'index_group_name',

purchase_group[['Baby/Children', 'Divided', 'Ladieswear', 'Menswear', 'Sport']] = np.where(p
```

For the construction of dependent variable and various other independent variables after feature engineering to be used for our predictive model, we are keeping customer's latest purchase records/transactions only within the chosen date timeline.

```
## Unique customer transactions within the chosen period (keeping latest purchase transactio
transactions_1 = transactions_1.sort_values('t_dat',ascending=False).drop_duplicates('custom

transactions_1 = transactions_1.drop(['index_group_name','price','article_id'],axis=1)

transactions_1 = transactions_1.merge(purchase_group,how='left',on ='customer_id')
```

Importing new library for datetime manipulation functions

```
from datetime import timedelta
```

In this below step, we are preparing our data set to include the feature *next_90_days_purchase* which signifies the purchase made by customer in the next 90 days from their last purchase.

This variable is based on the rolling 90 days from customers' last purchase.

```
## Purchase in next 90 days rolling from last purchase date (output variable for the model)

## Here, we are calculating the output variable tag. This tag represnts whether customer mad
## purchase within the next 90 days of his last purchase.

date_dic = {}
start_date = pd.to_datetime('2020-03-22').date()
```

```
for i in range(93):
    # start date of looking at transactions.
    small_date = str((start_date + timedelta(days =i)))
    # end date of looking at transactions over 90 days rolling period
    large_date = str((start_date + timedelta(days =i+90)))
    # saving customer ids for each start date in a dictionary
    date_dic[small_date] = list(transactions[(transactions['t_dat'] < large_date) & (transac

len(date_dic),date_dic.keys()

transactions_temp_1 = pd.DataFrame(columns=['t_dat', 'customer_id', 'channel', 'Baby/Childre
       'Ladieswear', 'Menswear', 'Sport', 'next_90_days_purchase'])

for i in range(93):
    start_date = pd.to_datetime('2020-03-22').date()
    date1 = str(start_date+timedelta(days=i))
    # print(date1)
    ## filtering transactions for each date
    transactions_temp = transactions_1[transactions_1['t_dat'] == date1]
    ## checking customer ids that made transactions in the next 90 days of the date.
    next_90_days_temp = pd.DataFrame(zip(date_dic[date1],[1]* len(date_dic[date1])),columns
    ## merging the data
    transactions_temp = transactions_temp.merge(next_90_days_temp,how='left',on ='customer_i
    transactions_temp_1 = transactions_temp_1.append(transactions_temp)

transactions_2 = transactions_temp_1.copy()

transactions_2['next_90_days_purchase'] = transactions_2.replace(np.nan,0)['next_90_days_pur

transactions_2['t_dat']= pd.to_datetime(transactions_2['t_dat'])

transactions_2.head()
```

	t_dat	customer_id	channel	Baby/Children	Divided
0	2020-03-22	53f385f0c0b4aaeb1b068c429506cfdb8ae9c635073bf0...	2	0	0
1	2020-03-22	5427b45555b5197e04bd8a734ba1f18c5785366248cc9e...	2	0	1
2	2020-03-22	54705646f873bf4117075cc74ef5f2e6fb9cb7763cbc30...	2	0	0
3	2020-03-22	5428334a49f0adc8314f876fbe80664227159dc45179fa...	2	0	0
4	2020-03-22	545732b4b16f87cd47ceba16e8fa3bef3d0250a9c5f676...	2	0	1

Next, we are extracting and preparing our data set to include the feature *last_30_days_purchase* which signifies the purchase made by customer in the last 30 days from their last purchase.

This variable represents the historic purchasing behaviour of the customer and the timeline considered is all set as one of the first steps in our analysis in this notebook.

```
## Historic purchase behaviour

### Last 30 days purchase
## Creating a feature for the immediate historic purchase behaviour. Checking if a customer
```

```
## made a transaction within the last 30 days of the latest purchase

date_dic = {}
start_date = pd.to_datetime('2020-06-22').date()
for i in range(93):
    # print(i)
    large_date = str((start_date - timedelta(days =i)))
    small_date = str((start_date - timedelta(days =i+30)))
    date_dic[large_date] = list(transactions[(transactions['t_dat'] < large_date) & (transac

len(date_dic),date_dic.keys()

transactions_temp_1 = pd.DataFrame(columns=['t_dat', 'customer_id', 'channel', 'Baby/Childre
       'Ladieswear', 'Menswear', 'Sport', 'next_90_days_purchase','last_30_days_purchase'])

for i in range(93):
    start_date = pd.to_datetime('2020-06-22').date()
    date1 = str(start_date-timedelta(days=i))
    #print(date1)
    transactions_temp = transactions_2[transactions_2['t_dat'] == date1]
    next_90_days_temp = pd.DataFrame(zip(date_dic[date1],[1]* len(date_dic[date1])),columns
    transactions_temp = transactions_temp.merge(next_90_days_temp,how='left',on ='customer_i
    #print(transactions_temp.shape,next_90_days_temp.shape)
    transactions_temp_1 = transactions_temp_1.append(transactions_temp)
    #print(transactions_temp_1.shape)

transactions_3 = transactions_temp_1.copy()
transactions_3['last_30_days_purchase'] = transactions_3.replace(np.nan,0)['last_30_days_pur
transactions_3['t_dat']= pd.to_datetime(transactions_3['t_dat'])
transactions_3.head()
```

	t_dat	customer_id	channel	Baby/Children	Divided
0	2020-06-22	ffffd7744cebcf3aca44ae7049d2a94b87074c3d4ffe38...	3	0	1
1	2020-06-22	5568cd1206b68b3a393d8061ef8b65280cd2cb53e056df...	2	0	1
2	2020-06-22	5562af7b2ae6fc5877c8d09dd1f0e5c146770cd1749ec1...	1	0	0
3	2020-06-22	5563362a23822df5b506b6454bbd56e979f519158740dd...	1	0	1
4	2020-06-22	55671e825ca601d36e09c3232f51300e2bfed7f29ecbcb...	3	0	1

As one of the last step for our data preparation, we are extracting information and preparing a feature *last_90_days_purchase* which signifies the purchase made by customer in the last 90 days from their last purchase respectively.

This variable represents the mid-term historic purchasing behaviour of the customer.

```
### Last 90 days purchase

## Creating a feature for the mid-term historic purchase behaviour. Checking if a customer
## made a transaction within the last 90 days of the latest purchase
```

```
date_dic = {}
start_date = pd.to_datetime('2020-06-22').date()
for i in range(93):
    #print(i)
    large_date = str((start_date - timedelta(days =i)))
    small_date = str((start_date - timedelta(days =i+90)))
    date_dic[large_date] = list(transactions[(transactions['t_dat'] < large_date) & (transac

len(date_dic),date_dic.keys()

transactions_temp_1 = pd.DataFrame(columns=['t_dat', 'customer_id', 'channel', 'Baby/Childre
       'Ladieswear', 'Menswear', 'Sport', 'next_90_days_purchase','last_30_days_purchase','l

for i in range(93):
    start_date = pd.to_datetime('2020-06-22').date()
    date1 = str(start_date-timedelta(days=i))
    #print(date1)
    transactions_temp = transactions_3[transactions_3['t_dat'] == date1]
    next_90_days_temp = pd.DataFrame(zip(date_dic[date1],[1]* len(date_dic[date1])),columns
    transactions_temp = transactions_temp.merge(next_90_days_temp,how='left',on ='customer_i
    transactions_temp_1 = transactions_temp_1.append(transactions_temp)

transactions_4 = transactions_temp_1.copy()

transactions_4['last_90_days_purchase'] = transactions_4.replace(np.nan,0)['last_90_days_pur

transactions_4['t_dat']= pd.to_datetime(transactions_4['t_dat'])

transactions_4.head()
```

	t_dat	customer_id	channel	Baby/Children	Divided
0	2020-06-22	ffffd7744cebcf3aca44ae7049d2a94b87074c3d4ffe38...	3	0	1
1	2020-06-22	5568cd1206b68b3a393d8061ef8b65280cd2cb53e056df...	2	0	1
2	2020-06-22	5562af7b2ae6fc5877c8d09dd1f0e5c146770cd1749ec1...	1	0	0
3	2020-06-22	5563362a23822df5b506b6454bbd56e979f519158740dd...	1	0	1
4	2020-06-22	55671e825ca601d36e09c3232f51300e2bfed7f29ecbcb...	3	0	1

After preparing the final dataset for our predicting modelling, we are going to append customers' with their respective details from the *customers* dataset in each transactions.

```
## adding customer attributes to transactions
transactions_4 = transactions_4.merge(customers,how='left',on='customer_id')

transactions_4['age'] = transactions_4['age'].fillna(transactions_4['age'].mean()).astype(in

transactions_4.drop(['t_dat','customer_id'],axis=1,inplace=True)

transactions_4
```

	channel	Baby/Children	Divided	Ladieswear	Menswear	Sport	next_90_days_purch
0	3	0	1	1	0	0	
1	2	0	1	1	0	0	
2	1	0	0	1	0	0	
3	1	0	1	1	1	0	
4	3	0	1	1	0	1	
...	...	...	...	...	...	...	
520042	2	0	0	1	0	0	
520043	2	0	0	1	0	0	
520044	2	0	0	1	0	0	
520045	2	0	1	1	0	0	
520046	2	0	1	1	0	0	

520047 rows × 12 columns

Predictive Modeling

Dependent & Independent Variables

Dependent Variable - *next_90_days_purchase*

Independent Variable - All of other features in the data set which were feature-engineered or taken directly from the data set except *next_90_days_purchase*

```
## Output and input variables for the model
Y = transactions_4['next_90_days_purchase']
X = transactions_4.drop('next_90_days_purchase',axis=1)
```

Data Preprocessing - One Hot Encoding, Scaling

Since, in our dataset, all the categorical variables are not ranked. Therefore, we are proceeding further with One Hot Encoding to these categorical variables.

We are applying scaling technique of *Standard Scaler* to scale the feature around unit variance only.

```
## Encoding the variables
one_hot = ['channel','club_member_status','fashion_news_frequency']
## Scaling this numeric feature.
std_scale  = ['age']

ct = ColumnTransformer(transformers=[('sc',StandardScaler(),std_scale),('encoder',OneHotEnco
```

```
X = ct.fit_transform(X)

X  = X.astype(float)
Y = Y.values
Y= Y.astype(int)
```

Test Train Split

Data Splicing is the process of splitting the data into a training set and a testing set. The training set is used to build the models and the testing set is used to validate the efficiency of the model. The splitting is performed in the above code snippet.

```
##Checking class ratio
sum(Y)/len(Y)

X_train, X_val, Y_train, Y_val = train_test_split(X, Y,stratify=Y, random_state=0)
print(X_train.shape, X_val.shape, Y.shape)
```

```
(390035, 19) (130012, 19) (520047,)
```

Model Building and Validation

Decision Tree

```
labels = transactions_4['next_90_days_purchase']
```

Hyper Parameter Tuning

We are moving forward with hyper-parameter tuning using GridSearchCV for the decision tree model to retrieve the best possible set of hyper-parameters for this data set.

```
from sklearn.model_selection import GridSearchCV

tuned_parameters = [{'max_depth': [1,2,3,4,5],
                     'min_samples_split': [2,4,6,8,10]}]
scores = ['recall']
for score in scores:
    print()
    print(f"Tuning hyperparameters for {score}")
    print()

    clf = GridSearchCV(
        DecisionTreeClassifier(), tuned_parameters,
        scoring = f'{score}_macro')
    clf.fit(X_train, Y_train)

    print("Best parameters set found on development set:")
    print()
    print(clf.best_params_)
    print()
    print("Grid scores on development set:")
```

```
    means = clf.cv_results_["mean_test_score"]
    stds = clf.cv_results_["std_test_score"]
    for mean, std, params in zip(means, stds,
                                 clf.cv_results_['params']):
        print(f"{mean:0.3f} (+/-{std*2:0.03f}) for {params}")
```

```
Tuning hyperparameters for recall

Best parameters set found on development set:

{'max_depth': 5, 'min_samples_split': 2}

Grid scores on development set:
0.655 (+/-0.002) for {'max_depth': 1, 'min_samples_split': 2}
0.655 (+/-0.002) for {'max_depth': 1, 'min_samples_split': 4}
0.655 (+/-0.002) for {'max_depth': 1, 'min_samples_split': 6}
0.655 (+/-0.002) for {'max_depth': 1, 'min_samples_split': 8}
0.655 (+/-0.002) for {'max_depth': 1, 'min_samples_split': 10}
0.667 (+/-0.002) for {'max_depth': 2, 'min_samples_split': 2}
0.667 (+/-0.002) for {'max_depth': 2, 'min_samples_split': 4}
0.667 (+/-0.002) for {'max_depth': 2, 'min_samples_split': 6}
0.667 (+/-0.002) for {'max_depth': 2, 'min_samples_split': 8}
0.667 (+/-0.002) for {'max_depth': 2, 'min_samples_split': 10}
0.680 (+/-0.002) for {'max_depth': 3, 'min_samples_split': 2}
0.680 (+/-0.002) for {'max_depth': 3, 'min_samples_split': 4}
0.680 (+/-0.002) for {'max_depth': 3, 'min_samples_split': 6}
0.680 (+/-0.002) for {'max_depth': 3, 'min_samples_split': 8}
0.680 (+/-0.002) for {'max_depth': 3, 'min_samples_split': 10}
0.685 (+/-0.003) for {'max_depth': 4, 'min_samples_split': 2}
0.685 (+/-0.003) for {'max_depth': 4, 'min_samples_split': 4}
0.685 (+/-0.003) for {'max_depth': 4, 'min_samples_split': 6}
0.685 (+/-0.003) for {'max_depth': 4, 'min_samples_split': 8}
0.685 (+/-0.003) for {'max_depth': 4, 'min_samples_split': 10}
0.685 (+/-0.003) for {'max_depth': 5, 'min_samples_split': 2}
0.685 (+/-0.003) for {'max_depth': 5, 'min_samples_split': 4}
0.685 (+/-0.003) for {'max_depth': 5, 'min_samples_split': 6}
0.685 (+/-0.003) for {'max_depth': 5, 'min_samples_split': 8}
0.685 (+/-0.003) for {'max_depth': 5, 'min_samples_split': 10}
```

Incorporating the results of hyper-parameters from hyperparameter tuning in the above step, we are building the decision tree model in the step.

Best parameters set found on development set:

{'max_depth': 5, 'min_samples_split': 2}

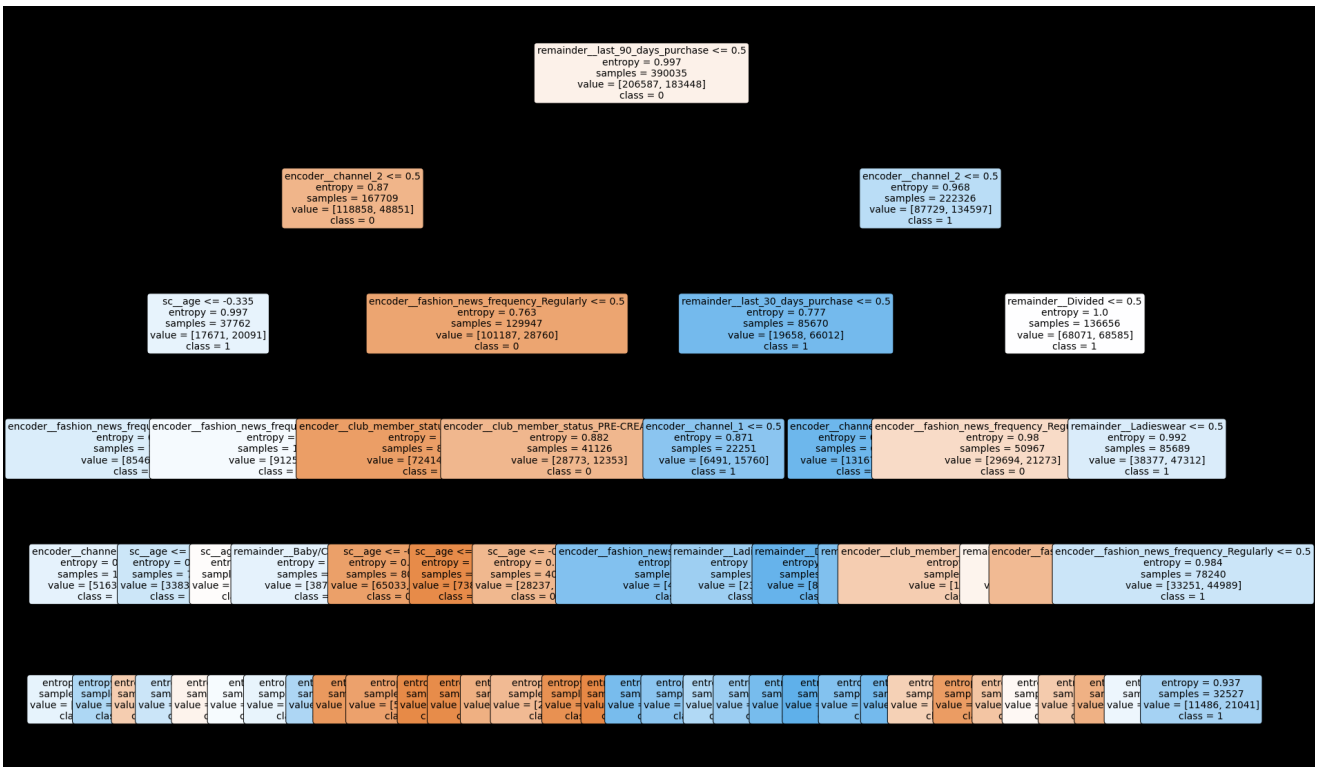
```
## Decision Tree Classifier
clf = DecisionTreeClassifier(criterion='entropy', max_depth = 5, min_samples_split = 2, rand
clf.fit(X_train, Y_train)
test_pred_decision_tree = clf.predict(X_val)
```

```
## Import Necessary Packages
from sklearn import tree
import matplotlib.pyplot as plt
```

Feature Importance

```
# Plot the figure, Setting a black background
plt.figure(figsize=(30,20), facecolor ='k')
```

```
# Create the Tree Plot
a = tree.plot_tree(
      clf, feature_names = ct.get_feature_names_out(), class_names = labels.astype('str'),
      rounded = True, filled = True, fontsize=14
    )

#show the plot
plt.show()
```




```
#import relevant functions
from sklearn.tree import export_text#export the decision rules
tree_rules = export_text(clf, feature_names = list(ct.get_feature_names_out()))#print the re

print(tree_rules)
```

```
|--- remainder__last_90_days_purchase <= 0.50
|   |--- encoder__channel_2 <= 0.50
|   |   |--- sc__age <= -0.34
|   |   |   |--- encoder__fashion_news_frequency_Regularly <= 0.50
|   |   |   |   |--- encoder__channel_3 <= 0.50
|   |   |   |   |   |--- class: 1
|   |   |   |   |--- encoder__channel_3 >  0.50
|   |   |   |   |   |--- class: 1
|   |   |   |--- encoder__fashion_news_frequency_Regularly >  0.50
|   |   |   |   |--- sc__age <= -1.27
|   |   |   |   |   |--- class: 0
|   |   |   |   |--- sc__age >  -1.27
|   |   |   |   |   |--- class: 1
|   |   |--- sc__age >  -0.34
|   |   |   |--- encoder__fashion_news_frequency_Regularly <= 0.50
|   |   |   |   |--- sc__age <= 0.82
|   |   |   |   |   |--- class: 0
|   |   |   |   |--- sc__age >  0.82
|   |   |   |   |   |--- class: 1
|   |   |   |--- encoder__fashion_news_frequency_Regularly >  0.50
```

```
|   |   |   |   |--- remainder__Baby/Children <= 0.50
|   |   |   |   |   |--- class: 1
|   |   |   |   |--- remainder__Baby/Children >  0.50
|   |   |   |   |   |--- class: 1
|   |   |--- encoder__channel_2 >  0.50
|   |   |   |--- encoder__fashion_news_frequency_Regularly <= 0.50
|   |   |   |   |--- encoder__club_member_status_PRE-CREATE <= 0.50
|   |   |   |   |   |--- sc__age <= -0.98
|   |   |   |   |   |   |--- class: 0
|   |   |   |   |   |--- sc__age >  -0.98
|   |   |   |   |   |   |--- class: 0
|   |   |   |   |--- encoder__club_member_status_PRE-CREATE >  0.50
|   |   |   |   |   |--- sc__age <= -0.84
|   |   |   |   |   |   |--- class: 0
|   |   |   |   |   |--- sc__age >  -0.84
|   |   |   |   |   |   |--- class: 0
|   |   |   |--- encoder__fashion_news_frequency_Regularly >  0.50
|   |   |   |   |--- encoder__club_member_status_PRE-CREATE <= 0.50
|   |   |   |   |   |--- sc__age <= -0.98
|   |   |   |   |   |   |--- class: 0
|   |   |   |   |   |--- sc__age >  -0.98
|   |   |   |   |   |   |--- class: 0
|   |   |   |   |--- encoder__club_member_status_PRE-CREATE >  0.50
|   |   |   |   |   |--- sc__age <= 1.46
|   |   |   |   |   |   |--- class: 0
|   |   |   |   |   |--- sc__age >  1.46
|   |   |   |   |   |   |--- class: 0
|   |--- remainder__last_90_days_purchase >  0.50
|   |   |--- encoder__channel_2 <= 0.50
|   |   |   |--- remainder__last_30_days_purchase <= 0.50
|   |   |   |   |--- encoder__channel_1 <= 0.50
|   |   |   |   |   |--- encoder__fashion_news_frequency_NONE <= 0.50
|   |   |   |   |   |   |--- class: 1
|   |   |   |   |   |--- encoder__fashion_news_frequency_NONE >  0.50
|   |   |   |   |   |   |--- class: 1
|   |   |   |   |--- encoder__channel_1 >  0.50
|   |   |   |   |   |--- remainder__Ladieswear <= 0.50
```

```
# Extract Importance
importance = pd.DataFrame({'feature': ct.get_feature_names_out(), 'importance' : np.round(cl
importance.sort_values('importance', ascending=False, inplace = True)
print(importance)
```

	feature	importance
18	remainder__last_90_days_purchase	0.487
2	encoder__channel_2	0.369
10	encoder__fashion_news_frequency_Regularly	0.048
13	remainder__Divided	0.034
14	remainder__Ladieswear	0.025
6	encoder__club_member_status_PRE-CREATE	0.016
1	encoder__channel_1	0.008
17	remainder__last_30_days_purchase	0.008
0	sc__age	0.004
9	encoder__fashion_news_frequency_NONE	0.002
8	encoder__fashion_news_frequency_Monthly	0.000
7	encoder__club_member_status_nan	0.000
11	encoder__fashion_news_frequency_nan	0.000
12	remainder__Baby/Children	0.000
5	encoder__club_member_status_LEFT CLUB	0.000
4	encoder__club_member_status_ACTIVE	0.000
15	remainder__Menswear	0.000
16	remainder__Sport	0.000
3	encoder__channel_3	0.000

```
confusion_matrix = confusion_matrix(Y_val, test_pred_decision_tree) #get the confusion matri
```

```
matrix_df = pd.DataFrame(confusion_matrix) #turn this into a dataframe
```

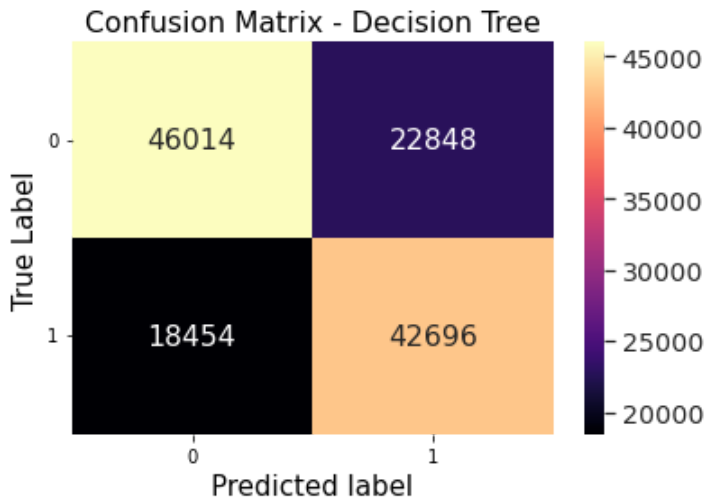
```
accuracy_score(Y_val, test_pred_decision_tree)
```

```
0.6823216318493678
```

Observation - The accuracy of the training and the test set is almost similar, around 69%, indicating that the model is not over fitting.

Confusion Matrix

```
ax = plt.axes() #plot the result
sns.set(font_scale=1.3)
plt.figure(figsize=(10,7))
sns.heatmap(matrix_df, annot=True, fmt="g", ax=ax, cmap="magma")
ax.set_title('Confusion Matrix - Decision Tree') #set axis titles
ax.set_xlabel("Predicted label", fontsize =15)
ax.set_ylabel("True Label", fontsize=15)
ax.set_yticklabels(list(labels), rotation = 0)
plt.show()
```




```
<Figure size 720x504 with 0 Axes>
```

We can infer from the confusion matrix as well as the accuracy the same kind of information that the model is not over-fitting.

The True-Positive Ratio with the Total Positives (Precision) also comes out to be essentially good at around 0.72 with a little fall in Recall metric.

We performed hyper-parameter tuning using Grid Search to find out the best possible parameter set for the model and the data set. The parameters 'min_sample_split' and 'max_depth' were provided as the best possible parameters after tuning by the model.

```
#get the precision score
precision = precision_score(Y_val,
                            test_pred_decision_tree,
                            average=None)

#turn it into a dataframe
precision_results = pd.DataFrame(precision, index=labels.unique())
```

```
#rename the results column
precision_results.rename(columns={0:'precision'}, inplace =True)
precision_results
```

	precision
0	0.713749
1	0.651410

```
recall = recall_score(Y_val, test_pred_decision_tree,
                              average =None)
recall_results = pd.DataFrame(recall, index= labels.unique())
recall_results.rename(columns ={0:'Recall'}, inplace =True)
recall_results
```

	Recall
0	0.668206
1	0.698217

```
f1 = f1_score(Y_val, test_pred_decision_tree, average=None)
f1_results = pd.DataFrame(f1, index=labels.unique())
f1_results.rename(columns={0:'f1'}, inplace=True)
f1_results
```

	f1
0	0.690227
1	0.674002

Observation - The precision and recall of the decision tree model is close to 69% for the '0' class and around 67% for the '1' class.

Random Forest

After the implementation of Decision Tree, we have applied Random Forest model to predict churning of customers.

To be able to validate the model, Out of the Bag Score (OOB Score) has been used which comes out ot be around 68%. We did not just go with the accuracy score of the model as the model may have been overfitting and accuracy wouldn't be the right metric to use.

```
clf = RandomForestClassifier(n_estimators= 100, oob_score = True, random_state=0)
clf.fit(X_train, Y_train)

print(clf.oob_score_)
```

```
0.6843821708308229
```

To apply hyperparameter tuning and find the best set of parameters to find the best solution for the built model, we are using *Randomized Search CV* technique for the model.

We are going to use the numbers of estimators in the range of 50 to 200 with intervals of 15 and maximum features from 0.1 till 1.

The model with parameters tuned has been created and will be used further to predict the churn of customers.

```
from sklearn.model_selection import RandomizedSearchCV

param_grid = {'n_estimators':np.arange(50,200,15),
              'max_features':np.arange(0.1, 1, 0.1),
              'max_depth': [3, 5, 7, 9],
              'max_samples': [0.3, 0.5, 0.8]}

clf = RandomizedSearchCV(RandomForestClassifier(), param_grid, n_iter = 15).fit(X_train, Y_t
clf = clf.best_estimator_
```

```
clf
```

```
RandomForestClassifier(max_depth=9, max_features=0.7000000000000001,
                       max_samples=0.3, n_estimators=110)
```

We have performed hyper-parameter tuning and retreived the best parameters to be used in Random Forest Classifier.

```
clf = RandomForestClassifier(max_depth=9, max_features=0.7000000000000001, max_samples=0.3,
clf.fit(X_train, Y_train)
test_pred_decision_tree = clf.predict(X_val)
```

In this step, we are using the random forest classifier model and predicting the churning of customers from the testing dataset.

Feature Importance

```
# Extract Importance
importance = pd.DataFrame({'feature': ct.get_feature_names_out(), 'importance' : np.round(cl
importance.sort_values('importance', ascending=False, inplace = True)

print(importance)
```

```
                                    feature  importance
18           remainder__last_90_days_purchase       0.371
2                          encoder__channel_2       0.323
3                          encoder__channel_3       0.058
13                         remainder__Divided       0.036
0                                   sc__age       0.033
10  encoder__fashion_news_frequency_Regularly       0.032
1                          encoder__channel_1       0.031
14                      remainder__Ladieswear       0.030
17           remainder__last_30_days_purchase       0.028
6      encoder__club_member_status_PRE-CREATE       0.013
15                        remainder__Menswear       0.012
```

```
9         encoder__fashion_news_frequency_NONE       0.012
16                          remainder__Sport       0.010
4           encoder__club_member_status_ACTIVE       0.005
12                    remainder__Baby/Children       0.004
11         encoder__fashion_news_frequency_nan       0.001
5        encoder__club_member_status_LEFT CLUB       0.000
7              encoder__club_member_status_nan       0.000
8      encoder__fashion_news_frequency_Monthly       0.000
```

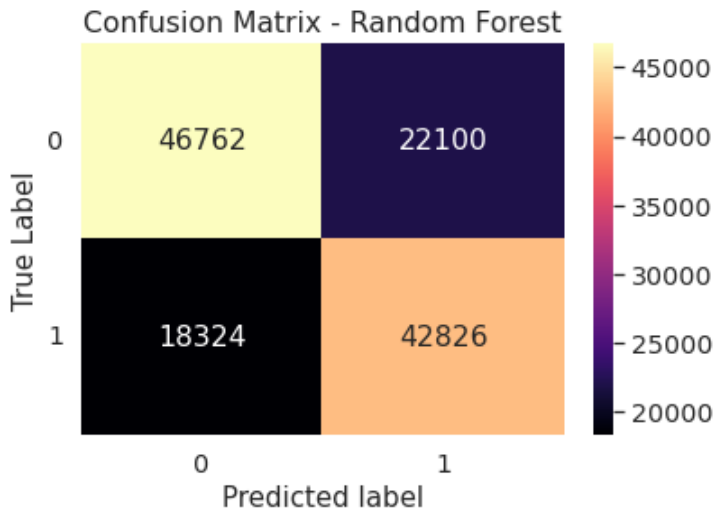
```
print(confusion_matrix(Y_val, test_pred_decision_tree))
print(classification_report(Y_val, test_pred_decision_tree))
print(accuracy_score(Y_val, test_pred_decision_tree))
```

```
[[46762 22100]
 [18324 42826]]
              precision    recall  f1-score   support

           0       0.72      0.68      0.70     68862
           1       0.66      0.70      0.68     61150

    accuracy                           0.69    130012
   macro avg       0.69      0.69      0.69    130012
weighted avg       0.69      0.69      0.69    130012

0.6890748546288035
```

Observation - The accuracy of the training and the test set is almost similar, around 69%, indicating that the model is not over fitting.

```
confusion_matrix = confusion_matrix(Y_val, test_pred_decision_tree) #get the confusion matri
matrix_df = pd.DataFrame(confusion_matrix) #turn this into a dataframe
```

```
ax = plt.axes() #plot the result
sns.set(font_scale=1.3)
plt.figure(figsize=(10,7))
sns.heatmap(matrix_df, annot=True, fmt="g", ax=ax, cmap="magma")
ax.set_title('Confusion Matrix - Random Forest') #set axis titles
ax.set_xlabel("Predicted label", fontsize =15)
ax.set_ylabel("True Label", fontsize=15)
ax.set_yticklabels(list(labels), rotation = 0)
plt.show()
```




```
<Figure size 720x504 with 0 Axes>
```

Similar information can be inferred from the confusion matrix as well. The accuracy, precision, and recall were tried to be improved using hyperparamter tuning of the model using RandomizedSearchCV.

The True-Positive Ratio with the Total Positives (Precision) also comes out to be essentially good at 0.72 with a little fall in Recall metric.

Gradient Boosting Classifier

After the implementation of Decision Tree and Random Forest Classification, we have applied another ensemble technique of Gradient Boosting Classification to predict whether a customer will buy in the next 90 days. Boosting is a sequential technique which works on the principle of ensemble. It combines a set of weak learners and delivers improved prediction accuracy.

A base GBM model is fit with 50 trees, learning rate of 0.1, and max depth of tree as 10.

```
clf = GradientBoostingClassifier(n_estimators=50, learning_rate=0.1, max_depth=10, random_st
clf.fit(X_train, Y_train)
```

```
GradientBoostingClassifier(max_depth=10, n_estimators=50, random_state=0)
```

```
print(accuracy_score(clf.predict(X_train),Y_train))
print(accuracy_score(clf.predict(X_val),Y_val))
```

```
0.6995910623405592
0.6881134049164692
```

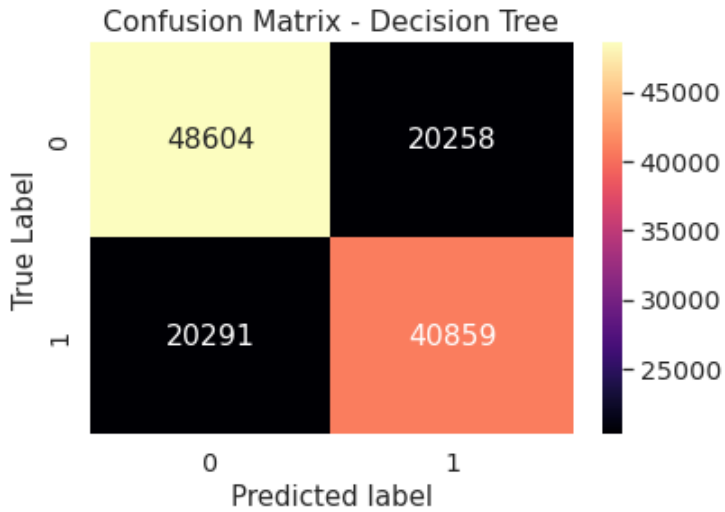
Observation - The accuracy of the training and the test set is almost similar, around 69%, indicating that the model is not over fitting.

```
from sklearn.metrics import classification_report
print("Classification Report")
print(classification_report(clf.predict(X_val), Y_val))
```

```
Classification Report
              precision    recall  f1-score   support

           0       0.71      0.71      0.71     68895
           1       0.67      0.67      0.67     61117

    accuracy                           0.69    130012
   macro avg       0.69      0.69      0.69    130012
weighted avg       0.69      0.69      0.69    130012
```

```
confusion_matrix = confusion_matrix(Y_val,clf.predict(X_val))

matrix_df = pd.DataFrame(confusion_matrix)
#plot the result
ax = plt.axes()
sns.set(font_scale=1.3)
plt.figure(figsize=(10,7))
sns.heatmap(matrix_df, annot=True, fmt="g", ax=ax, cmap="magma")
```

```
#set axis titles
ax.set_title('Confusion Matrix - Decision Tree')
ax.set_xlabel("Predicted label", fontsize =15)
ax.set_ylabel("True Label", fontsize=15)4

plt.show()
```




```
<Figure size 720x504 with 0 Axes>
```

Similar information can be inferred from the confusion matrix. The accuracy, precision, and recall can be tried to improve using hyperparamter tuning of the model using grid search and indentifying the optimal learning rate.

Grid search is used to tune the hyperparameters of the individual models and the boosting operation in GBM. The parameters 'min_sample_split' and 'min_samples_leaf' will be tuned. The 'min_sample_split' is kept in the range of 1000 and 2100, whereaas 'min_samples_leaf' is in the range of 30 and 71.

The other parameters of the model are kept constant. The attribute 'best_params_' of the GridSearchCV class gives the best parameters on which the model will have the highest accuracy.

```
param_test = {'min_samples_split':range(1000,2100,200), 'min_samples_leaf':range(30,71,10)}
gsearch = GridSearchCV(estimator = GradientBoostingClassifier(learning_rate=0.1, n_estimator
                       param_grid = param_test, scoring='roc_auc', n_jobs=4, cv=5)
gsearch.fit(X_train, Y_train)
gsearch.best_params_, gsearch.best_score_
```

```
({'min_samples_leaf': 50, 'min_samples_split': 1800}, 0.7581624118938551)
```

Another paramter that can be tuned is the learning rate. Learning rate helps determine the optimal value of the optimisation process and the execution speed. In order to find a better solution to our built model by changing parameters, we built a method to change the learning rate parameter of the model and check the model's accuracy on both of the training and testing data sets. The model with best parameters generated through grid search is run across loop to indentify the optimal learning rate.

```
lr_list = [0.05, 0.075, 0.1, 0.25, 0.5, 0.75, 1]

for learning_rate in lr_list:
    gb_clf = GradientBoostingClassifier(learning_rate=learning_rate, n_estimators=60,max_dep
```

```
    gb_clf.fit(X_train, Y_train)

    print("Learning rate: ", learning_rate)
    print("Accuracy score (training): {0:.3f}".format(gb_clf.score(X_train, Y_train)))
    print("Accuracy score (validation): {0:.3f}".format(gb_clf.score(X_val, Y_val)))
```

```
Learning rate:  0.05
Accuracy score (training): 0.694
Accuracy score (validation): 0.690
Learning rate:  0.075
Accuracy score (training): 0.696
Accuracy score (validation): 0.690
Learning rate:  0.1
Accuracy score (training): 0.696
Accuracy score (validation): 0.689
Learning rate:  0.25
Accuracy score (training): 0.698
Accuracy score (validation): 0.688
Learning rate:  0.5
Accuracy score (training): 0.699
Accuracy score (validation): 0.686
Learning rate:  0.75
Accuracy score (training): 0.700
Accuracy score (validation): 0.685
Learning rate:  1
Accuracy score (training): 0.698
Accuracy score (validation): 0.685
```

In this case, the learning rate does not have any major impact on the training and the validation accuracy score. With the change in learning rate, there is a 0.1% increment in the accuracy.

Since the difference in accuracy is not very high, we will keep a smaller learning rate to improve the execution speed of the model. Fitting a new model with the paramters obtained from the grid search and learning rate.

```
clf = GradientBoostingClassifier(n_estimators=100, min_samples_leaf=50, min_samples_split=18
clf.fit(X_train, Y_train)
```

```
GradientBoostingClassifier(learning_rate=0.5, max_depth=10, min_samples_leaf=50,
                           min_samples_split=1800, random_state=0)
```

Feature Importance

```
# Extract Importance
importance = pd.DataFrame({'feature': ct.get_feature_names_out(), 'importance' : np.round(cl
importance.sort_values('importance', ascending=False, inplace = True)

print(importance)
```

```
                                    feature  importance
18           remainder__last_90_days_purchase       0.468
2                          encoder__channel_2       0.328
10  encoder__fashion_news_frequency_Regularly       0.042
13                         remainder__Divided       0.034
0                                  sc__age       0.032
14                      remainder__Ladieswear       0.030
6      encoder__club_member_status_PRE-CREATE       0.015
15                        remainder__Menswear       0.013
16                           remainder__Sport       0.010
17           remainder__last_30_days_purchase       0.009
```

```
12                    remainder__Baby/Children       0.006
1                           encoder__channel_1       0.005
4           encoder__club_member_status_ACTIVE       0.002
3                           encoder__channel_3       0.002
9         encoder__fashion_news_frequency_NONE       0.002
5        encoder__club_member_status_LEFT CLUB       0.000
7              encoder__club_member_status_nan       0.000
8      encoder__fashion_news_frequency_Monthly       0.000
11         encoder__fashion_news_frequency_nan       0.000
```

```
print(accuracy_score(clf.predict(X_train),Y_train))
print(accuracy_score(clf.predict(X_val),Y_val))
```

```
0.6965349263527633
0.688897947881734
```

Model Comparison

Decision Tree

Pros:

- A decision tree does not require normalization of data and nor does it require scaling of the data.
- Missing values in the data also do not affect the process of building a decision tree to any considerable extent.
- Compared to other algorithms (like Gradient Boosting), decision trees requires less effort for data preparation during pre-processing.

Cons:

- One of the major problems with Decision Tree is that a small change in the data can cause a large change in the structure of the decision tree causing instability.
- Decision tree often involves higher time to train the model.
- These trees are prone to overfitting.

Random Forest Classifier

Pros:

- Random Forests are prone and robust to outliers.
- They have lower risks of overfitting from which decision trees suffer easily.
- Random Forests can run efficiently on large data sets.

Cons:

- Random forests are found to be biased while dealing with categorical variables.
- Random forests are prone to slow training of the model.

Gradient Boosting

Pros:

- Gradient Boosting models generate more accurate results than other models.
- They train faster especially on larger datasets.

- Gradient Boosting models handle missing values natively and automatically.

Cons:

- GB models are prone to overfitting. However, this can be solved by applying L1 and L2 regularization techniques or building and testing the model with a low learning rate as well.
- They sometimes can be hard to interpret and also becomes expensive to compute.

Observations and Findings

1.Bagging or Random Forest Machine Learning creates a number of models at the same time separately. Each of the models is independent of the other. We can still improve our model accuracy using Boosting. Boosting, unlike Bagging, creates models one by one. Firstly, the first model learns from the training data. The second model then also learns from the same training dataset and the mistakes done by the first model. The third model, again, learns from the same training dataset and the mistakes of the previous model. In the case of our data we can observe that the accuaracy of the model has increased when we have implemented Bagging or Boosting methods over the less iterative methods like Decision Tree.

2.While analyzing the confusion matrix we can see that metrics like Precision and Recall are narrating that there is more predictor variable's that can be derived from the data since the current crop are yielding a lower False Positive count. This observation will help us in going back to the Model data and figuring out more predictor variables that can be derived.