GROUP 8 : CUSTOMER CHURN IN BANK

1)Anuj Suchchal

2)Jagdeep Chawla

3)Rajeev Sharma

4)Sourabh Mahapatro

https://github.com/anujsuchchal/msds422b_group_8

Problem statement :

Bank XYZ has been observing a lot of customers closing their accounts or switching to competitor banks over the past couple of quarters. As such, this has caused a huge dent in the quarterly revenues and might drastically affect annual revenues for the ongoing financial year, causing stocks to plunge and market cap to reduce by X %. A team of business, product, engineering and data science folks have been put together to arrest this slide.

Objective : Can we build a model to predict, with a reasonable accuracy, the customers who are going to churn in the near future? Being able to accurately estimate when they are going to churn will be an added bonus

Definition of churn : A customer having closed all their active accounts with the bank is said to have churned. Churn can be defined in other ways as well, based on the context of the problem. A customer not transacting for 6 months or 1 year can also be defined as to have churned, based on the business requirements

From a Biz team/Product Manager's perspective :

(1) Business goal : Arrest slide in revenues or loss of active bank customers

(2) Identify data source : Transactional systems, event-based logs, Data warehouse (MySQL DBs, Redshift/AWS), Data Lakes, NoSQL DBs

(3) Audit for data quality : De-duplication of events/transactions, Complete or partial absence of data for chunks of time in between, Obscuring PII (personal identifiable information) data

(4) Define business and data-related metrics : Tracking of these metrics over time, probably through some intuitive visualizations

```
(i) Business metrics : Churn rate (month-on-month, weekly/quarterly), Trend of avg. number of products per customer,
    %age of dormant customers, Other such descriptive metrics

(ii) Data-related metrics : F1-score, Recall, Precision
     Recall = TP/(TP + FN)
     Precision = TP/(TP + FP)
     F1-score = Harmonic mean of Recall and Precision
     where, TP = True Positive, FP = False Positive and FN = False Negative
```

(5) Prediction model output format : Since this is not going to be an online model, it doesn't require deployment. Instead, periodic (monthly/quarterly) model runs could be made and the list of customers, along with their propensity to churn shared with the business (Sales/Marketing) or Product team

(6) Action to be taken based on model's output/insights : Based on the output obtained from Data Science team as above, various business interventions can be made to save the customer from getting churned. Customer-centric bank offers, getting in touch with customers to address grievances etc. Here, also Data Science team can help with basic EDA to highlight different customer groups/segments and the appropriate intervention to be applied against them

Collaboration with Engineering and DevOps :

(1) Application deployment on production servers (In the context of this problem statement, not required)

(2) [DevOps] Monitoring the scale aspects of model performance over time (Again, not required, in this case)

How to set the target/goal for the metrics?

- Data science-related metrics :
  - Recall : >70%
  - Precision : >70%
  - F1-score : >70%

- Business metrics : Usually, it's top down. But a good practice is to consider it to make atleast half the impact of the data science metric. For e.g., If we take Recall target as **70%** which means correctly identifying 70% of customers who's going to churn in the near future, we can expect that due to business intervention (offers, getting in touch with customers etc.), 50% of the customers can be saved from being churned, which means atleast a **35%** improvement in Churn Rate

Show me the code!

In [1]:
```
import warnings
warnings.filterwarnings("ignore")
```

In [ ]:
```
!pip install ipython==7.22.0
!pip install joblib==1.0.1
!pip install lightgbm==3.3.1
!pip install matplotlib==3.3.4
!pip install numpy==1.20
!pip install pandas==1.3.5
!pip install scikit_learn==0.24.1
!pip install seaborn==0.11.1
!pip install shap==0.40.0
!pip install xgboost==1.5.1"""
```

In [ ]:
```
!pip install scikit_learn==0.24.1
```

In [3]:
```
%matplotlib inline
```

In [4]:
```
## Import required libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

In [5]:

```python
## Get multiple outputs in the same cell
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"

## Ignore all warnings
import warnings
warnings.filterwarnings('ignore')
warnings.filterwarnings(action='ignore', category=DeprecationWarning)
```

In [6]:

```python
## Display all rows and columns of a dataframe instead of a truncated version
from IPython.display import display
pd.set_option('display.max_columns', None)
pd.set_option('display.max_rows', None)
```

In [7]:

```python
## Reading the dataset
# This might be present in S3, or obtained through a query on a database
df = pd.read_csv("Churn_Modelling.csv")
```

In [8]:

```python
df.shape
```

Out[8]:

```
(10000, 14)
```

In [9]:

```python
df.head(10).T
```

Out[9]:

	0	1	2	3	4	5	6	7	8
RowNumber	1	2	3	4	5	6	7	8	9
CustomerId	15634602	15647311	15619304	15701354	15737888	15574012	15592531	15656148	15792365
Surname	Hargrave	Hill	Onio	Boni	Mitchell	Chu	Bartlett	Obinna	He
CreditScore	619	608	502	699	850	645	822	376	501
Geography	France	Spain	France	France	Spain	Spain	France	Germany	France
Gender	Female	Female	Female	Female	Female	Male	Male	Female	Male
Age	42	41	42	39	43	44	50	29	44
Tenure	2	1	8	1	2	8	7	4	4
Balance	0.0	83807.86	159660.8	0.0	125510.82	113755.78	0.0	115046.74	142051.07
NumOfProducts	1	1	3	2	1	2	2	4	2
HasCrCard	1	0	1	0	1	1	1	1	0
IsActiveMember	1	1	0	0	1	0	1	0	1
EstimatedSalary	101348.88	112542.58	113931.57	93826.63	79084.1	149756.71	10062.8	119346.88	74940.5
Exited	1	0	1	0	0	1	0	1	0

Basic EDA

In [10]:

```python
df.describe() # Describe all numerical columns
df.describe(include = ['O']) # Describe all non-numerical/categorical columns
```

Out[10]:

	RowNumber	CustomerId	CreditScore	Age	Tenure	Balance	NumOfProducts	Has(

	RowNumber	CustomerId	CreditScore	Age	Tenure	Balance	NumOfProducts	Has(
count	10000.00000	1.000000e+04	10000.000000	10000.000000	10000.000000	10000.000000	10000.000000	10000
mean	5000.50000	1.569094e+07	650.528800	38.921800	5.012800	76485.889288	1.530200	0
std	2886.89568	7.193619e+04	96.653299	10.487806	2.892174	62397.405202	0.581654	0
min	1.00000	1.556570e+07	350.000000	18.000000	0.000000	0.000000	1.000000	0
25%	2500.75000	1.562853e+07	584.000000	32.000000	3.000000	0.000000	1.000000	0
50%	5000.50000	1.569074e+07	652.000000	37.000000	5.000000	97198.540000	1.000000	1
75%	7500.25000	1.575323e+07	718.000000	44.000000	7.000000	127644.240000	2.000000	1
max	10000.00000	1.581569e+07	850.000000	92.000000	10.000000	250898.090000	4.000000	1

Out[10]:

	Surname	Geography	Gender
count	10000	10000	10000
unique	2932	3	2
top	Smith	France	Male
freq	32	5014	5457

In [11]:

```
## Checking number of unique customers in the dataset
df.shape[0], df.CustomerId.nunique()
```

Out[11]:

```
(10000, 10000)
```

In [12]:

```
df_t = df.groupby(['Surname']).agg({'RowNumber':'count', 'Exited':'mean'}
                                  ).reset_index().sort_values(by='RowNumber', ascending=Fa
```

In [13]:

```
df_t.head()
```

Out[13]:

	Surname	RowNumber	Exited
2473	Smith	32	0.281250
1689	Martin	29	0.310345
2389	Scott	29	0.103448
2751	Walker	28	0.142857
336	Brown	26	0.192308

In [14]:

```
df.Geography.value_counts(normalize=True)
```

Out[14]:

```
France     0.5014
Germany    0.2509
Spain      0.2477
Name: Geography, dtype: float64
```

Conclusion

- Discard row number
- Discard CustomerID as well, since it doesn't convey any extra info. Each row pertains to a unique customer

- Based on the above, columns/features can be segregated into non-essential, numerical, categorical and target variables

In general, CustomerID is a very useful feature on the basis of which we can calculate a lot of user-centric features. Here, the dataset is not sufficient to calculate any extra customer features

In [15]:

```
## Separating out different columns into various categories as defined above
target_var = ['Exited']
cols_to_remove = ['RowNumber', 'CustomerId']
num_feats = ['CreditScore', 'Age', 'Tenure', 'Balance', 'NumOfProducts', 'EstimatedSalary'
cat_feats = ['Surname', 'Geography', 'Gender', 'HasCrCard', 'IsActiveMember']
```

Among these, Tenure and NumOfProducts are ordinal variables. HasCrCard and IsActiveMember are actually binary categorical variables.

In [16]:

```
## Separating out target variable and removing the non-essential columns
y = df[target_var].values
df.drop(cols_to_remove, axis=1, inplace=True)
```

Questioning the data :

- No date/time column. A lot of useful features can be built using date/time columns
- When was the data snapshot taken? There are certain customer features like : Balance, Tenure, NumOfProducts, EstimatedSalary, which will have different values across time
- Are all these values/features pertaining to the same single date or spread across multiple dates?
- How frequently are customer features updated?
- Will it be possible to have the values of these features over a period of time as opposed to a single, snapshot date?
- Some customers who have exited still have balance in their account, or a non-zero NumOfProducts. Does this mean they have churned only from a specific product and not the entire bank, or are these snapshots of just before they churned?
- Some features like, number and kind of transactions, can help us estimate the degree of activity of the customer, instead of trusting the binary variable IsActiveMember
- Customer transaction patterns can also help us ascertain whether the customer has actually churned or not. For example, a customer might transact daily/weekly vs a customer who transacts annuallly

  Here, the objective is to understand the data and distill the problem statement and the stated goal further. In the process, if more data/context can be obtained, that adds to the end result of the model performance

Separating out train-test-valid sets

Since this is the only data available to us, we keep aside a holdout/test set to evaluate our model at the very end in order to estimate our chosen model's performance on unseen data / new data.

A validation set is also created which we'll use in our baseline models to evaluate and tune our models

In [17]:

```
from sklearn.model_selection import train_test_split
```

In [18]:

```
## Keeping aside a test/holdout set
df_train_val, df_test, y_train_val, y_test = train_test_split(df, y.ravel(), test_size = (
```

```
## Splitting into train and validation set
df_train, df_val, y_train, y_val = train_test_split(df_train_val, y_train_val, test_size =
```

In [19]:
```
df_train.shape, df_val.shape, df_test.shape, y_train.shape, y_val.shape, y_test.shape
np.mean(y_train), np.mean(y_val), np.mean(y_test)
```

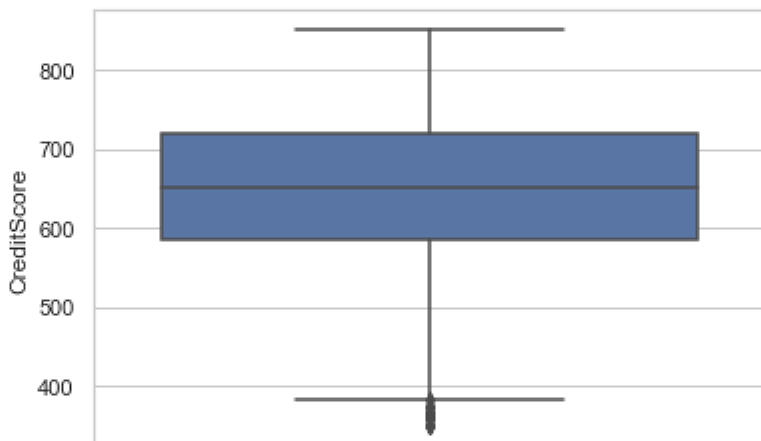
Out[19]: ((7920, 12), (1080, 12), (1000, 12), (7920,), (1080,), (1000,))

Out[19]: (0.20303030303030303, 0.22037037037037038, 0.191)

Univariate plots of numerical variables in training set

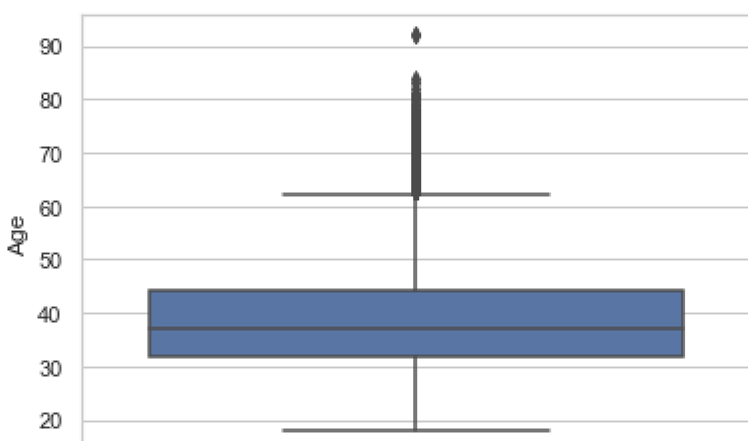
In [20]:
```
## CreditScore
sns.set(style="whitegrid")
sns.boxplot(y = df_train['CreditScore'])
```

Out[20]: <AxesSubplot:ylabel='CreditScore'>



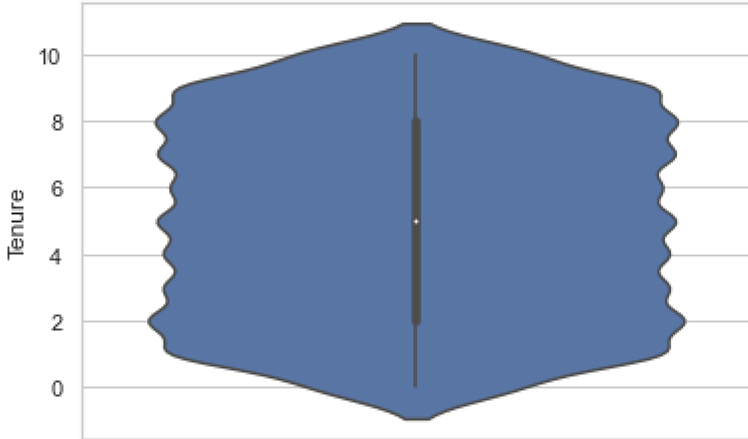
In [21]:
```
## Age
sns.boxplot(y = df_train['Age'])
```

Out[21]: <AxesSubplot:ylabel='Age'>



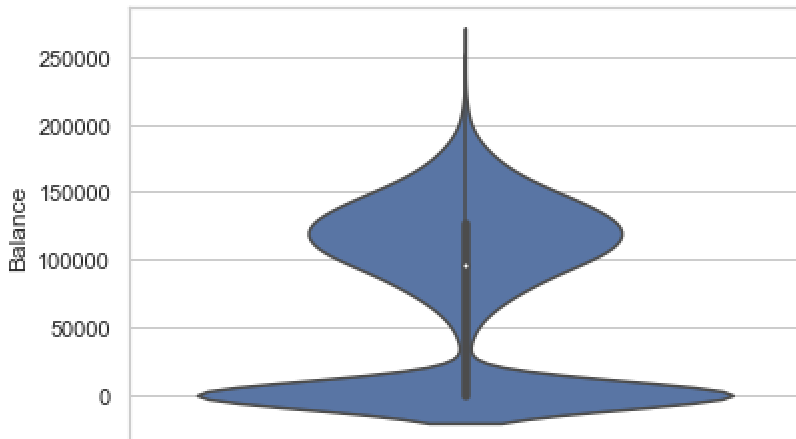
In [22]:
```
## Tenure
sns.violinplot(y = df_train.Tenure)
```

Out[22]: <AxesSubplot:ylabel='Tenure'>



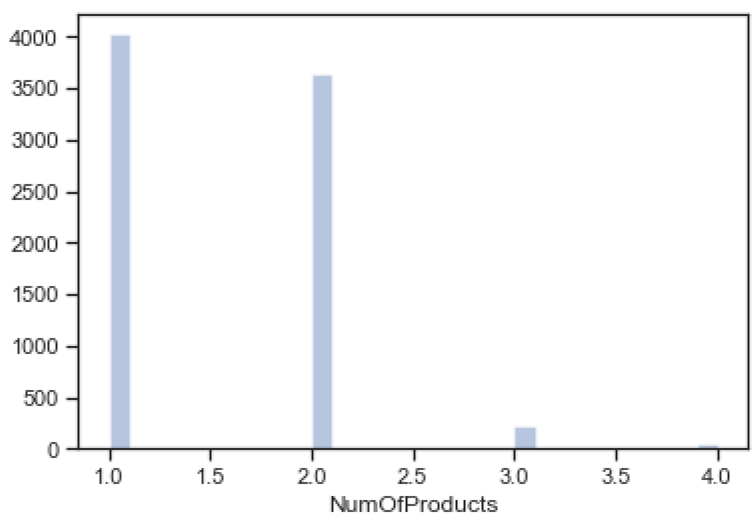
In [23]:
```
## Balance
sns.violinplot(y = df_train['Balance'])
```

Out[23]: <AxesSubplot:ylabel='Balance'>



In [24]:
```
## NumOfProducts
sns.set(style = 'ticks')
sns.distplot(df_train.NumOfProducts, hist=True, kde=False)
```

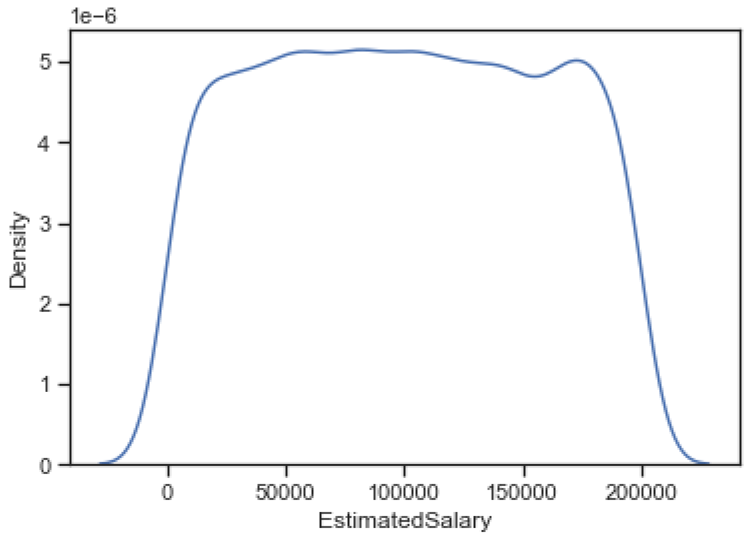
Out[24]: <AxesSubplot:xlabel='NumOfProducts'>



In [25]:
```
## EstimatedSalary
sns.kdeplot(df_train.EstimatedSalary)
```

<AxesSubplot:xlabel='EstimatedSalary', ylabel='Density'>

Out[25]:



- From the univariate plots, we get an indication that *EstimatedSalary* , being uniformly distributed, might not turn out to be an important predictor
- Similarly, for *NumOfProducts* , there are predominantly only two values (1 and 2). Hence, its chances of being a strong predictor is also very unlikely
- On the other hand, *Balance* has a multi-modal distribution. We'll see a little later if that helps in separation of the two target classes

Missing values and outlier treatment

Outliers

- Can be observed from univariate plots of different features

- Outliers can either be logically improbable (as per the feature definition) or just an extreme value as compared to the feature distribution

- As part of outlier treatment, the particular row containing the outlier can be removed from the training set, provided they do not form a significant chunk of the dataset (< 0.5-1%)

- In cases where the value of outlier is logically faulty, e.g. negative Age or CreditScore > 900, the particular record can be replaced with mean of the feature or the nearest among min/max logical value of the feature

Outliers in numerical features can be of a very high/low value, lying in the top 1% or bottom 1% of the distribution or values which are not possible as per the feature definition.

Outliers in categorical features are usually levels with a very low frequency/no. of samples as compared to other categorical levels.

No outliers observed in any feature of this dataset

Is outlier treatment always required ?

No, Not all ML algorithms are sensitive to outliers. Algorithms like linear/logistic regression are sensitive to outliers.

Tree algorithms, kNN, clustering algorithms etc. are in general, robust to outliers

Outliers affect metrics such as mean, std. deviation

Missing values

In [26]:

```
## No missing values!
df_train.isnull().sum()
```

Out[26]:

```
Surname            0
CreditScore        0
Geography          0
Gender             0
Age                0
Tenure             0
Balance            0
NumOfProducts      0
HasCrCard          0
IsActiveMember     0
EstimatedSalary    0
Exited             0
dtype: int64
```

No missing values present in this dataset. Can also be observed from df.describe() commands. However, most real-world datasets might have missing values. A couple of things which can be done in such cases :

- If the column/feature has too many missing values, it can be dropped as it might not add much relevance to the data
- If there a few missing values, the column/feature can be imputed with its summary statistics (mean/median/mode) and/or numbers like 0, -1 etc. which add value depending on the data and context. For example, say, BalanceInAccount.

In [27]:

```
## Making all changes in a temporary dataframe
df_missing = df_train.copy()
```

In [28]:

```
## Modify few records to add missing values/outliers

# Introducing 10% nulls in Age
na_idx = df_missing.sample(frac = 0.1).index
df_missing.loc[na_idx, 'Age'] = np.NaN

# Introducing 30% nulls in Geography
na_idx = df_missing.sample(frac = 0.3).index
df_missing.loc[na_idx, 'Geography'] = np.NaN

# Introducing 5% nulls in HasCrCard
na_idx = df_missing.sample(frac = 0.05).index
df_missing.loc[na_idx, 'HasCrCard'] = np.NaN
```

In [29]:

```
df_missing.isnull().sum()/df_missing.shape[0]
```

Out[29]:

```
Surname            0.00
CreditScore        0.00
Geography          0.30
Gender             0.00
Age                0.10
Tenure             0.00
Balance            0.00
NumOfProducts      0.00
HasCrCard          0.05
IsActiveMember     0.00
EstimatedSalary    0.00
```

```
Exited             0.00
dtype: float64
```

In [30]:
```python
## Calculating mean statistics
age_mean = df_missing.Age.mean()
```

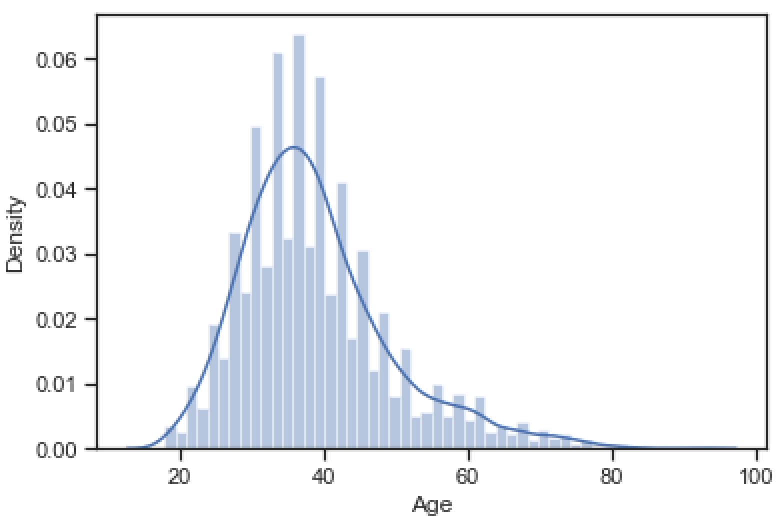
In [31]:
```python
age_mean
```

Out[31]:
```
38.84890572390572
```

In [32]:
```python
# Filling nulls in Age by mean value (numeric column)

#df_missing.Age.fillna(age_mean, inplace=True)

df_missing['Age'] = df_missing.Age.apply(lambda x: int(np.random.normal(age_mean,3)) if np
```

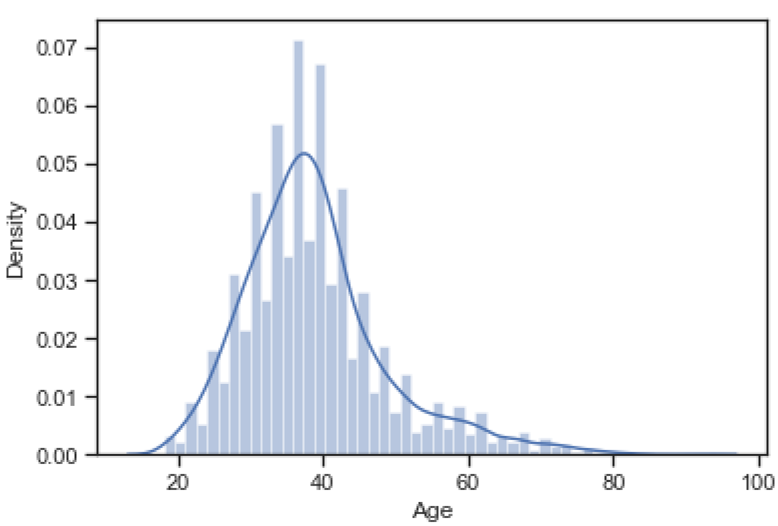
In [33]:
```python
## Distribution of "Age" feature before data imputation
sns.distplot(df_train.Age)
```

Out[33]:
```
&lt;AxesSubplot:xlabel='Age', ylabel='Density'&gt;
```



In [34]:
```python
## Distribution of "Age" feature after data imputation
sns.distplot(df_missing.Age)
```

Out[34]:
```
&lt;AxesSubplot:xlabel='Age', ylabel='Density'&gt;
```



In [35]:
```
# Filling nulls in Geography (categorical feature with a high %age of missing values)
geog_fill_value = 'UNK'
df_missing.Geography.fillna(geog_fill_value, inplace=True)

# Filling nulls in HasCrCard (boolean feature) - 0 for few nulls, -1 for lots of nulls
df_missing.HasCrCard.fillna(0, inplace=True)
```

In [36]:
```
df_missing.Geography.value_counts(normalize=True)
```

Out[36]:
```
France     0.353030
UNK        0.300000
Spain      0.176894
Germany    0.170076
Name: Geography, dtype: float64
```

In [37]:
```
df_missing.isnull().sum()/df_missing.shape[0]
```

Out[37]:
```
Surname            0.0
CreditScore        0.0
Geography          0.0
Gender             0.0
Age                0.0
Tenure             0.0
Balance            0.0
NumOfProducts      0.0
HasCrCard          0.0
IsActiveMember     0.0
EstimatedSalary    0.0
Exited             0.0
dtype: float64
```

Categorical variable encoding

As a rule of thumb, we can consider using :

1. Label Encoding ---> Binary categorical variables and Ordinal variables
2. One-Hot Encoding ---> Non-ordinal categorical variables with low to mid cardinality (< 5-10 levels)
3. Target encoding ---> Categorical variables with > 10 levels

- HasCrCard and IsActiveMember are already label encoded
- For Gender, a simple Label encoding should be fine.
- For Geography, since there are 3 levels, OneHotEncoding should do the trick
- For Surname, we'll try Target/Frequency Encoding

Label Encoding for binary variables

In [38]:
```
## The non-sklearn method
df_train['Gender_cat'] = df_train.Gender.astype('category').cat.codes
```

In [39]:
```
df_train.sample(10)
```

Out[39]:

	Surname	CreditScore	Geography	Gender	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsActiveM
3114	O'Donnell	619	France	Female	40	10	0.00	1	1	
9727	Ferri	530	France	Female	45	1	0.00	1	0	

	Surname	CreditScore	Geography	Gender	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsActiveM
4806	Lung	697	France	Female	33	1	87347.70	1	1	
7208	Degtyarev	547	Germany	Male	25	4	98141.57	2	1	
3150	Olisaemeka	573	Germany	Female	35	9	206868.78	2	0	
4166	Ma	850	Spain	Female	45	5	174088.30	4	1	
5113	Pai	754	France	Female	47	1	185513.67	1	1	
584	Begum	647	Germany	Female	51	1	119741.77	2	0	
133	Alekseeva	686	France	Male	25	1	0.00	2	0	
982	Clark	668	France	Male	32	7	0.00	2	1	

In [40]:
```
df_train.drop('Gender_cat', axis=1, inplace = True)
```

In [41]:
```
## The sklearn method
from sklearn.preprocessing import LabelEncoder
```

In [42]:
```
le = LabelEncoder()
```

We fit only on train dataset as that's the only data we'll assume we have. We'll treat validation and test sets as unseen data. Hence, they can't be used for fitting the encoders.

In [43]:
```
## Label encoding of Gender variable
df_train['Gender'] = le.fit_transform(df_train['Gender'])
```

In [44]:
```
le_name_mapping = dict(zip(le.classes_, le.transform(le.classes_)))
le_name_mapping
```

Out[44]:
```
{'Female': 0, 'Male': 1}
```

In [45]:
```
## What if Gender column has new values in test or val set?
le.transform([['Male']])
#le.transform([['ABC']])
```

Out[45]:
```
array([1])
```

In [46]:
```
pd.Series(['ABC']).map(le_name_mapping)
```

Out[46]:
```
0    NaN
dtype: float64
```

In [47]:
```
## Encoding Gender feature for validation and test set
df_val['Gender'] = df_val.Gender.map(le_name_mapping)
df_test['Gender'] = df_test.Gender.map(le_name_mapping)

## Filling missing/NaN values created due to new categorical levels
df_val['Gender'].fillna(-1, inplace=True)
df_test['Gender'].fillna(-1, inplace=True)
```

In [48]:
```
df_train.Gender.unique(), df_val.Gender.unique(), df_test.Gender.unique()
```
Out[48]:
```
(array([1, 0]), array([1, 0]), array([1, 0]))
```

One-Hot encoding for categorical variables with multiple levels

In [49]:
```
## The non-sklearn method
t = pd.get_dummies(df_train, prefix_sep = "_", columns = ['Geography'])
t.head()
```
Out[49]:

	Surname	CreditScore	Gender	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsActiveMember	Estim
4562	Yermakova	678	1	36	1	117864.85	2	1	0	
6498	Warlow-Davies	613	0	27	5	125167.74	1	1	0	
6072	Fu	628	1	45	9	0.00	2	1	1	
5813	Shih	513	1	30	5	0.00	2	1	0	
7407	Mahmood	639	1	22	4	0.00	2	1	0	

In [50]:
```
### Dropping dummy column
t.drop(['Geography_France'], axis=1, inplace=True)
t.head()
```
Out[50]:

	Surname	CreditScore	Gender	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsActiveMember	Estim
4562	Yermakova	678	1	36	1	117864.85	2	1	0	
6498	Warlow-Davies	613	0	27	5	125167.74	1	1	0	
6072	Fu	628	1	45	9	0.00	2	1	1	
5813	Shih	513	1	30	5	0.00	2	1	0	
7407	Mahmood	639	1	22	4	0.00	2	1	0	

In [51]:
```
## The sklearn method
from sklearn.preprocessing import LabelEncoder, OneHotEncoder
```
In [52]:
```
le_ohe = LabelEncoder()
ohe = OneHotEncoder(handle_unknown = 'ignore', sparse=False)
```
In [53]:
```
enc_train = le_ohe.fit_transform(df_train.Geography).reshape(df_train.shape[0],1)
enc_train.shape
np.unique(enc_train)
```
Out[53]:
```
(7920, 1)
```
Out[53]:
```
array([0, 1, 2])
```
In [54]:
```
ohe_train = ohe.fit_transform(enc_train)
ohe_train
```
```
array([[0., 1., 0.],
```

Out[54]:
```
        [1., 0., 0.],
        [1., 0., 0.],
        ...,
        [1., 0., 0.],
        [0., 1., 0.],
        [0., 1., 0.]])
```

In [55]:
```
le_ohe_name_mapping = dict(zip(le_ohe.classes_, le_ohe.transform(le_ohe.classes_)))
le_ohe_name_mapping
```

Out[55]:
```
{'France': 0, 'Germany': 1, 'Spain': 2}
```

In [56]:
```
## Encoding Geography feature for validation and test set
enc_val = df_val.Geography.map(le_ohe_name_mapping).ravel().reshape(-1,1)
enc_test = df_test.Geography.map(le_ohe_name_mapping).ravel().reshape(-1,1)

## Filling missing/NaN values created due to new categorical levels
enc_val[np.isnan(enc_val)] = 9999
enc_test[np.isnan(enc_test)] = 9999
```

In [57]:
```
np.unique(enc_val)
np.unique(enc_test)
```

Out[57]:
```
array([0, 1, 2])
```

Out[57]:
```
array([0, 1, 2])
```

In [58]:
```
ohe_val = ohe.transform(enc_val)
ohe_test = ohe.transform(enc_test)
```

In [59]:
```
### Show what happens when a new value is inputted into the OHE
ohe.transform(np.array([[9999]]))
```

Out[59]:
```
array([[0., 0., 0.]])
```

Adding the one-hot encoded columns to the dataframe and removing the original feature

In [60]:
```
cols = ['country_' + str(x) for x in le_ohe_name_mapping.keys()]
cols
```

Out[60]:
```
['country_France', 'country_Germany', 'country_Spain']
```

In [61]:
```
## Adding to the respective dataframes
df_train = pd.concat([df_train.reset_index(), pd.DataFrame(ohe_train, columns = cols)], ax
df_val = pd.concat([df_val.reset_index(), pd.DataFrame(ohe_val, columns = cols)], axis = 1
df_test = pd.concat([df_test.reset_index(), pd.DataFrame(ohe_test, columns = cols)], axis
```

In [62]:
```
print("Training set")
df_train.head()
print("\n\nValidation set")
df_val.head()
print("\n\nTest set")
df_test.head()
```

```
Training set
```

Out[62]:

	Surname	CreditScore	Geography	Gender	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsActiveMemb
0	Yermakova	678	Germany	1	36	1	117864.85	2	1	
1	Warlow-Davies	613	France	0	27	5	125167.74	1	1	
2	Fu	628	France	1	45	9	0.00	2	1	
3	Shih	513	France	1	30	5	0.00	2	1	
4	Mahmood	639	France	1	22	4	0.00	2	1	

```
Validation set
```

Out[62]:

	Surname	CreditScore	Geography	Gender	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsActiveMembe
0	Sun	757	France	1	36	7	144852.06	1	0	[illegible]
1	Russo	552	France	1	29	10	0.00	2	1	[illegible]
2	Munro	619	France	0	30	7	70729.17	1	1	[illegible]
3	Perkins	633	France	1	35	10	0.00	2	1	[illegible]
4	Aliyeva	698	Spain	1	38	10	95010.92	1	1	[illegible]

```
Test set
```

Out[62]:

	Surname	CreditScore	Geography	Gender	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsActiveMembe
0	Anderson	596	Germany	1	32	3	96709.07	2	0	[illegible]
1	Herring	623	France	1	43	1	0.00	2	1	[illegible]
2	Amechi	601	Spain	0	44	4	0.00	2	1	[illegible]
3	Liang	506	Germany	1	59	8	119152.10	2	1	[illegible]
4	Chuang	560	Spain	0	27	7	124995.98	1	1	[illegible]

In [63]:

```
## Drop the Geography column
df_train.drop(['Geography'], axis = 1, inplace=True)
df_val.drop(['Geography'], axis = 1, inplace=True)
df_test.drop(['Geography'], axis = 1, inplace=True)
```

In [ ]:

In [ ]:

Target encoding

Target encoding is generally useful when dealing with categorical variables of high cardinality (high number of levels).

Here, we'll encode the column 'Surname' (which has 2932 different values!) with the mean of target variable for that level

In [64]:

```
df_train.head()
```

Out[64]:

	Surname	CreditScore	Gender	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsActiveMember	Estimate
0	Yermakova	678	1	36	1	117864.85	2	1	0	2
1	Warlow-Davies	613	0	27	5	125167.74	1	1	0	19
2	Fu	628	1	45	9	0.00	2	1	1	9
3	Shih	513	1	30	5	0.00	2	1	0	16
4	Mahmood	639	1	22	4	0.00	2	1	0	2

In [65]:

```
means = df_train.groupby(['Surname']).Exited.mean()
means.head()
```

Out[65]:

```
Surname
Abazu       0.00
Abbie       0.00
Abbott      0.25
Abdullah    1.00
Abdulov     0.00
Name: Exited, dtype: float64
```

In [66]:

```
global_mean = y_train.mean()
global_mean
```

Out[66]:

```
0.20303030303030303
```

In [67]:

```
## Creating new encoded features for surname - Target (mean) encoding
df_train['Surname_mean_churn'] = df_train.Surname.map(means)
df_train['Surname_mean_churn'].fillna(global_mean, inplace=True)
```

But, the problem with Target encoding is that it might cause data leakage, as we are considering feedback from the target variable while computing any summary statistic.

A solution is to use a modified version : Leave-one-out Target encoding.

In this, for a particular data point or row, the mean of the target is calculated by considering all rows in the same categorical level except itself. This mitigates data leakage and overfitting to some extent.

Mean for a category, $\mathbf{m_c = S_c / n_c}$ ..... (1)

What we need to find is the mean excluding a single sample. This can be expressed as : $\mathbf{m_i = (S_c - t_i) / (n_c - 1)}$ ..... (2)

Using (1) and (2), we can get : $\mathbf{m_i = (n_c m_c - t_i) / (n_c - 1)}$

Here, S_c = Sum of target variable for category c

n_c = Number of rows in category c

t_i = Target value of the row whose encoding is being calculated

In [68]:

```
## Calculate frequency of each category
freqs = df_train.groupby(['Surname']).size()
freqs.head()
```

Out[68]:
```
Surname
Abazu       2
Abbie       1
Abbott      4
Abdullah    1
Abdulov     1
dtype: int64
```

In [69]:
```
## Create frequency encoding - Number of instances of each category in the data
df_train['Surname_freq'] = df_train.Surname.map(freqs)
df_train['Surname_freq'].fillna(0, inplace=True)
```

In [70]:
```
## Create Leave-one-out target encoding for Surname
df_train['Surname_enc'] = ((df_train.Surname_freq * df_train.Surname_mean_churn) - df_trai
df_train.head(10)
```

Out[70]:

	Surname	CreditScore	Gender	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsActiveMember	Estimated
0	Yermakova	678	1	36	1	117864.85	2	1	0	2
1	Warlow-Davies	613	0	27	5	125167.74	1	1	0	19
2	Fu	628	1	45	9	0.00	2	1	1	9
3	Shih	513	1	30	5	0.00	2	1	0	16
4	Mahmood	639	1	22	4	0.00	2	1	0	2
5	Miller	562	1	30	3	111099.79	2	0	0	14
6	Padovesi	635	1	43	5	78992.75	2	0	0	15
7	Edments	705	1	33	7	68423.89	1	1	1	6
8	Chan	694	1	42	8	133767.19	1	1	0	3
9	Matthews	711	1	26	9	128793.63	1	1	0	1

In [71]:
```
## Fill NaNs occuring due to category frequency being 1 or less
df_train['Surname_enc'].fillna((((df_train.shape[0] * global_mean) - df_train.Exited) / (d
df_train.head(10)
```

Out[71]:

	Surname	CreditScore	Gender	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsActiveMember	Estimated
0	Yermakova	678	1	36	1	117864.85	2	1	0	2
1	Warlow-Davies	613	0	27	5	125167.74	1	1	0	19
2	Fu	628	1	45	9	0.00	2	1	1	9
3	Shih	513	1	30	5	0.00	2	1	0	16
4	Mahmood	639	1	22	4	0.00	2	1	0	2
5	Miller	562	1	30	3	111099.79	2	0	0	14
6	Padovesi	635	1	43	5	78992.75	2	0	0	15
7	Edments	705	1	33	7	68423.89	1	1	1	6
8	Chan	694	1	42	8	133767.19	1	1	0	3
9	Matthews	711	1	26	9	128793.63	1	1	0	1

On validation and test set, we'll apply the normal Target encoding mapping as obtained from the training set

In [72]:

```
## Replacing by category means and new category levels by global mean
df_val['Surname_enc'] = df_val.Surname.map(means)
df_val['Surname_enc'].fillna(global_mean, inplace=True)

df_test['Surname_enc'] = df_test.Surname.map(means)
df_test['Surname_enc'].fillna(global_mean, inplace=True)
```

In [73]:

```
## Show that using LOO Target encoding decorrelates features
df_train[['Surname_mean_churn', 'Surname_enc', 'Exited']].corr()
```

Out[73]:

	Surname_mean_churn	Surname_enc	Exited
Surname_mean_churn	1.000000	0.54823	0.562677
Surname_enc	0.548230	1.00000	-0.026440
Exited	0.562677	-0.02644	1.000000

In [74]:

```
### Deleting the 'Surname' and other redundant column across the three datasets
df_train.drop(['Surname_mean_churn'], axis=1, inplace=True)
df_train.drop(['Surname_freq'], axis=1, inplace=True)
df_train.drop(['Surname'], axis=1, inplace=True)
df_val.drop(['Surname'], axis=1, inplace=True)
df_test.drop(['Surname'], axis=1, inplace=True)
```

In [75]:

```
df_train.head()
df_val.head()
df_test.head()
```

Out[75]:

	CreditScore	Gender	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsActiveMember	EstimatedSalary	Exi
0	678	1	36	1	117864.85	2	1	0	27619.06	
1	613	0	27	5	125167.74	1	1	0	199104.52	
2	628	1	45	9	0.00	2	1	1	96862.56	
3	513	1	30	5	0.00	2	1	0	162523.66	
4	639	1	22	4	0.00	2	1	0	28188.96	

Out[75]:

	CreditScore	Gender	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsActiveMember	EstimatedSalary	Exi
0	757	1	36	7	144852.06	1	0	0	130861.95	
1	552	1	29	10	0.00	2	1	0	12186.83	
2	619	0	30	7	70729.17	1	1	1	160948.87	
3	633	1	35	10	0.00	2	1	0	65675.47	
4	698	1	38	10	95010.92	1	1	1	105227.86	

Out[75]:

	CreditScore	Gender	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsActiveMember	EstimatedSalary	Exi
0	596	1	32	3	96709.07	2	0	0	41788.37	
1	623	1	43	1	0.00	2	1	1	146379.30	

	CreditScore	Gender	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsActiveMember	EstimatedSalary	Exi
2	601	0	44	4	0.00	2	1	0	58561.31	
3	506	1	59	8	119152.10	2	1	1	170679.74	
4	560	0	27	7	124995.98	1	1	1	114669.79	

Summarize : How to handle unknown categorical levels/values in unseen data in production?

- Use LabelEncoding, OneHotEncoding on training set and then save the mapping and apply on the test set. For missing values, use 0, -1 etc.
- Target/Frequency encoding : Create a mapping between each level and a statistical measure (mean, median, sum etc.) of the target from the training dataset. For the new categorical levels, impute the missing values suitably (can be 0, -1, or mean/mode/median)
- Leave-one-out or Cross fold Target encoding avoid data leakage and help in generalization of the model

In [ ]:

```

```

Bivariate analysis

In [76]:

```
## Check linear correlation (rho) between individual features and the target variable
corr = df_train.corr()
corr
```

Out[76]:

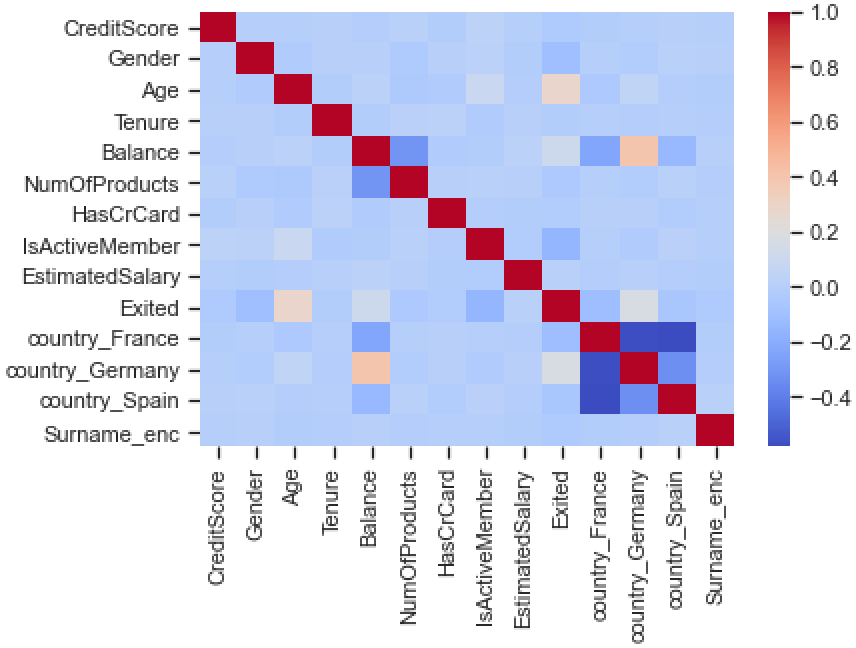
	CreditScore	Gender	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsActiveMemb
CreditScore	1.000000	0.000354	0.002099	0.005994	-0.001507	0.014110	-0.011868	0.0350
Gender	0.000354	1.000000	-0.024446	0.010749	0.009380	-0.026795	0.007550	0.0280
Age	0.002099	-0.024446	1.000000	-0.011384	0.027721	-0.033305	-0.019633	0.0935
Tenure	0.005994	0.010749	-0.011384	1.000000	-0.013081	0.018231	0.026148	-0.0212
Balance	-0.001507	0.009380	0.027721	-0.013081	1.000000	-0.304318	-0.021464	-0.0080
NumOfProducts	0.014110	-0.026795	-0.033305	0.018231	-0.304318	1.000000	0.007202	0.0148
HasCrCard	-0.011868	0.007550	-0.019633	0.026148	-0.021464	0.007202	1.000000	-0.0065
IsActiveMember	0.035057	0.028094	0.093573	-0.021263	-0.008085	0.014809	-0.006526	1.0000
EstimatedSalary	0.000358	-0.011007	-0.006827	0.010145	0.027247	0.009769	-0.008413	-0.0164
Exited	-0.028117	-0.102331	0.288221	-0.010660	0.113377	-0.039200	-0.013659	-0.1524
country_France	-0.009481	0.000823	-0.038881	0.000021	-0.231770	0.002991	0.005881	0.0021
country_Germany	0.003393	-0.018412	0.048764	-0.003131	0.405616	-0.015926	0.008197	-0.0205
country_Spain	0.007561	0.017361	-0.003648	0.003090	-0.136044	0.012388	-0.014934	0.0180
Surname_enc	-0.000739	0.008002	-0.010844	-0.006753	0.006925	-0.002020	-0.000551	0.0049

In [77]:

```
sns.heatmap(corr, cmap = 'coolwarm')
```

<AxesSubplot:>

Out[77]:



None of the features are highly correlated with the target variable. But some of them have slight linear associations with the target variable.

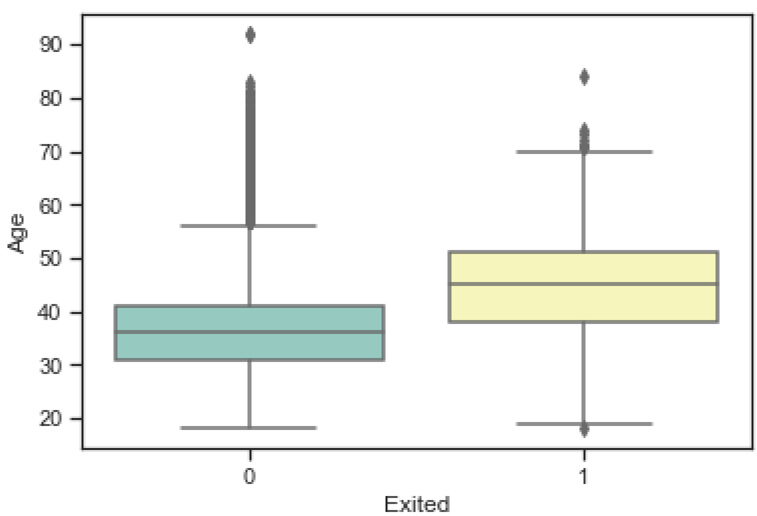
- Continuous features - Age, Balance
- Categorical variables - Gender, IsActiveMember, country_Germany, country_France

Individual features versus their distibution across target variable values

In [78]:

```
sns.boxplot(x = "Exited", y = "Age", data = df_train, palette="Set3")
```

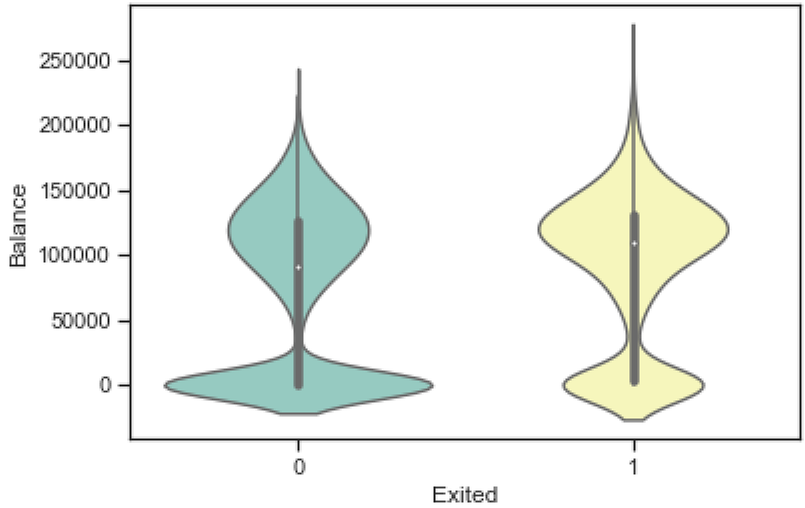
Out[78]: <AxesSubplot:xlabel='Exited', ylabel='Age'>



In [79]:

```
sns.violinplot(x = "Exited", y = "Balance", data = df_train, palette="Set3")
```

Out[79]: <AxesSubplot:xlabel='Exited', ylabel='Balance'>



In [80]:
```
# Check association of categorical features with target variable
cat_vars_bv = ['Gender', 'IsActiveMember', 'country_Germany', 'country_France']

for col in cat_vars_bv:
    df_train.groupby([col]).Exited.mean()
```

Out[80]:
```
Gender
0    0.248191
1    0.165511
Name: Exited, dtype: float64
```

Out[80]:
```
IsActiveMember
0    0.266285
1    0.143557
Name: Exited, dtype: float64
```

Out[80]:
```
country_Germany
0.0    0.163091
1.0    0.324974
Name: Exited, dtype: float64
```

Out[80]:
```
country_France
0.0    0.245877
1.0    0.160593
Name: Exited, dtype: float64
```

In [81]:
```
col = 'NumOfProducts'
df_train.groupby([col]).Exited.mean()
df_train[col].value_counts()
```

Out[81]:
```
NumOfProducts
1    0.273428
2    0.076881
3    0.825112
4    1.000000
Name: Exited, dtype: float64
```

Out[81]:
```
1    4023
2    3629
3     223
4      45
Name: NumOfProducts, dtype: int64
```

In [ ]:

In [ ]:

Some basic feature engineering

In [82]:
```
df_train.columns
```

Out[82]:
```
Index(['CreditScore', 'Gender', 'Age', 'Tenure', 'Balance', 'NumOfProducts',
       'HasCrCard', 'IsActiveMember', 'EstimatedSalary', 'Exited',
       'country_France', 'country_Germany', 'country_Spain', 'Surname_enc'],
      dtype='object')
```

Creating some new features based on simple interactions between the existing features.

- Balance/NumOfProducts
- Balance/EstimatedSalary
- Tenure/Age
- Age * Surname_enc

In [83]:
```
eps = 1e-6

df_train['bal_per_product'] = df_train.Balance/(df_train.NumOfProducts + eps)
df_train['bal_by_est_salary'] = df_train.Balance/(df_train.EstimatedSalary + eps)
df_train['tenure_age_ratio'] = df_train.Tenure/(df_train.Age + eps)
df_train['age_surname_mean_churn'] = np.sqrt(df_train.Age) * df_train.Surname_enc
```

In [84]:
```
df_train.head()
```

Out[84]:

	CreditScore	Gender	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsActiveMember	EstimatedSalary	Exi
0	678	1	36	1	117864.85	2	1	0	27619.06	
1	613	0	27	5	125167.74	1	1	0	199104.52	
2	628	1	45	9	0.00	2	1	1	96862.56	
3	513	1	30	5	0.00	2	1	0	162523.66	
4	639	1	22	4	0.00	2	1	0	28188.96	

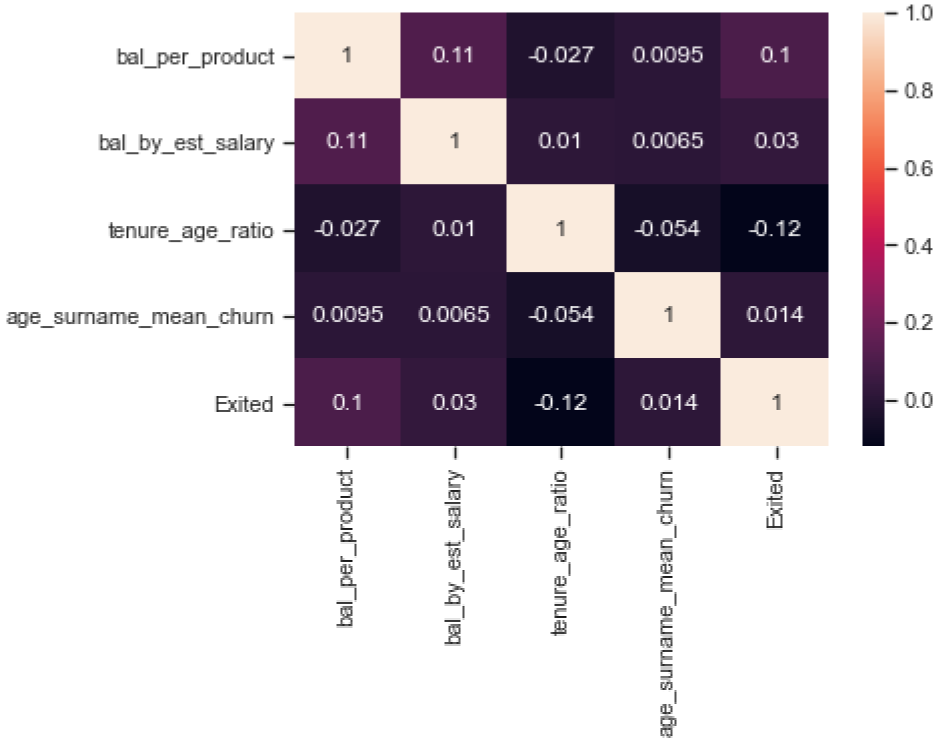
In [85]:
```
new_cols = ['bal_per_product','bal_by_est_salary','tenure_age_ratio','age_surname_mean_chu
```

In [86]:
```
## Ensuring that the new column doesn't have any missing values
df_train[new_cols].isnull().sum()
```

Out[86]:
```
bal_per_product           0
bal_by_est_salary         0
tenure_age_ratio          0
age_surname_mean_churn    0
dtype: int64
```

In [87]:
```
## Linear association of new columns with target variables to judge importance
sns.heatmap(df_train[new_cols + ['Exited']].corr(), annot=True)
```

Out[87]: <AxesSubplot:>



Out of the new features, ones with slight linear association/correlation are : bal_per_product and tenure_age_ratio

In [88]:

```
## Creating new interaction feature terms for validation set
eps = 1e-6

df_val['bal_per_product'] = df_val.Balance/(df_val.NumOfProducts + eps)
df_val['bal_by_est_salary'] = df_val.Balance/(df_val.EstimatedSalary + eps)
df_val['tenure_age_ratio'] = df_val.Tenure/(df_val.Age + eps)
df_val['age_surname_mean_churn'] = np.sqrt(df_val.Age) * df_val.Surname_enc
```

In [89]:

```
## Creating new interaction feature terms for test set
eps = 1e-6

df_test['bal_per_product'] = df_test.Balance/(df_test.NumOfProducts + eps)
df_test['bal_by_est_salary'] = df_test.Balance/(df_test.EstimatedSalary + eps)
df_test['tenure_age_ratio'] = df_test.Tenure/(df_test.Age + eps)
df_test['age_surname_mean_churn'] = np.sqrt(df_test.Age) * df_test.Surname_enc
```

In [ ]:

```
```

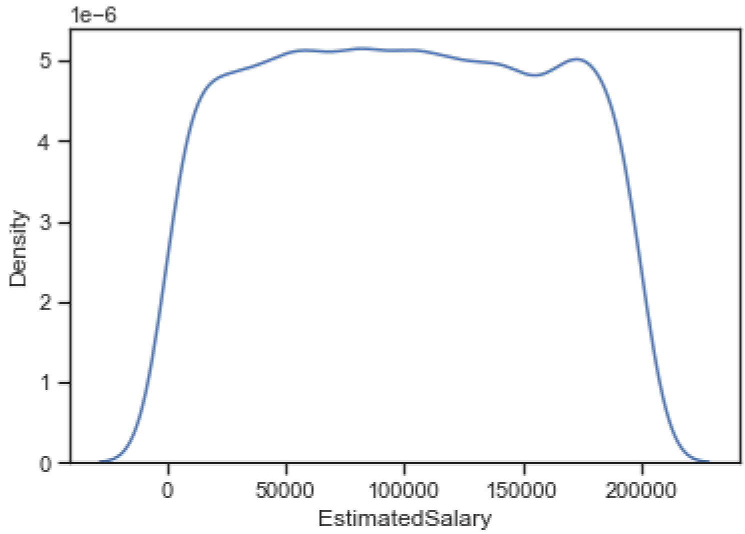
Feature scaling and normalization

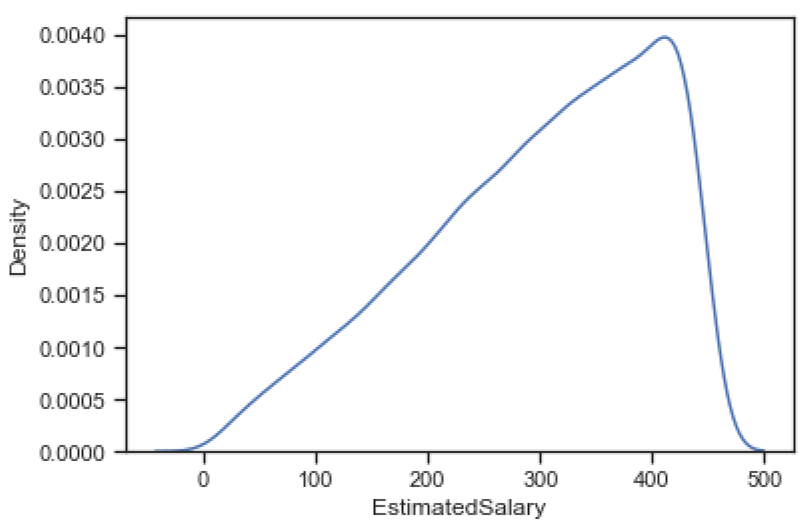
Different methods :

1. Feature transformations - Using log, log10, sqrt, pow
2. MinMaxScaler - Brings all feature values between 0 and 1
3. StandardScaler - Mean normalization. Feature values are an estimate of their z-score

- Why is scaling and normalization required ?
- How do we normalize unseen data?

Feature transformations

In [90]:
```
### Demo-ing feature transformations
sns.distplot(df_train.EstimatedSalary, hist=False)
```
Out[90]:
```
<AxesSubplot:xlabel='EstimatedSalary', ylabel='Density'>
```


In [91]:
```
sns.distplot(np.sqrt(df_train.EstimatedSalary), hist=False)
#sns.distplot(np.log10(1+df_train.EstimatedSalary), hist=False)
```
Out[91]:
```
<AxesSubplot:xlabel='EstimatedSalary', ylabel='Density'>
```


StandardScaler

In [92]:
```
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
```
In [93]:
```
df_train.columns
```
Out[93]:
```
Index(['CreditScore', 'Gender', 'Age', 'Tenure', 'Balance', 'NumOfProducts',
       'HasCrCard', 'IsActiveMember', 'EstimatedSalary', 'Exited',
       'country_France', 'country_Germany', 'country_Spain', 'Surname_enc',
       'bal_per_product', 'bal_by_est_salary', 'tenure_age_ratio',
       'age_surname_mean_churn'],
      dtype='object')
```
Scaling only continuous variables

In [94]:

```python
cont_vars = ['CreditScore', 'Age', 'Tenure', 'Balance', 'NumOfProducts', 'EstimatedSalary'
             , 'bal_by_est_salary', 'tenure_age_ratio', 'age_surname_mean_churn']
cat_vars = ['Gender', 'HasCrCard', 'IsActiveMember', 'country_France', 'country_Germany',
```

In [95]:

```python
## Scaling only continuous columns
cols_to_scale = cont_vars
```

In [96]:

```python
sc_X_train = sc.fit_transform(df_train[cols_to_scale])
```

In [97]:

```python
## Converting from array to dataframe and naming the respective features/columns
sc_X_train = pd.DataFrame(data = sc_X_train, columns = cols_to_scale)
sc_X_train.shape
sc_X_train.head()
```

Out[97]:

```
(7920, 11)
```

Out[97]:

	CreditScore	Age	Tenure	Balance	NumOfProducts	EstimatedSalary	Surname_enc	bal_per_product	bal
0	0.284761	-0.274383	-1.389130	0.670778	0.804059	-1.254732	-1.079210	-0.062389	
1	-0.389351	-1.128482	-0.004763	0.787860	-0.912423	1.731950	-1.079210	1.104840	
2	-0.233786	0.579716	1.379604	-1.218873	0.804059	-0.048751	0.094549	-1.100925	
3	-1.426446	-0.843782	-0.004763	-1.218873	0.804059	1.094838	0.505364	-1.100925	
4	-0.119706	-1.602981	-0.350855	-1.218873	0.804059	-1.244806	1.561746	-1.100925	

In [98]:

```python
## Mapping learnt on the continuous features
sc_map = {'mean':sc.mean_, 'std':np.sqrt(sc.var_)}
sc_map
```

Out[98]:

```
{'mean': array([6.50542424e+02, 3.88912879e+01, 5.01376263e+00, 7.60258447e+04,
        1.53156566e+00, 9.96616540e+04, 2.04321788e-01, 6.24727199e+04,
        2.64665647e+00, 1.38117689e-01, 1.26136416e+00]),
 'std': array([9.64231806e+01, 1.05374237e+01, 2.88940724e+00, 6.23738902e+04,
        5.82587032e-01, 5.74167173e+04, 1.89325378e-01, 5.67456646e+04,
        1.69816787e+01, 8.95590667e-02, 1.18715858e+00])}
```

In [99]:

```python
## Scaling validation and test sets by transforming the mapping obtained through the trai
sc_X_val = sc.transform(df_val[cols_to_scale])
sc_X_test = sc.transform(df_test[cols_to_scale])
```

In [100…

```python
## Converting val and test arrays to dataframes for re-usability
sc_X_val = pd.DataFrame(data = sc_X_val, columns = cols_to_scale)
sc_X_test = pd.DataFrame(data = sc_X_test, columns = cols_to_scale)
```

Feature scaling is important for algorithms like Logistic Regression and SVM. Not necessary for Tree-based models

In [ ]:

In [ ]:

Feature selection - RFE

Features shortlisted through EDA/manual inspection and bivariate analysis :

Age, Gender, Balance, NumOfProducts, IsActiveMember, the 3 country/Geography variables, bal per product, tenure age ratio

Now, let's see whether feature selection/elimination through RFE (Recursive Feature Elimination) gives us the same list of features, other extra features or lesser number of features.

To begin with, we'll feed all features to RFE + LogReg model.

In [101…

```
cont_vars
cat_vars
```

Out[101…

```
['CreditScore',
 'Age',
 'Tenure',
 'Balance',
 'NumOfProducts',
 'EstimatedSalary',
 'Surname_enc',
 'bal_per_product',
 'bal_by_est_salary',
 'tenure_age_ratio',
 'age_surname_mean_churn']
```

Out[101…

```
['Gender',
 'HasCrCard',
 'IsActiveMember',
 'country_France',
 'country_Germany',
 'country_Spain']
```

In [102…

```
## Creating feature-set and target for RFE model
y = df_train['Exited'].values
#X = pd.concat([df_train[cat_vars], sc_X_train[cont_vars]], ignore_index=True, axis = 1)
X = df_train[cat_vars + cont_vars]
X.columns = cat_vars + cont_vars
```

In [103…

```
from sklearn.feature_selection import RFE
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
```

In [104…

```
# for logistics regression
est = LogisticRegression()
num_features_to_select = 10
```

In [105…

```
# for decision trees
est_dt = DecisionTreeClassifier(max_depth = 4, criterion = 'entropy')
num_features_to_select = 10
```

In [106…

```
# for logistics regression
rfe = RFE(est, n_features_to_select=10)
rfe = rfe.fit(X.values, y)
print(rfe.support_)
print(rfe.ranking_)
```

```
[ True  True  True  True  True  True False  True False False  True False
  True False False  True False]
[1 1 1 1 1 1 4 1 3 6 1 8 1 7 5 1 2]
```

In [107…

```
# for decision trees
rfe_dt = RFE(est_dt, n_features_to_select=10)
rfe_dt = rfe_dt.fit(X.values, y)
print(rfe_dt.support_)
print(rfe_dt.ranking_)
```

```
[False False  True False  True False False  True False  True  True  True
 False  True  True  True  True]
[8 7 1 6 1 5 4 1 3 1 1 1 2 1 1 1 1]
```

In [108…

```
## Logistic Regression (Linear model)
mask = rfe.support_.tolist()
selected_feats = [b for a,b in zip(mask, X.columns) if a]
selected_feats
```

Out[108…

```
['Gender',
 'HasCrCard',
 'IsActiveMember',
 'country_France',
 'country_Germany',
 'country_Spain',
 'Age',
 'NumOfProducts',
 'Surname_enc',
 'tenure_age_ratio']
```

In [109…

```
## Decision Tree (Non-linear model)
mask = rfe_dt.support_.tolist()
selected_feats_dt = [b for a,b in zip(mask, X.columns) if a]
selected_feats_dt
```

Out[109…

```
['IsActiveMember',
 'country_Germany',
 'Age',
 'Balance',
 'NumOfProducts',
 'EstimatedSalary',
 'bal_per_product',
 'bal_by_est_salary',
 'tenure_age_ratio',
 'age_surname_mean_churn']
```

In [ ]:

In [ ]:

Baseline model : Logistic Regression

We'll train the linear models on the features selected through RFE

In [110…

```
from sklearn.linear_model import LogisticRegression
```

In [111…

```
## Importing relevant metrics
from sklearn.metrics import roc_auc_score, f1_score, recall_score, confusion_matrix, class
```

In [112…

```
selected_cat_vars = [x for x in selected_feats if x in cat_vars]
selected_cont_vars = [x for x in selected_feats if x in cont_vars]
```

In [113…

```
## Using categorical features and scaled numerical features
X_train = np.concatenate((df_train[selected_cat_vars].values, sc_X_train[selected_cont_var
X_val = np.concatenate((df_val[selected_cat_vars].values, sc_X_val[selected_cont_vars].val
X_test = np.concatenate((df_test[selected_cat_vars].values, sc_X_test[selected_cont_vars].

X_train.shape, X_val.shape, X_test.shape
```

Out[113…

```
((7920, 10), (1080, 10), (1000, 10))
```

- #### Solving class imbalance

In [114…

```
# Obtaining class weights based on the class samples imbalance ratio
_, num_samples = np.unique(y_train, return_counts = True)
weights = np.max(num_samples)/num_samples
weights
num_samples
```

Out[114…

```
array([1.        , 3.92537313])
```

Out[114…

```
array([6312, 1608], dtype=int64)
```

In [115…

```
weights_dict = dict()
class_labels = [0,1]
for a,b in zip(class_labels,weights):
    weights_dict[a] = b

weights_dict
```

Out[115…

```
{0: 1.0, 1: 3.925373134328358}
```

In [116…

```
## Defining model
lr = LogisticRegression(C = 1.0, penalty = 'l2', class_weight = weights_dict, n_jobs = -1)
```

In [117…

```
## Fitting model
lr.fit(X_train, y_train)
```

Out[117…

```
LogisticRegression(class_weight={0: 1.0, 1: 3.925373134328358}, n_jobs=-1)
```

In [118…

```
## Fitted model parameters
selected_cat_vars + selected_cont_vars

lr.coef_
lr.intercept_
```

Out[118…

```
['Gender',
 'HasCrCard',
 'IsActiveMember',
 'country_France',
 'country_Germany',
 'country_Spain',
 'Age',
 'NumOfProducts',
```

```
 'Surname_enc',
 'tenure_age_ratio']
```

Out[118…

```
array([[-0.5190172 , -0.06938782, -0.90843476, -0.33748839,  0.58664742,
        -0.24918718,  0.80999582, -0.05061525, -0.0659637 , -0.05143544]])
```

Out[118…

```
array([0.60235927])
```

In [119…

```
## Training metrics
roc_auc_score(y_train, lr.predict(X_train))
recall_score(y_train, lr.predict(X_train))
confusion_matrix(y_train, lr.predict(X_train))
print(classification_report(y_train, lr.predict(X_train)))
```

Out[119…

```
0.70684363354331
```

Out[119…

```
0.6983830845771144
```

Out[119…

```
array([[4515, 1797],
       [ 485, 1123]], dtype=int64)
              precision    recall  f1-score   support

           0       0.90      0.72      0.80      6312
           1       0.38      0.70      0.50      1608

    accuracy                           0.71      7920
   macro avg       0.64      0.71      0.65      7920
weighted avg       0.80      0.71      0.74      7920
```

In [120…

```
## Validation metrics
roc_auc_score(y_val, lr.predict(X_val))
recall_score(y_val, lr.predict(X_val))
confusion_matrix(y_val, lr.predict(X_val))
print(classification_report(y_val, lr.predict(X_val)))
```

Out[120…

```
0.7011966306712709
```

Out[120…

```
0.7016806722689075
```

Out[120…

```
array([[590, 252],
       [ 71, 167]], dtype=int64)
              precision    recall  f1-score   support

           0       0.89      0.70      0.79       842
           1       0.40      0.70      0.51       238

    accuracy                           0.70      1080
   macro avg       0.65      0.70      0.65      1080
weighted avg       0.78      0.70      0.72      1080
```

In [ ]:

In [ ]:

More linear models - SVM

In [121…

```
from sklearn.svm import SVC

## Importing relevant metrics
from sklearn.metrics import roc_auc_score, f1_score, recall_score, confusion_matrix, class
```

In [122…

```
## Using categorical features and scaled numerical features
X_train = np.concatenate((df_train[selected_cat_vars].values, sc_X_train[selected_cont_var
X_val = np.concatenate((df_val[selected_cat_vars].values, sc_X_val[selected_cont_vars].val
X_test = np.concatenate((df_test[selected_cat_vars].values, sc_X_test[selected_cont_vars].

X_train.shape, X_val.shape, X_test.shape
```

Out[122…

```
((7920, 10), (1080, 10), (1000, 10))
```

In [123…

```
weights_dict = {0: 1.0, 1: 3.92}
weights_dict
```

Out[123…

```
{0: 1.0, 1: 3.92}
```

In [124…

```
svm = SVC(C = 1.0, kernel = "linear", class_weight = weights_dict)
```

In [125…

```
svm.fit(X_train, y_train)
```

Out[125…

```
SVC(class_weight={0: 1.0, 1: 3.92}, kernel='linear')
```

In [126…

```
## Fitted model parameters
selected_cat_vars + selected_cont_vars

svm.coef_
svm.intercept_
```

Out[126…

```
['Gender',
 'HasCrCard',
 'IsActiveMember',
 'country_France',
 'country_Germany',
 'country_Spain',
 'Age',
 'NumOfProducts',
 'Surname_enc',
 'tenure_age_ratio']
```

Out[126…

```
array([[-0.47120317, -0.05289943, -0.73126806, -0.30819839,  0.55381363,
        -0.24561524,  0.87497379, -0.04729496, -0.05552899, -0.03858295]])
```

Out[126…

```
array([0.45527487])
```

In [127…

```
## Training metrics
roc_auc_score(y_train, svm.predict(X_train))
recall_score(y_train, svm.predict(X_train))
confusion_matrix(y_train, svm.predict(X_train))
print(classification_report(y_train, svm.predict(X_train)))
```

Out[127…

```
0.7125033104439777
```

Out[127…

```
0.6946517412935324
```

Out[127…

```
array([[4610, 1702],
       [ 491, 1117]], dtype=int64)
              precision    recall  f1-score   support

           0       0.90      0.73      0.81      6312
           1       0.40      0.69      0.50      1608
```

```
    accuracy                           0.72      7920
   macro avg       0.65      0.71      0.66      7920
weighted avg       0.80      0.72      0.75      7920
```

In [128…

```
## Validation metrics
roc_auc_score(y_val, svm.predict(X_val))
recall_score(y_val, svm.predict(X_val))
confusion_matrix(y_val, svm.predict(X_val))
print(classification_report(y_val, svm.predict(X_val)))
```

Out[128…

```
0.6984570550310385
```

Out[128…

```
0.6890756302521008
```

Out[128…

```
array([[596, 246],
       [ 74, 164]], dtype=int64)
              precision    recall  f1-score   support

           0       0.89      0.71      0.79       842
           1       0.40      0.69      0.51       238

    accuracy                           0.70      1080
   macro avg       0.64      0.70      0.65      1080
weighted avg       0.78      0.70      0.73      1080
```

In [ ]:

In [ ]:

Plot decision boundaries of linear models

To plot decision boundaries of classification models in a 2-D space, we first need to train our models on a 2-D space. The best option is to use our existing data (with > 2 features) and apply dimensionality reduction techniques (like PCA) on it and then train our models on this data with a reduced number of features

In [129…

```
from sklearn.decomposition import PCA
```

In [130…

```
pca = PCA(n_components=2)
```

In [131…

```
## Transforming the dataset using PCA
X = pca.fit_transform(X_train)
y = y_train
X_train.shape
X.shape
y.shape
```

Out[131…

```
(7920, 10)
```

Out[131…

```
(7920, 2)
```

Out[131…

```
(7920,)
```

In [132…

```
## Checking the variance explained by the reduced features
pca.explained_variance_ratio_
```

Out[132… array([0.2602733 , 0.18789887])

In [133…
```
# Creating a mesh region where the boundary will be plotted
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
                     np.arange(y_min, y_max, 0.1))
```

In [134…
```
## Fitting LR model on 2 features
lr.fit(X, y)
```

Out[134… LogisticRegression(class_weight={0: 1.0, 1: 3.925373134328358}, n_jobs=-1)

In [135…
```
## Fitting SVM model on 2 features
svm.fit(X,y)
```

Out[135… SVC(class_weight={0: 1.0, 1: 3.92}, kernel='linear')

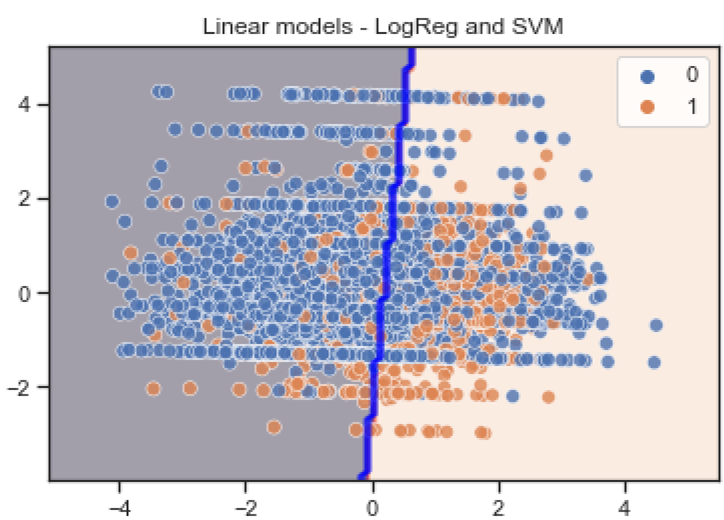
In [136…
```
## Plotting decision boundary for LR
z1 = lr.predict(np.c_[xx.ravel(), yy.ravel()])
z1 = z1.reshape(xx.shape)

## Plotting decision boundary for SVM
z2 = svm.predict(np.c_[xx.ravel(), yy.ravel()])
z2 = z2.reshape(xx.shape)

# Displaying the result
plt.contourf(xx, yy, z1, alpha=0.4) # LR
plt.contour(xx, yy, z2, alpha=0.4, colors = 'blue') # SVM
sns.scatterplot(X[:,0], X[:,1], hue = y_train, s = 50, alpha = 0.8)
plt.title('Linear models - LogReg and SVM')
```

Out[136… <matplotlib.contour.QuadContourSet at 0x14f94f46f70>

Out[136… <matplotlib.contour.QuadContourSet at 0x14f94f521c0>

Out[136… <AxesSubplot:>

Out[136… Text(0.5, 1.0, 'Linear models - LogReg and SVM')



In [ ]:

In [ ]:

More baseline models (Non-linear) : Decision Tree

In [137…

```
from sklearn.tree import DecisionTreeClassifier

## Importing relevant metrics
from sklearn.metrics import roc_auc_score, f1_score, recall_score, confusion_matrix, class
```

In [138…

```
weights_dict = {0: 1.0, 1: 3.92}
weights_dict
```

Out[138…

```
{0: 1.0, 1: 3.92}
```

In [139…

```
## Features selected from the RFE process
selected_feats_dt
```

Out[139…

```
['IsActiveMember',
 'country_Germany',
 'Age',
 'Balance',
 'NumOfProducts',
 'EstimatedSalary',
 'bal_per_product',
 'bal_by_est_salary',
 'tenure_age_ratio',
 'age_surname_mean_churn']
```

In [140…

```
## Re-defining X_train and X_val to consider original unscaled continuous features. y_trai
X_train = df_train[selected_feats_dt].values
X_val = df_val[selected_feats_dt].values
X_train.shape, y_train.shape
X_val.shape, y_val.shape
```

Out[140…

```
((7920, 10), (7920,))
```

Out[140…

```
((1080, 10), (1080,))
```

In [141…

```
clf = DecisionTreeClassifier(criterion = 'entropy', class_weight = weights_dict, max_depth
                             , min_samples_split = 25, min_samples_leaf = 15)
```

In [142…

```
clf.fit(X_train, y_train)
```

Out[142…

```
DecisionTreeClassifier(class_weight={0: 1.0, 1: 3.92}, criterion='entropy',
                       max_depth=4, min_samples_leaf=15, min_samples_split=25)
```

In [143…

```
## Checking the importance of different features of the model
pd.DataFrame({'features': selected_feats,
              'importance': clf.feature_importances_
             }).sort_values(by = 'importance', ascending=False)
```

Out[143…

	features	importance
2	IsActiveMember	0.476857
4	country_Germany	0.351836

	features	importance
0	Gender	0.096427
3	country_France	0.032250
1	HasCrCard	0.028357
7	NumOfProducts	0.011373
5	country_Spain	0.002900
6	Age	0.000000
8	Surname_enc	0.000000
9	tenure_age_ratio	0.000000

Evaluating the model - Metrics

In [144…

```
## Training metrics
roc_auc_score(y_train, clf.predict(X_train))
recall_score(y_train, clf.predict(X_train))
confusion_matrix(y_train, clf.predict(X_train))
print(classification_report(y_train, clf.predict(X_train)))
```

Out[144…

```
0.7514707829672929
```

Out[144…

```
0.7369402985074627
```

Out[144…

```
array([[4835, 1477],
       [ 423, 1185]], dtype=int64)
              precision    recall  f1-score   support

           0       0.92      0.77      0.84      6312
           1       0.45      0.74      0.56      1608

    accuracy                           0.76      7920
   macro avg       0.68      0.75      0.70      7920
weighted avg       0.82      0.76      0.78      7920
```

In [145…

```
## Validation metrics
roc_auc_score(y_val, clf.predict(X_val))
recall_score(y_val, clf.predict(X_val))
confusion_matrix(y_val, clf.predict(X_val))
print(classification_report(y_val, clf.predict(X_val)))
```

Out[145…

```
0.7477394758378411
```

Out[145…

```
0.7436974789915967
```

Out[145…

```
array([[633, 209],
       [ 61, 177]], dtype=int64)
              precision    recall  f1-score   support

           0       0.91      0.75      0.82       842
           1       0.46      0.74      0.57       238

    accuracy                           0.75      1080
   macro avg       0.69      0.75      0.70      1080
weighted avg       0.81      0.75      0.77      1080
```

In [ ]:

In [ ]:

Plot decision boundaries of non-linear model

In [146…

```python
from sklearn.decomposition import PCA
```

In [147…

```python
pca = PCA(n_components=2)
```

In [148…

```python
## Transforming the dataset using PCA
X = pca.fit_transform(X_train)
y = y_train
X_train.shape
X.shape
y.shape
```

Out[148…
```
(7920, 10)
```

Out[148…
```
(7920, 2)
```

Out[148…
```
(7920,)
```

In [149…

```python
## Checking the variance explained by the reduced features
pca.explained_variance_ratio_
```

Out[149…
```
array([0.65049371, 0.31643934])
```

In [150…

```python
# Creating a mesh region where the boundary will be plotted
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, 100),
                     np.arange(y_min, y_max, 100))
```

In [151…

```python
## Fitting tree model on 2 features
clf.fit(X, y)
```

Out[151…
```
DecisionTreeClassifier(class_weight={0: 1.0, 1: 3.92}, criterion='entropy',
                       max_depth=4, min_samples_leaf=15, min_samples_split=25)
```

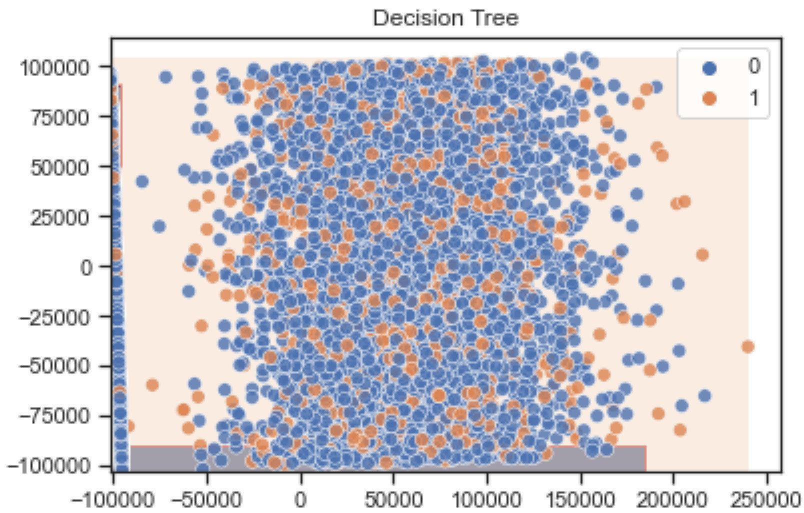
In [152…

```python
## Plotting decision boundary for Decision Tree (DT)
z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
z = z.reshape(xx.shape)

# Displaying the result
plt.contourf(xx, yy, z, alpha=0.4) # DT
sns.scatterplot(X[:,0], X[:,1], hue = y_train, s = 50, alpha = 0.8)
plt.title('Decision Tree')
```

Out[152…
```
<matplotlib.contour.QuadContourSet at 0x14fa32722b0>
```

Out[152…
```
<AxesSubplot:>
```

Out[152…
```
Text(0.5, 1.0, 'Decision Tree')
```



Decision tree rule engine visualization

In [153…
```
from sklearn.tree import export_graphviz
import subprocess
```

In [154…
```
clf = DecisionTreeClassifier(criterion = 'entropy', class_weight = weights_dict, max_deptl
                            , min_samples_split = 25, min_samples_leaf = 15)

clf.fit(X_train, y_train)
```

Out[154…
```
DecisionTreeClassifier(class_weight={0: 1.0, 1: 3.92}, criterion='entropy',
                       max_depth=3, min_samples_leaf=15, min_samples_split=25)
```

In [155…
```
## Export as dot file
dot_data = export_graphviz(clf, out_file = 'tree.dot'
                          , feature_names = selected_feats_dt
                          , class_names = ['Did not churn', 'Churned']
                          , rounded = True, proportion = False
                          , precision = 2, filled = True)
```

In [156…
```
## Convert to png using system command (requires Graphviz)
#subprocess.run(['dot', '-Tpng','tree.dot', '-o', 'tree.png', '-Gdpi=600'])
```

In [157…
```
## Display the rule-set of a single tree
#from IPython.display import Image
#Image(filename = 'tree.png')
```

In [ ]:
```
```

In [ ]:
```
```

Spot-checking various ML algorithms

Steps :

- Automate data preparation and model run through Pipelines

- Model Zoo : List of all models to compare/spot-check
- Evaluate using k-fold Cross validation framework

Note : Restart the kernel and read the original dataset again followed by train-test split and then come directly to this section of the notebook

Automating data preparation and model run through Pipelines

In [158…

```
from sklearn.base import BaseEstimator, TransformerMixin
```

In [159…

```
class CategoricalEncoder(BaseEstimator, TransformerMixin):
    """
    Encodes categorical columns using LabelEncoding, OneHotEncoding and TargetEncoding.
    LabelEncoding is used for binary categorical columns
    OneHotEncoding is used for columns with <= 10 distinct values
    TargetEncoding is used for columns with higher cardinality (>10 distinct values)

    """

    def __init__(self, cols = None, lcols = None, ohecols = None, tcols = None, reduce_df
        """

        Parameters
        ----------
        cols : list of str
            Columns to encode.  Default is to one-hot/target/label encode all categorical
        reduce_df : bool
            Whether to use reduced degrees of freedom for encoding
            (that is, add N-1 one-hot columns for a column with N
            categories). E.g. for a column with categories A, B,
            and C: When reduce_df is True, A=[1, 0], B=[0, 1],
            and C=[0, 0].  When reduce_df is False, A=[1, 0, 0],
            B=[0, 1, 0], and C=[0, 0, 1]
            Default = False

        """

        if isinstance(cols,str):
            self.cols = [cols]
        else :
            self.cols = cols

        if isinstance(lcols,str):
            self.lcols = [lcols]
        else :
            self.lcols = lcols

        if isinstance(ohecols,str):
            self.ohecols = [ohecols]
        else :
            self.ohecols = ohecols

        if isinstance(tcols,str):
            self.tcols = [tcols]
        else :
            self.tcols = tcols

        self.reduce_df = reduce_df


    def fit(self, X, y):
```

```
"""Fit label/one-hot/target encoder to X and y

Parameters
----------
X : pandas DataFrame, shape [n_samples, n_columns]
    DataFrame containing columns to encode
y : pandas Series, shape = [n_samples]
    Target values.

Returns
-------
self : encoder
    Returns self.
"""

# Encode all categorical cols by default
if self.cols is None:
    self.cols = [c for c in X if str(X[c].dtype)=='object']

# Check columns are in X
for col in self.cols:
    if col not in X:
        raise ValueError('Column \''+col+'\' not in X')

# Separating out lcols, ohecols and tcols
if self.lcols is None:
    self.lcols = [c for c in self.cols if X[c].nunique() <= 2]

if self.ohecols is None:
    self.ohecols = [c for c in self.cols if ((X[c].nunique() > 2) & (X[c].nunique

if self.tcols is None:
    self.tcols = [c for c in self.cols if X[c].nunique() > 10]


## Create Label Encoding mapping
self.lmaps = dict()
for col in self.lcols:
    self.lmaps[col] = dict(zip(X[col].values, X[col].astype('category').cat.codes.


## Create OneHot Encoding mapping
self.ohemaps = dict() #dict to store map for each column
for col in self.ohecols:
    self.ohemaps[col] = []
    uniques = X[col].unique()
    for unique in uniques:
        self.ohemaps[col].append(unique)
    if self.reduce_df:
        del self.ohemaps[col][-1]


## Create Target Encoding mapping
self.global_target_mean = y.mean().round(2)
self.sum_count = dict()
for col in self.tcols:
    self.sum_count[col] = dict()
    uniques = X[col].unique()
    for unique in uniques:
        ix = X[col]==unique
        self.sum_count[col][unique] = (y[ix].sum(),ix.sum())


## Return the fit object
return self
```

```python
    def transform(self, X, y=None):
        """Perform label/one-hot/target encoding transformation.

        Parameters
        ----------
        X : pandas DataFrame, shape [n_samples, n_columns]
            DataFrame containing columns to label encode

        Returns
        -------
        pandas DataFrame
            Input DataFrame with transformed columns
        """

        Xo = X.copy()
        ## Perform label encoding transformation
        for col, lmap in self.lmaps.items():

            # Map the column
            Xo[col] = Xo[col].map(lmap)
            Xo[col].fillna(-1, inplace=True) ## Filling new values with -1


        ## Perform one-hot encoding transformation
        for col, vals in self.ohemaps.items():
            for val in vals:
                new_col = col+'_'+str(val)
                Xo[new_col] = (Xo[col]==val).astype('uint8')
            del Xo[col]


        ## Perform LOO target encoding transformation
        # Use normal target encoding if this is test data
        if y is None:
            for col in self.sum_count:
                vals = np.full(X.shape[0], np.nan)
                for cat, sum_count in self.sum_count[col].items():
                    vals[X[col]==cat] = (sum_count[0]/sum_count[1]).round(2)
                Xo[col] = vals
                Xo[col].fillna(self.global_target_mean, inplace=True) # Filling new values

        # LOO target encode each column
        else:
            for col in self.sum_count:
                vals = np.full(X.shape[0], np.nan)
                for cat, sum_count in self.sum_count[col].items():
                    ix = X[col]==cat
                    if sum_count[1] > 1:
                        vals[ix] = ((sum_count[0]-y[ix].reshape(-1,))/(sum_count[1]-1)).ro
                    else :
                        vals[ix] = ((y.sum() - y[ix])/(X.shape[0] - 1)).round(2) # Caterin
                                                                                  # catego

                Xo[col] = vals
                Xo[col].fillna(self.global_target_mean, inplace=True) # Filling new values


        ## Return encoded DataFrame
        return Xo


    def fit_transform(self, X, y=None):
        """Fit and transform the data via label/one-hot/target encoding.

        Parameters
```

```
        ----------
        X : pandas DataFrame, shape [n_samples, n_columns]
            DataFrame containing columns to encode
        y : pandas Series, shape = [n_samples]
            Target values (required!).

        Returns
        -------
        pandas DataFrame
            Input DataFrame with transformed columns
        """

        return self.fit(X, y).transform(X, y)
```

In [160…

```
class AddFeatures(BaseEstimator):
    """
    Add new, engineered features using original categorical and numerical features of the
    """

    def __init__(self, eps = 1e-6):
        """
        Parameters
        ----------
        eps : A small value to avoid divide by zero error. Default value is 0.000001
        """

        self.eps = eps


    def fit(self, X, y=None):
        return self


    def transform(self, X):
        """
        Parameters
        ----------
        X : pandas DataFrame, shape [n_samples, n_columns]
            DataFrame containing base columns using which new interaction-based features c
        """
        Xo = X.copy()
        ## Add 4 new columns - bal_per_product, bal_by_est_salary, tenure_age_ratio, age_s
        Xo['bal_per_product'] = Xo.Balance/(Xo.NumOfProducts + self.eps)
        Xo['bal_by_est_salary'] = Xo.Balance/(Xo.EstimatedSalary + self.eps)
        Xo['tenure_age_ratio'] = Xo.Tenure/(Xo.Age + self.eps)
        Xo['age_surname_enc'] = np.sqrt(Xo.Age) * Xo.Surname_enc

        ## Returning the updated dataframe
        return Xo


    def fit_transform(self, X, y=None):
        """
        Parameters
        ----------
        X : pandas DataFrame, shape [n_samples, n_columns]
            DataFrame containing base columns using which new interaction-based features c
        """
        return self.fit(X,y).transform(X)
```

In [161…

```
class CustomScaler(BaseEstimator, TransformerMixin):
    """
    A custom standard scaler class with the ability to apply scaling on selected columns
    """

    def __init__(self, scale_cols = None):
        """
        Parameters
        ----------
        scale_cols : list of str
            Columns on which to perform scaling and normalization. Default is to scale al

        """
        self.scale_cols = scale_cols


    def fit(self, X, y=None):
        """
        Parameters
        ----------
        X : pandas DataFrame, shape [n_samples, n_columns]
            DataFrame containing columns to scale
        """

        # Scaling all non-categorical columns if user doesn't provide the list of columns
        if self.scale_cols is None:
            self.scale_cols = [c for c in X if ((str(X[c].dtype).find('float') != -1) or


        ## Create mapping corresponding to scaling and normalization
        self.maps = dict()
        for col in self.scale_cols:
            self.maps[col] = dict()
            self.maps[col]['mean'] = np.mean(X[col].values).round(2)
            self.maps[col]['std_dev'] = np.std(X[col].values).round(2)

        # Return fit object
        return self


    def transform(self, X):
        """
        Parameters
        ----------
        X : pandas DataFrame, shape [n_samples, n_columns]
            DataFrame containing columns to scale
        """
        Xo = X.copy()

        ## Map transformation to respective columns
        for col in self.scale_cols:
            Xo[col] = (Xo[col] - self.maps[col]['mean']) / self.maps[col]['std_dev']


        # Return scaled and normalized DataFrame
        return Xo


    def fit_transform(self, X, y=None):
        """
        Parameters
        ----------
        X : pandas DataFrame, shape [n_samples, n_columns]
            DataFrame containing columns to scale
        """
        # Fit and return transformed dataframe
```

```
        return self.fit(X).transform(X)
```

In [ ]:

In [ ]:

Pipeline in action for a single model

In [162…

```
from sklearn.pipeline import Pipeline
from sklearn.tree import DecisionTreeClassifier

## Importing relevant metrics
from sklearn.metrics import roc_auc_score, f1_score, recall_score, confusion_matrix, class
```

In [163…

```
X = df_train.drop(columns = ['Exited'], axis = 1)
X_val = df_val.drop(columns = ['Exited'], axis = 1)

cols_to_scale = ['CreditScore', 'Age', 'Balance', 'EstimatedSalary', 'bal_per_product', 'b
                ,'age_surname_enc']
```

In [164…

```
weights_dict = {0 : 1.0, 1 : 3.92}

clf = DecisionTreeClassifier(criterion = 'entropy', class_weight = weights_dict, max_depth
                            , min_samples_split = 25, min_samples_leaf = 15)
```

In [165…

```
model = Pipeline(steps = [('categorical_encoding', CategoricalEncoder()),
                          ('add_new_features', AddFeatures()),
                          ('standard_scaling', CustomScaler(cols_to_scale)),
                          ('classifier', clf)
                         ])
```

In [166…

```
# Fit pipeline with training data
model.fit(X,y_train)
```

Out[166…

```
Pipeline(steps=[('categorical_encoding',
                 CategoricalEncoder(cols=[], lcols=[], ohecols=[], tcols=[])),
                ('add_new_features', AddFeatures()),
                ('standard_scaling',
                 CustomScaler(scale_cols=['CreditScore', 'Age', 'Balance',
                                          'EstimatedSalary', 'bal_per_product',
                                          'bal_by_est_salary',
                                          'tenure_age_ratio',
                                          'age_surname_enc'])),
                ('classifier',
                 DecisionTreeClassifier(class_weight={0: 1.0, 1: 3.92},
                                        criterion='entropy', max_depth=4,
                                        min_samples_leaf=15,
                                        min_samples_split=25))])
```

In [167…

```
# Predict target values on val data
val_preds = model.predict(X_val)
```

In [168…

```
## Validation metrics
roc_auc_score(y_val, val_preds)
recall_score(y_val, val_preds)
confusion_matrix(y_val, val_preds)
print(classification_report(y_val, val_preds))
```

Out[168…

```
0.7477394758378411
```

Out[168…

```
0.7436974789915967
```

Out[168…

```
array([[633, 209],
       [ 61, 177]], dtype=int64)
              precision    recall  f1-score   support

           0       0.91      0.75      0.82       842
           1       0.46      0.74      0.57       238

    accuracy                           0.75      1080
   macro avg       0.69      0.75      0.70      1080
weighted avg       0.81      0.75      0.77      1080
```

In [ ]:

```

```

Model Zoo + k-fold Cross Validation

Models : RF, LGBM, XGB, Naive Bayes (Gaussian/Multinomial), kNN

How are models selected ?

- Why only tree models ? Why not SVM or ANNs?

In [169…

```
from sklearn.model_selection import cross_val_score
```

In [170…

```
## Preparing data and a few common model parameters
X = df_train.drop(columns = ['Exited'], axis = 1)
y = y_train.ravel()

weights_dict = {0 : 1.0, 1 : 3.93}
_, num_samples = np.unique(y_train, return_counts = True)
weight = (num_samples[0]/num_samples[1]).round(2)
weight

cols_to_scale = ['CreditScore', 'Age', 'Balance', 'EstimatedSalary', 'bal_per_product', 'b
                 ,'age_surname_enc']
```

Out[170…

```
3.93
```

In [171…

```
## Importing the models to be tried out
from sklearn.ensemble import RandomForestClassifier, ExtraTreesClassifier
from lightgbm import LGBMClassifier
from xgboost import XGBClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB, MultinomialNB, ComplementNB, BernoulliNB
```

Read more about XGB parameters from here : https://xgboost.readthedocs.io/en/latest/parameter.html

Tips to tune parameters for LightGBM : https://lightgbm.readthedocs.io/en/latest/Parameters-Tuning.html

```python
In [172…
## Preparing a list of models to try out in the spot-checking process
def model_zoo(models = dict()):
    # Tree models
    for n_trees in [21, 1001]:
        models['rf_' + str(n_trees)] = RandomForestClassifier(n_estimators = n_trees, n_jo
                                                             , class_weight = weights_dic
                                                             , min_samples_split = 30, mi

        models['lgb_' + str(n_trees)] = LGBMClassifier(boosting_type='dart', num_leaves=31
                                                       , n_estimators=n_trees, class_weigh
                                                       , colsample_bytree=0.6, reg_alpha=(
                                                       , importance_type = 'gain')

        models['xgb_' + str(n_trees)] = XGBClassifier(objective='binary:logistic', n_estir
                                                      , learning_rate = 0.03, n_jobs = -1,
                                                      , reg_alpha = 0.3, reg_lambda = 0.1,

        models['et_' + str(n_trees)] = ExtraTreesClassifier(n_estimators=n_trees, criterio
                                                            , max_features = 0.6, n_jobs =
                                                            , min_samples_split = 30, min_

    # kNN models
    for n in [3,5,11]:
        models['knn_' + str(n)] = KNeighborsClassifier(n_neighbors=n)

    # Naive-Bayes models
    models['gauss_nb'] = GaussianNB()
    models['multi_nb'] = MultinomialNB()
    models['compl_nb'] = ComplementNB()
    models['bern_nb'] = BernoulliNB()

    return models
```

```python
In [173…
## Automation of data preparation and model run through pipelines
def make_pipeline(model):
    '''
    Creates pipeline for the model passed as the argument. Uses standard scaling only in c
    Ignores scaling step for tree/Naive Bayes models
    '''

    if (str(model).find('KNeighborsClassifier') != -1):
        pipe =  Pipeline(steps = [('categorical_encoding', CategoricalEncoder()),
                                  ('add_new_features', AddFeatures()),
                                  ('standard_scaling', CustomScaler(cols_to_scale)),
                                  ('classifier', model)
                                 ])
    else :
        pipe =  Pipeline(steps = [('categorical_encoding', CategoricalEncoder()),
                                  ('add_new_features', AddFeatures()),
                                  ('classifier', model)
                                 ])


    return pipe
```

```python
In [174…
## Run/Evaluate all 15 models using KFold cross-validation (5 folds)
def evaluate_models(X, y, models, folds = 5, metric = 'recall'):
    results = dict()
    for name, model in models.items():
        # Evaluate model through automated pipelines
        pipeline = make_pipeline(model)
        scores = cross_val_score(pipeline, X, y, cv = folds, scoring = metric, n_jobs = -1
```

```
        # Store results of the evaluated model
        results[name] = scores
        mu, sigma = np.mean(scores), np.std(scores)
        # Printing individual model results
        print('Model {}: mean = {}, std_dev = {}'.format(name, mu, sigma))

    return results
```

In [175…

```
## Spot-checking in action
models = model_zoo()
print('Recall metric')
results = evaluate_models(X, y , models, metric = 'recall')
print('F1-score metric')
results = evaluate_models(X, y , models, metric = 'f1')
```

```
Recall metric
Model rf_21: mean = 0.7518391671987772, std_dev = 0.029205327813349013
Model lgb_21: mean = 0.7866856291480427, std_dev = 0.015745566437193475
Model xgb_21: mean = 0.7692594957527912, std_dev = 0.02427969281921412
Model et_21: mean = 0.7512470733925427, std_dev = 0.01103147015267978
Model rf_1001: mean = 0.7518449720400147, std_dev = 0.02564101340498873
Model lgb_1001: mean = 0.6884232116251622, std_dev = 0.014573973874519829
Model xgb_1001: mean = 0.6772334126661635, std_dev = 0.011687132255824195
Model et_1001: mean = 0.733830614732687, std_dev = 0.00696950210534982
Model knn_3: mean = 0.32214933921557243, std_dev = 0.021051639994704833
Model knn_5: mean = 0.2879356049612043, std_dev = 0.00639668044045995 3
Model knn_11: mean = 0.23568622898163735, std_dev = 0.023099705052575383
Model gauss_nb: mean = 0.0360906329211896, std_dev = 0.0151162576177723
Model multi_nb: mean = 0.5404191095373541, std_dev = 0.022285871235774777
Model compl_nb: mean = 0.5404191095373541, std_dev = 0.022285871235774777
Model bern_nb: mean = 0.31030552814380524, std_dev = 0.022201596952259223
F1-score metric
Model rf_21: mean = 0.6253662711845134, std_dev = 0.01989037113986081
Model lgb_21: mean = 0.6445713376921776, std_dev = 0.010347896896123705
Model xgb_21: mean = 0.6441267256402282, std_dev = 0.013483753302888951
Model et_21: mean = 0.5886032098517683, std_dev = 0.010825466341616312
Model rf_1001: mean = 0.6281573127588186, std_dev = 0.017376276917760322
Model lgb_1001: mean = 0.677231392541388, std_dev = 0.009841732603586511
Model xgb_1001: mean = 0.6888397756457623, std_dev = 0.010820878580110008
Model et_1001: mean = 0.5901450745948775, std_dev = 0.0065883881399132
Model knn_3: mean = 0.4067382505578322, std_dev = 0.022720962890263006
Model knn_5: mean = 0.3899028888667188, std_dev = 0.007862325744140088
Model knn_11: mean = 0.3512153712304775, std_dev = 0.027579669538701175
Model gauss_nb: mean = 0.06337492524758484, std_dev = 0.024499096874076205
Model multi_nb: mean = 0.329272413622277, std_dev = 0.011346796699221388
Model compl_nb: mean = 0.329272413622277, std_dev = 0.011346796699221388
Model bern_nb: mean = 0.34121749133649887, std_dev = 0.016767819528172967
```

Based on the relevant metric, a suitable model can be chosen for further hyperparameter tuning.

LightGBM is chosen for further hyperparameter tuning because it has the best performance on recall metric and it came close second when comparing using F1-scores

In [ ]:

In [ ]:

Hyperparameter tuning

RandomSearchCV vs GridSearchCV

- Random Search is more suitable for large datasets, with a large number of parameter settings
- Grid Search results in a more precise hyperparameter tuning, thus resulting in better model performance. Intelligent tuning mechanism can also help reduce the time taken in GridSearch by a large factor

- Will optimize on F1 metric. We could easily reach 75% Recall from the default parameters as seen earlier

In [176…
```
from sklearn.pipeline import Pipeline
from sklearn.model_selection import GridSearchCV, RandomizedSearchCV
from lightgbm import LGBMClassifier
```

In [177…
```
## Preparing data and a few common model parameters
# Unscaled features will be used since it's a tree model

X_train = df_train.drop(columns = ['Exited'], axis = 1)
X_val = df_val.drop(columns = ['Exited'], axis = 1)

X_train.shape, y_train.shape
X_val.shape, y_val.shape
```

Out[177…
```
((7920, 17), (7920,))
```

Out[177…
```
((1080, 17), (1080,))
```

In [178…
```
lgb = LGBMClassifier(boosting_type = 'dart', min_child_samples = 20, n_jobs = - 1, importa
```

In [179…
```
model = Pipeline(steps = [('categorical_encoding', CategoricalEncoder()),
                          ('add_new_features', AddFeatures()),
                          ('classifier', lgb)
                         ])
```

Randomized Search

In [180…
```
## Exhaustive list of parameters
parameters = {'classifier__n_estimators':[10, 21, 51, 100, 201, 350, 501]
             ,'classifier__max_depth': [3, 4, 6, 9]
             ,'classifier__num_leaves':[7, 15, 31]
             ,'classifier__learning_rate': [0.03, 0.05, 0.1, 0.5, 1]
             ,'classifier__colsample_bytree': [0.3, 0.6, 0.8]
             ,'classifier__reg_alpha': [0, 0.3, 1, 5]
             ,'classifier__reg_lambda': [0.1, 0.5, 1, 5, 10]
             ,'classifier__class_weight': [{0:1,1:1.0}, {0:1,1:1.96}, {0:1,1:3.0}, {0:1,1:
             }
```

In [181…
```
search = RandomizedSearchCV(model, parameters, n_iter = 20, cv = 5, scoring = 'f1')
```

In [182…
```
search.fit(X_train, y_train.ravel())
```

Out[182…
```
RandomizedSearchCV(cv=5,
                   estimator=Pipeline(steps=[('categorical_encoding',
                                              CategoricalEncoder()),
                                             ('add_new_features',
                                              AddFeatures()),
                                             ('classifier',
                                              LGBMClassifier(boosting_type='dart',
                                                             importance_type='gain'))]),
```

```
                        n_iter=20,
                        param_distributions={'classifier__class_weight': [{0: 1,
                                                                            1: 1.0},
                                                                           {0: 1,
                                                                            1: 1.96},
                                                                           {0: 1,
                                                                            1: 3.0},
                                                                           {0: 1,
                                                                            1: 3.93}],
                                             'classifier__colsample_bytree': [0.3,
                                                                               0.6,
                                                                               0.8],
                                             'classifier__learning_rate': [0.03,
                                                                            0.05, 0.1,
                                                                            0.5, 1],
                                             'classifier__max_depth': [3, 4, 6, 9],
                                             'classifier__n_estimators': [10, 21, 51,
                                                                           100, 201,
                                                                           350, 501],
                                             'classifier__num_leaves': [7, 15, 31],
                                             'classifier__reg_alpha': [0, 0.3, 1, 5],
                                             'classifier__reg_lambda': [0.1, 0.5, 1,
                                                                         5, 10]},
                        scoring='f1')
```

In [183…

```
search.best_params_
search.best_score_
```

Out[183…

```
{'classifier__reg_lambda': 0.1,
 'classifier__reg_alpha': 0.3,
 'classifier__num_leaves': 15,
 'classifier__n_estimators': 501,
 'classifier__max_depth': 3,
 'classifier__learning_rate': 0.05,
 'classifier__colsample_bytree': 0.3,
 'classifier__class_weight': {0: 1, 1: 1.96}}
```

Out[183…

```
0.6855727578346781
```

In [184…

```
search.cv_results_
```

Out[184…

```
{'mean_fit_time': array([0.02788539, 0.1097023 , 0.20534282, 0.09811134, 0.31898508,
        0.08786659, 0.10689993, 0.01852937, 0.19821901, 0.31216998,
        0.51698999, 0.04714422, 0.75696902, 0.04974709, 0.018432  ,
        0.45006599, 0.21165037, 0.02485542, 0.47234678, 0.22145462]),
 'std_fit_time': array([0.00537894, 0.00567755, 0.01777777, 0.00578158, 0.00283643,
        0.01243339, 0.00638118, 0.00395629, 0.01614255, 0.02135362,
        0.05601457, 0.00852145, 0.03326349, 0.00700945, 0.00427063,
        0.02777142, 0.00712501, 0.00755138, 0.01266172, 0.02687798]),
 'mean_score_time': array([0.00106015, 0.00300145, 0.00320153, 0.00362582, 0.0081717 ,
        0.00312529, 0.00316548, 0.00983548, 0.00625052, 0.00316777,
        0.00402641, 0.00634933, 0.00937557, 0.00372586, 0.01297755,
        0.01439257, 0.00549116, 0.00631437, 0.00825205, 0.00763245]),
 'std_score_time': array([0.0021203 , 0.00368959, 0.00640306, 0.00607788, 0.00507851,
        0.00625057, 0.00633097, 0.00610359, 0.0076553 , 0.00633554,
        0.00605744, 0.00777786, 0.00765512, 0.00606258, 0.00431581,
        0.00784415, 0.00452954, 0.00773352, 0.00704386, 0.00710255]),
 'param_classifier__reg_lambda': masked_array(data=[0.5, 10, 1, 1, 0.5, 10, 0.5, 1, 1, 0.
1, 0.1, 1, 10,
                    0.1, 0.5, 1, 5, 0.5, 0.1, 0.5],
              mask=[False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False],
        fill_value='?',
```

```
              dtype=object),
 'param_classifier__reg_alpha': masked_array(data=[0.3, 0, 0, 0.3, 0.3, 0.3, 1, 5, 1, 0.3,
0.3, 5, 1, 1,
                   5, 1, 5, 0, 0.3, 1],
             mask=[False, False, False, False, False, False, False, False,
                   False, False, False, False, False, False, False, False,
                   False, False, False, False],
        fill_value='?',
              dtype=object),
 'param_classifier__num_leaves': masked_array(data=[31, 7, 31, 7, 7, 31, 7, 31, 31, 7, 31,
15, 31, 7, 7, 7,
                   15, 7, 15, 31],
             mask=[False, False, False, False, False, False, False, False,
                   False, False, False, False, False, False, False, False,
                   False, False, False, False],
        fill_value='?',
              dtype=object),
 'param_classifier__n_estimators': masked_array(data=[21, 201, 201, 201, 501, 100, 201, 5
1, 201, 501, 350,
                   100, 501, 100, 51, 501, 201, 21, 501, 201],
             mask=[False, False, False, False, False, False, False, False,
                   False, False, False, False, False, False, False, False,
                   False, False, False, False],
        fill_value='?',
              dtype=object),
 'param_classifier__max_depth': masked_array(data=[6, 9, 6, 3, 3, 6, 6, 3, 6, 4, 6, 4, 9,
4, 6, 4, 6, 3,
                   3, 4],
             mask=[False, False, False, False, False, False, False, False,
                   False, False, False, False, False, False, False, False,
                   False, False, False, False],
        fill_value='?',
              dtype=object),
 'param_classifier__learning_rate': masked_array(data=[0.05, 0.05, 0.03, 1, 1, 1, 0.03, 0.
5, 0.5, 1, 0.1,
                   0.03, 0.03, 0.05, 1, 0.05, 0.5, 0.1, 0.05, 0.1],
             mask=[False, False, False, False, False, False, False, False,
                   False, False, False, False, False, False, False, False,
                   False, False, False, False],
        fill_value='?',
              dtype=object),
 'param_classifier__colsample_bytree': masked_array(data=[0.6, 0.8, 0.3, 0.8, 0.8, 0.3, 0.
6, 0.6, 0.3, 0.3, 0.8,
                   0.3, 0.3, 0.6, 0.8, 0.6, 0.8, 0.3, 0.3, 0.6],
             mask=[False, False, False, False, False, False, False, False,
                   False, False, False, False, False, False, False, False,
                   False, False, False, False],
        fill_value='?',
              dtype=object),
 'param_classifier__class_weight': masked_array(data=[{0: 1, 1: 3.0}, {0: 1, 1: 1.0}, {0:
1, 1: 1.0},
                   {0: 1, 1: 1.96}, {0: 1, 1: 1.96}, {0: 1, 1: 3.93},
                   {0: 1, 1: 3.93}, {0: 1, 1: 1.0}, {0: 1, 1: 3.93},
                   {0: 1, 1: 1.96}, {0: 1, 1: 3.93}, {0: 1, 1: 1.96},
                   {0: 1, 1: 1.96}, {0: 1, 1: 3.93}, {0: 1, 1: 3.0},
                   {0: 1, 1: 1.0}, {0: 1, 1: 3.93}, {0: 1, 1: 1.96},
                   {0: 1, 1: 1.96}, {0: 1, 1: 3.0}],
             mask=[False, False, False, False, False, False, False, False,
                   False, False, False, False, False, False, False, False,
                   False, False, False, False],
        fill_value='?',
              dtype=object),
 'params': [{'classifier__reg_lambda': 0.5,
   'classifier__reg_alpha': 0.3,
   'classifier__num_leaves': 31,
   'classifier__n_estimators': 21,
```

```
 'classifier__max_depth': 6,
 'classifier__learning_rate': 0.05,
 'classifier__colsample_bytree': 0.6,
 'classifier__class_weight': {0: 1, 1: 3.0}},
{'classifier__reg_lambda': 10,
 'classifier__reg_alpha': 0,
 'classifier__num_leaves': 7,
 'classifier__n_estimators': 201,
 'classifier__max_depth': 9,
 'classifier__learning_rate': 0.05,
 'classifier__colsample_bytree': 0.8,
 'classifier__class_weight': {0: 1, 1: 1.0}},
{'classifier__reg_lambda': 1,
 'classifier__reg_alpha': 0,
 'classifier__num_leaves': 31,
 'classifier__n_estimators': 201,
 'classifier__max_depth': 6,
 'classifier__learning_rate': 0.03,
 'classifier__colsample_bytree': 0.3,
 'classifier__class_weight': {0: 1, 1: 1.0}},
{'classifier__reg_lambda': 1,
 'classifier__reg_alpha': 0.3,
 'classifier__num_leaves': 7,
 'classifier__n_estimators': 201,
 'classifier__max_depth': 3,
 'classifier__learning_rate': 1,
 'classifier__colsample_bytree': 0.8,
 'classifier__class_weight': {0: 1, 1: 1.96}},
{'classifier__reg_lambda': 0.5,
 'classifier__reg_alpha': 0.3,
 'classifier__num_leaves': 7,
 'classifier__n_estimators': 501,
 'classifier__max_depth': 3,
 'classifier__learning_rate': 1,
 'classifier__colsample_bytree': 0.8,
 'classifier__class_weight': {0: 1, 1: 1.96}},
{'classifier__reg_lambda': 10,
 'classifier__reg_alpha': 0.3,
 'classifier__num_leaves': 31,
 'classifier__n_estimators': 100,
 'classifier__max_depth': 6,
 'classifier__learning_rate': 1,
 'classifier__colsample_bytree': 0.3,
 'classifier__class_weight': {0: 1, 1: 3.93}},
{'classifier__reg_lambda': 0.5,
 'classifier__reg_alpha': 1,
 'classifier__num_leaves': 7,
 'classifier__n_estimators': 201,
 'classifier__max_depth': 6,
 'classifier__learning_rate': 0.03,
 'classifier__colsample_bytree': 0.6,
 'classifier__class_weight': {0: 1, 1: 3.93}},
{'classifier__reg_lambda': 1,
 'classifier__reg_alpha': 5,
 'classifier__num_leaves': 31,
 'classifier__n_estimators': 51,
 'classifier__max_depth': 3,
 'classifier__learning_rate': 0.5,
 'classifier__colsample_bytree': 0.6,
 'classifier__class_weight': {0: 1, 1: 1.0}},
{'classifier__reg_lambda': 1,
 'classifier__reg_alpha': 1,
 'classifier__num_leaves': 31,
 'classifier__n_estimators': 201,
 'classifier__max_depth': 6,
 'classifier__learning_rate': 0.5,
```

```
 'classifier__colsample_bytree': 0.3,
 'classifier__class_weight': {0: 1, 1: 3.93}},
{'classifier__reg_lambda': 0.1,
 'classifier__reg_alpha': 0.3,
 'classifier__num_leaves': 7,
 'classifier__n_estimators': 501,
 'classifier__max_depth': 4,
 'classifier__learning_rate': 1,
 'classifier__colsample_bytree': 0.3,
 'classifier__class_weight': {0: 1, 1: 1.96}},
{'classifier__reg_lambda': 0.1,
 'classifier__reg_alpha': 0.3,
 'classifier__num_leaves': 31,
 'classifier__n_estimators': 350,
 'classifier__max_depth': 6,
 'classifier__learning_rate': 0.1,
 'classifier__colsample_bytree': 0.8,
 'classifier__class_weight': {0: 1, 1: 3.93}},
{'classifier__reg_lambda': 1,
 'classifier__reg_alpha': 5,
 'classifier__num_leaves': 15,
 'classifier__n_estimators': 100,
 'classifier__max_depth': 4,
 'classifier__learning_rate': 0.03,
 'classifier__colsample_bytree': 0.3,
 'classifier__class_weight': {0: 1, 1: 1.96}},
{'classifier__reg_lambda': 10,
 'classifier__reg_alpha': 1,
 'classifier__num_leaves': 31,
 'classifier__n_estimators': 501,
 'classifier__max_depth': 9,
 'classifier__learning_rate': 0.03,
 'classifier__colsample_bytree': 0.3,
 'classifier__class_weight': {0: 1, 1: 1.96}},
{'classifier__reg_lambda': 0.1,
 'classifier__reg_alpha': 1,
 'classifier__num_leaves': 7,
 'classifier__n_estimators': 100,
 'classifier__max_depth': 4,
 'classifier__learning_rate': 0.05,
 'classifier__colsample_bytree': 0.6,
 'classifier__class_weight': {0: 1, 1: 3.93}},
{'classifier__reg_lambda': 0.5,
 'classifier__reg_alpha': 5,
 'classifier__num_leaves': 7,
 'classifier__n_estimators': 51,
 'classifier__max_depth': 6,
 'classifier__learning_rate': 1,
 'classifier__colsample_bytree': 0.8,
 'classifier__class_weight': {0: 1, 1: 3.0}},
{'classifier__reg_lambda': 1,
 'classifier__reg_alpha': 1,
 'classifier__num_leaves': 7,
 'classifier__n_estimators': 501,
 'classifier__max_depth': 4,
 'classifier__learning_rate': 0.05,
 'classifier__colsample_bytree': 0.6,
 'classifier__class_weight': {0: 1, 1: 1.0}},
{'classifier__reg_lambda': 5,
 'classifier__reg_alpha': 5,
 'classifier__num_leaves': 15,
 'classifier__n_estimators': 201,
 'classifier__max_depth': 6,
 'classifier__learning_rate': 0.5,
 'classifier__colsample_bytree': 0.8,
 'classifier__class_weight': {0: 1, 1: 3.93}},
```

```
 {'classifier__reg_lambda': 0.5,
  'classifier__reg_alpha': 0,
  'classifier__num_leaves': 7,
  'classifier__n_estimators': 21,
  'classifier__max_depth': 3,
  'classifier__learning_rate': 0.1,
  'classifier__colsample_bytree': 0.3,
  'classifier__class_weight': {0: 1, 1: 1.96}},
 {'classifier__reg_lambda': 0.1,
  'classifier__reg_alpha': 0.3,
  'classifier__num_leaves': 15,
  'classifier__n_estimators': 501,
  'classifier__max_depth': 3,
  'classifier__learning_rate': 0.05,
  'classifier__colsample_bytree': 0.3,
  'classifier__class_weight': {0: 1, 1: 1.96}},
 {'classifier__reg_lambda': 0.5,
  'classifier__reg_alpha': 1,
  'classifier__num_leaves': 31,
  'classifier__n_estimators': 201,
  'classifier__max_depth': 4,
  'classifier__learning_rate': 0.1,
  'classifier__colsample_bytree': 0.6,
  'classifier__class_weight': {0: 1, 1: 3.0}}],
'split0_test_score': array([0.66346154, 0.60080645, 0.51901566, 0.65189873, 0.65365854,
       0.65081724, 0.61256545, 0.64393939, 0.66871166, 0.64968153,
       0.67349927, 0.59557344, 0.69609508, 0.61315789, 0.6637931 ,
       0.65769231, 0.66762178, 0.52192982, 0.69230769, 0.66764275]),
'split1_test_score': array([0.6885759 , 0.63095238, 0.51818182, 0.67515924, 0.66123779,
       0.64264264, 0.6259542 , 0.67041199, 0.6863354 , 0.64238411,
       0.68103448, 0.60416667, 0.68676717, 0.63076923, 0.64534075,
       0.67041199, 0.66101695, 0.54065934, 0.69966997, 0.68      ]),
'split2_test_score': array([0.65740741, 0.6035503 , 0.51685393, 0.65732087, 0.67086614,
       0.64739884, 0.64240903, 0.64650284, 0.66466165, 0.65822785,
       0.68299712, 0.57322176, 0.68020305, 0.64713376, 0.67609618,
       0.65142857, 0.65454545, 0.49443207, 0.68965517, 0.68278805]),
'split3_test_score': array([0.68217054, 0.64367816, 0.48291572, 0.67807154, 0.67405063,
       0.66960352, 0.64720812, 0.68411215, 0.68161435, 0.67088608,
       0.6951567 , 0.5987526 , 0.69609508, 0.64766839, 0.66481994,
       0.67790262, 0.68428373, 0.52494577, 0.68932039, 0.6851312 ]),
'split4_test_score': array([0.65432099, 0.58365759, 0.49773756, 0.63414634, 0.62662338,
       0.64      , 0.63346105, 0.60754717, 0.65178571, 0.63829787,
       0.66951567, 0.56211813, 0.61976549, 0.62924282, 0.64978903,
       0.61333333, 0.6456044 , 0.5173913 , 0.65691057, 0.67048711]),
'mean_test_score': array([0.66918728, 0.61252898, 0.50694094, 0.65931934, 0.65728729,
       0.65009245, 0.63231957, 0.65050271, 0.67062176, 0.65189549,
       0.68044065, 0.58676652, 0.67578517, 0.63359442, 0.6599678 ,
       0.65415376, 0.66261446, 0.51987166, 0.68557276, 0.67720982]),
'std_test_score': array([0.0136897 , 0.02173147, 0.01436897, 0.01609526, 0.0169378 ,
       0.01044816, 0.01229238, 0.02619247, 0.0123428 , 0.01167051,
       0.00884294, 0.01621726, 0.0286472 , 0.0128527 , 0.01109828,
       0.02242585, 0.01305093, 0.0149361 , 0.01480758, 0.00690469]),
'rank_test_score': array([ 6, 17, 20,  9, 10, 14, 16, 13,  5, 12,  2, 18,  4, 15,  8, 11,
 7,
       19,  1,  3])}
```

In [ ]:

In [ ]:

Grid Search

In [185…

```python
## Current list of parameters
parameters = {'classifier__n_estimators':[201]
              ,'classifier__max_depth': [6]
              ,'classifier__num_leaves': [63]
              ,'classifier__learning_rate': [0.1]
              ,'classifier__colsample_bytree': [0.6, 0.8]
              ,'classifier__reg_alpha': [0, 1, 10]
              ,'classifier__reg_lambda': [0.1, 1, 5]
              ,'classifier__class_weight': [{0:1,1:3.0}]
              }
```

In [186…

```python
grid = GridSearchCV(model, parameters, cv = 5, scoring = 'f1', n_jobs = -1)
```

In [187…

```python
grid.fit(X_train, y_train.ravel())
```

Out[187…

```
GridSearchCV(cv=5,
             estimator=Pipeline(steps=[('categorical_encoding',
                                        CategoricalEncoder()),
                                       ('add_new_features', AddFeatures()),
                                       ('classifier',
                                        LGBMClassifier(boosting_type='dart',
                                                       importance_type='gain'))]),
             n_jobs=-1,
             param_grid={'classifier__class_weight': [{0: 1, 1: 3.0}],
                         'classifier__colsample_bytree': [0.6, 0.8],
                         'classifier__learning_rate': [0.1],
                         'classifier__max_depth': [6],
                         'classifier__n_estimators': [201],
                         'classifier__num_leaves': [63],
                         'classifier__reg_alpha': [0, 1, 10],
                         'classifier__reg_lambda': [0.1, 1, 5]},
             scoring='f1')
```

In [188…

```python
grid.best_params_
grid.best_score_
```

Out[188…

```
{'classifier__class_weight': {0: 1, 1: 3.0},
 'classifier__colsample_bytree': 0.6,
 'classifier__learning_rate': 0.1,
 'classifier__max_depth': 6,
 'classifier__n_estimators': 201,
 'classifier__num_leaves': 63,
 'classifier__reg_alpha': 1,
 'classifier__reg_lambda': 1}
```

Out[188…

```
0.6827227378996369
```

In [189…

```python
grid.cv_results_
```

Out[189…

```
{'mean_fit_time': array([1.551682  , 1.45580788, 1.59996123, 1.35228295, 1.28855724,
        1.27937255, 1.16682906, 1.22718458, 1.25055814, 1.28368692,
        1.27696409, 1.33582411, 1.37465501, 1.34594679, 1.33351326,
        1.19251485, 1.14840469, 1.1209866 ]),
 'std_fit_time': array([0.17222523, 0.18362189, 0.01284884, 0.17152396, 0.03048302,
        0.03230484, 0.01349604, 0.08480037, 0.10345779, 0.02119726,
        0.01003686, 0.06578677, 0.03867837, 0.01181787, 0.02064513,
        0.06840128, 0.01696706, 0.02072507]),
 'mean_score_time': array([0.03906202, 0.03771191, 0.04093952, 0.04086208, 0.04406519,
        0.03461056, 0.04040656, 0.03108444, 0.03445716, 0.03858347,
        0.03795414, 0.04078708, 0.03163085, 0.04060898, 0.03176899,
        0.03120394, 0.03259797, 0.02080879]),
```

```
 'std_score_time': array([0.00700396, 0.00780073, 0.00759134, 0.00599274, 0.00621384,
        0.00613468, 0.00906042, 0.00074439, 0.00793874, 0.00717185,
        0.00762982, 0.00762787, 0.00058282, 0.0067198 , 0.00463154,
        0.00060968, 0.00237596, 0.00619783]),
 'param_classifier__class_weight': masked_array(data=[{0: 1, 1: 3.0}, {0: 1, 1: 3.0}, {0:
1, 1: 3.0},
                    {0: 1, 1: 3.0}, {0: 1, 1: 3.0}, {0: 1, 1: 3.0},
                    {0: 1, 1: 3.0}, {0: 1, 1: 3.0}, {0: 1, 1: 3.0},
                    {0: 1, 1: 3.0}, {0: 1, 1: 3.0}, {0: 1, 1: 3.0},
                    {0: 1, 1: 3.0}, {0: 1, 1: 3.0}, {0: 1, 1: 3.0},
                    {0: 1, 1: 3.0}, {0: 1, 1: 3.0}, {0: 1, 1: 3.0}],
              mask=[False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False],
        fill_value='?',
             dtype=object),
 'param_classifier__colsample_bytree': masked_array(data=[0.6, 0.6, 0.6, 0.6, 0.6, 0.6, 0.
6, 0.6, 0.6, 0.8, 0.8,
                    0.8, 0.8, 0.8, 0.8, 0.8, 0.8, 0.8],
              mask=[False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False],
        fill_value='?',
             dtype=object),
 'param_classifier__learning_rate': masked_array(data=[0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1,
0.1, 0.1, 0.1, 0.1,
                    0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1],
              mask=[False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False],
        fill_value='?',
             dtype=object),
 'param_classifier__max_depth': masked_array(data=[6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6,
6, 6, 6, 6, 6],
              mask=[False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False],
        fill_value='?',
             dtype=object),
 'param_classifier__n_estimators': masked_array(data=[201, 201, 201, 201, 201, 201, 201, 2
01, 201, 201, 201,
                    201, 201, 201, 201, 201, 201, 201],
              mask=[False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False],
        fill_value='?',
             dtype=object),
 'param_classifier__num_leaves': masked_array(data=[63, 63, 63, 63, 63, 63, 63, 63, 63, 6
3, 63, 63, 63, 63,
                    63, 63, 63, 63],
              mask=[False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False],
        fill_value='?',
             dtype=object),
 'param_classifier__reg_alpha': masked_array(data=[0, 0, 0, 1, 1, 1, 10, 10, 10, 0, 0, 0,
1, 1, 1, 10, 10,
                    10],
              mask=[False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False],
        fill_value='?',
             dtype=object),
 'param_classifier__reg_lambda': masked_array(data=[0.1, 1, 5, 0.1, 1, 5, 0.1, 1, 5, 0.1,
1, 5, 0.1, 1, 5,
                    0.1, 1, 5],
```

```
             mask=[False, False, False, False, False, False, False, False,
                   False, False, False, False, False, False, False, False,
                   False, False],
       fill_value='?',
            dtype=object),
 'params': [{'classifier__class_weight': {0: 1, 1: 3.0},
   'classifier__colsample_bytree': 0.6,
   'classifier__learning_rate': 0.1,
   'classifier__max_depth': 6,
   'classifier__n_estimators': 201,
   'classifier__num_leaves': 63,
   'classifier__reg_alpha': 0,
   'classifier__reg_lambda': 0.1},
  {'classifier__class_weight': {0: 1, 1: 3.0},
   'classifier__colsample_bytree': 0.6,
   'classifier__learning_rate': 0.1,
   'classifier__max_depth': 6,
   'classifier__n_estimators': 201,
   'classifier__num_leaves': 63,
   'classifier__reg_alpha': 0,
   'classifier__reg_lambda': 1},
  {'classifier__class_weight': {0: 1, 1: 3.0},
   'classifier__colsample_bytree': 0.6,
   'classifier__learning_rate': 0.1,
   'classifier__max_depth': 6,
   'classifier__n_estimators': 201,
   'classifier__num_leaves': 63,
   'classifier__reg_alpha': 0,
   'classifier__reg_lambda': 5},
  {'classifier__class_weight': {0: 1, 1: 3.0},
   'classifier__colsample_bytree': 0.6,
   'classifier__learning_rate': 0.1,
   'classifier__max_depth': 6,
   'classifier__n_estimators': 201,
   'classifier__num_leaves': 63,
   'classifier__reg_alpha': 1,
   'classifier__reg_lambda': 0.1},
  {'classifier__class_weight': {0: 1, 1: 3.0},
   'classifier__colsample_bytree': 0.6,
   'classifier__learning_rate': 0.1,
   'classifier__max_depth': 6,
   'classifier__n_estimators': 201,
   'classifier__num_leaves': 63,
   'classifier__reg_alpha': 1,
   'classifier__reg_lambda': 1},
  {'classifier__class_weight': {0: 1, 1: 3.0},
   'classifier__colsample_bytree': 0.6,
   'classifier__learning_rate': 0.1,
   'classifier__max_depth': 6,
   'classifier__n_estimators': 201,
   'classifier__num_leaves': 63,
   'classifier__reg_alpha': 1,
   'classifier__reg_lambda': 5},
  {'classifier__class_weight': {0: 1, 1: 3.0},
   'classifier__colsample_bytree': 0.6,
   'classifier__learning_rate': 0.1,
   'classifier__max_depth': 6,
   'classifier__n_estimators': 201,
   'classifier__num_leaves': 63,
   'classifier__reg_alpha': 10,
   'classifier__reg_lambda': 0.1},
  {'classifier__class_weight': {0: 1, 1: 3.0},
   'classifier__colsample_bytree': 0.6,
   'classifier__learning_rate': 0.1,
   'classifier__max_depth': 6,
   'classifier__n_estimators': 201,
```

```
 'classifier__num_leaves': 63,
 'classifier__reg_alpha': 10,
 'classifier__reg_lambda': 1},
{'classifier__class_weight': {0: 1, 1: 3.0},
 'classifier__colsample_bytree': 0.6,
 'classifier__learning_rate': 0.1,
 'classifier__max_depth': 6,
 'classifier__n_estimators': 201,
 'classifier__num_leaves': 63,
 'classifier__reg_alpha': 10,
 'classifier__reg_lambda': 5},
{'classifier__class_weight': {0: 1, 1: 3.0},
 'classifier__colsample_bytree': 0.8,
 'classifier__learning_rate': 0.1,
 'classifier__max_depth': 6,
 'classifier__n_estimators': 201,
 'classifier__num_leaves': 63,
 'classifier__reg_alpha': 0,
 'classifier__reg_lambda': 0.1},
{'classifier__class_weight': {0: 1, 1: 3.0},
 'classifier__colsample_bytree': 0.8,
 'classifier__learning_rate': 0.1,
 'classifier__max_depth': 6,
 'classifier__n_estimators': 201,
 'classifier__num_leaves': 63,
 'classifier__reg_alpha': 0,
 'classifier__reg_lambda': 1},
{'classifier__class_weight': {0: 1, 1: 3.0},
 'classifier__colsample_bytree': 0.8,
 'classifier__learning_rate': 0.1,
 'classifier__max_depth': 6,
 'classifier__n_estimators': 201,
 'classifier__num_leaves': 63,
 'classifier__reg_alpha': 0,
 'classifier__reg_lambda': 5},
{'classifier__class_weight': {0: 1, 1: 3.0},
 'classifier__colsample_bytree': 0.8,
 'classifier__learning_rate': 0.1,
 'classifier__max_depth': 6,
 'classifier__n_estimators': 201,
 'classifier__num_leaves': 63,
 'classifier__reg_alpha': 1,
 'classifier__reg_lambda': 0.1},
{'classifier__class_weight': {0: 1, 1: 3.0},
 'classifier__colsample_bytree': 0.8,
 'classifier__learning_rate': 0.1,
 'classifier__max_depth': 6,
 'classifier__n_estimators': 201,
 'classifier__num_leaves': 63,
 'classifier__reg_alpha': 1,
 'classifier__reg_lambda': 1},
{'classifier__class_weight': {0: 1, 1: 3.0},
 'classifier__colsample_bytree': 0.8,
 'classifier__learning_rate': 0.1,
 'classifier__max_depth': 6,
 'classifier__n_estimators': 201,
 'classifier__num_leaves': 63,
 'classifier__reg_alpha': 1,
 'classifier__reg_lambda': 5},
{'classifier__class_weight': {0: 1, 1: 3.0},
 'classifier__colsample_bytree': 0.8,
 'classifier__learning_rate': 0.1,
 'classifier__max_depth': 6,
 'classifier__n_estimators': 201,
 'classifier__num_leaves': 63,
 'classifier__reg_alpha': 10,
```

```
  'classifier__reg_lambda': 0.1},
 {'classifier__class_weight': {0: 1, 1: 3.0},
  'classifier__colsample_bytree': 0.8,
  'classifier__learning_rate': 0.1,
  'classifier__max_depth': 6,
  'classifier__n_estimators': 201,
  'classifier__num_leaves': 63,
  'classifier__reg_alpha': 10,
  'classifier__reg_lambda': 1},
 {'classifier__class_weight': {0: 1, 1: 3.0},
  'classifier__colsample_bytree': 0.8,
  'classifier__learning_rate': 0.1,
  'classifier__max_depth': 6,
  'classifier__n_estimators': 201,
  'classifier__num_leaves': 63,
  'classifier__reg_alpha': 10,
  'classifier__reg_lambda': 5}],
'split0_test_score': array([0.67278287, 0.65749235, 0.67272727, 0.6779661 , 0.67384615,
       0.67781155, 0.66081871, 0.66371681, 0.66372981, 0.67697063,
       0.68711656, 0.67480916, 0.67173252, 0.66769231, 0.67269985,
       0.66863905, 0.66568915, 0.66763848]),
'split1_test_score': array([0.68072289, 0.67878788, 0.68059701, 0.67867868, 0.68350669,
       0.68017366, 0.68390805, 0.67335244, 0.66762178, 0.68468468,
       0.6935725 , 0.67751479, 0.68537666, 0.67362146, 0.66666667,
       0.67323944, 0.67134831, 0.66950355]),
'split2_test_score': array([0.68683812, 0.68358209, 0.67851852, 0.68065967, 0.6875    ,
       0.68135095, 0.68405797, 0.6851312 , 0.68405797, 0.6918429 ,
       0.67756315, 0.67941176, 0.69161677, 0.69253731, 0.68955224,
       0.6850508 , 0.6849711 , 0.68587896]),
'split3_test_score': array([0.69448584, 0.69683258, 0.69399707, 0.69448584, 0.70014771,
       0.69128508, 0.68292683, 0.69064748, 0.68847795, 0.68862275,
       0.69058296, 0.69321534, 0.68740741, 0.68537666, 0.69321534,
       0.68847795, 0.6875    , 0.68473609]),
'split4_test_score': array([0.66176471, 0.66861314, 0.66371681, 0.65885798, 0.66861314,
       0.65979381, 0.67528736, 0.65997131, 0.66191155, 0.66071429,
       0.65275708, 0.65592972, 0.65875371, 0.65878877, 0.65875371,
       0.6618705 , 0.6695279 , 0.66954023]),
'mean_test_score': array([0.67931889, 0.67706161, 0.67791134, 0.67812965, 0.68272274,
       0.67808301, 0.67739978, 0.67456385, 0.67315981, 0.68056705,
       0.68031845, 0.67617616, 0.67897741, 0.6756033 , 0.67617756,
       0.67545555, 0.67580729, 0.67545946]),
'std_test_score': array([0.01130854, 0.01334706, 0.00994677, 0.01136355, 0.01099946,
       0.01023634, 0.00890653, 0.0118526 , 0.01095019, 0.0111001 ,
       0.01479503, 0.01195309, 0.01210996, 0.01209561, 0.0132311 ,
       0.00997578, 0.00874511, 0.00807836]),
'rank_test_score': array([ 4, 10,  8,  6,  1,  7,  9, 17, 18,  2,  3, 12,  5, 14, 11, 16,
13,
       15])}
```

In [ ]:

In [ ]:

Can we do better? - Ensembles

In [190…

```python
from lightgbm import LGBMClassifier
from sklearn.pipeline import Pipeline
```

In [191…

```python
## Preparing data for error analysis
# Unscaled features will be used since it's a tree model
```

```
X_train = df_train.drop(columns = ['Exited'], axis = 1)
X_val = df_val.drop(columns = ['Exited'], axis = 1)

X_train.shape, y_train.shape
X_val.shape, y_val.shape
```

Out[191…

```
((7920, 17), (7920,))
```

Out[191…

```
((1080, 17), (1080,))
```

In [192…

```
## Three versions of the final model with best params for F1-score metric

# Equal weights to both target classes (no class imbalance correction)
lgb1 = LGBMClassifier(boosting_type = 'dart', class_weight = {0: 1, 1: 1}, min_child_sampl
                      , importance_type = 'gain', max_depth = 4, num_leaves = 31, colsample
                      , n_estimators = 21, reg_alpha = 0, reg_lambda = 0.5)

# Addressing class imbalance completely by weighting the undersampled class by the class i
lgb2 = LGBMClassifier(boosting_type = 'dart', class_weight = {0: 1, 1: 3.93}, min_child_sa
                      , importance_type = 'gain', max_depth = 6, num_leaves = 63, colsample
                      , n_estimators = 201, reg_alpha = 1, reg_lambda = 1)


# Best class_weight parameter settings (partial class imbalance correction)
lgb3 = LGBMClassifier(boosting_type = 'dart', class_weight = {0: 1, 1: 3.0}, min_child_sar
                      , importance_type = 'gain', max_depth = 6, num_leaves = 63, colsample
                      , n_estimators = 201, reg_alpha = 1, reg_lambda = 1)
```

In [193…

```
## 3 different Pipeline objects for the 3 models defined above
model_1 = Pipeline(steps = [('categorical_encoding', CategoricalEncoder()),
                            ('add_new_features', AddFeatures()),
                            ('classifier', lgb1)
                           ])

model_2 = Pipeline(steps = [('categorical_encoding', CategoricalEncoder()),
                            ('add_new_features', AddFeatures()),
                            ('classifier', lgb2)
                           ])

model_3 = Pipeline(steps = [('categorical_encoding', CategoricalEncoder()),
                            ('add_new_features', AddFeatures()),
                            ('classifier', lgb3)
                           ])
```

In [194…

```
## Fitting each of these models
model_1.fit(X_train, y_train.ravel())
model_2.fit(X_train, y_train.ravel())
model_3.fit(X_train, y_train.ravel())
```

Out[194…

```
Pipeline(steps=[('categorical_encoding',
                 CategoricalEncoder(cols=[], lcols=[], ohecols=[], tcols=[])),
                ('add_new_features', AddFeatures()),
                ('classifier',
                 LGBMClassifier(boosting_type='dart', class_weight={0: 1, 1: 1},
                                colsample_bytree=0.6, importance_type='gain',
                                max_depth=4, n_estimators=21, reg_alpha=0,
                                reg_lambda=0.5))])
```

Out[194…

```
Pipeline(steps=[('categorical_encoding',
                 CategoricalEncoder(cols=[], lcols=[], ohecols=[], tcols=[])),
                ('add_new_features', AddFeatures()),
```

```
                        ('classifier',
                         LGBMClassifier(boosting_type='dart',
                                        class_weight={0: 1, 1: 3.93},
                                        colsample_bytree=0.6, importance_type='gain',
                                        max_depth=6, n_estimators=201, num_leaves=63,
                                        reg_alpha=1, reg_lambda=1))])
```

Out[194…

```
Pipeline(steps=[('categorical_encoding',
                 CategoricalEncoder(cols=[], lcols=[], ohecols=[], tcols=[])),
                ('add_new_features', AddFeatures()),
                ('classifier',
                 LGBMClassifier(boosting_type='dart',
                                class_weight={0: 1, 1: 3.0},
                                colsample_bytree=0.6, importance_type='gain',
                                max_depth=6, n_estimators=201, num_leaves=63,
                                reg_alpha=1, reg_lambda=1))])
```

In [195…

```
## Getting prediction probabilities from each of these models
m1_pred_probs_trn = model_1.predict_proba(X_train)
m2_pred_probs_trn = model_2.predict_proba(X_train)
m3_pred_probs_trn = model_3.predict_proba(X_train)
```

In [196…

```
## Checking correlations between the predictions of the 3 models
df_t = pd.DataFrame({'m1_pred': m1_pred_probs_trn[:,1], 'm2_pred': m2_pred_probs_trn[:,1],
df_t.shape
df_t.corr()
```

Out[196…

```
(7920, 3)
```

Out[196…

	m1_pred	m2_pred	m3_pred
m1_pred	1.000000	0.894747	0.911251
m2_pred	0.894747	1.000000	0.994593
m3_pred	0.911251	0.994593	1.000000

Although models m1 and m2 are highly correlated (0.9), they are still less closely associated than m2 and m3. Thus, we'll try to form an ensemble of m1 and m2 (model averaging/stacking) and see if that improves the model accuracy

In [197…

```
## Importing relevant metric libraries
from sklearn.metrics import roc_auc_score, f1_score, recall_score, confusion_matrix, class
```

In [198…

```
## Getting prediction probabilities from each of these models
m1_pred_probs_val = model_1.predict_proba(X_val)
m2_pred_probs_val = model_2.predict_proba(X_val)
m3_pred_probs_val = model_3.predict_proba(X_val)
```

In [199…

```
threshold = 0.5
```

In [200…

```
## Best model (Model 3) predictions
m3_preds = np.where(m3_pred_probs_val[:,1] >= threshold, 1, 0)
```

In [201…

```
## Model averaging predictions (Weighted average)
m1_m2_preds = np.where(((0.1*m1_pred_probs_val[:,1]) + (0.9*m2_pred_probs_val[:,1])) >= th
```

In [202…

```
## Model 3 (Best model, tuned by GridSearch) performance on validation set
roc_auc_score(y_val, m3_preds)
recall_score(y_val, m3_preds)
confusion_matrix(y_val, m3_preds)
print(classification_report(y_val, m3_preds))
```

Out[202…

```
0.7469310764685922
```

Out[202…

```
0.592436974789916
```

Out[202…

```
array([[759,  83],
       [ 97, 141]], dtype=int64)
              precision    recall  f1-score   support

           0       0.89      0.90      0.89       842
           1       0.63      0.59      0.61       238

    accuracy                           0.83      1080
   macro avg       0.76      0.75      0.75      1080
weighted avg       0.83      0.83      0.83      1080
```

In [203…

```
## Ensemble model prediction on validation set
roc_auc_score(y_val, m1_m2_preds)
recall_score(y_val, m1_m2_preds)
confusion_matrix(y_val, m1_m2_preds)
print(classification_report(y_val, m1_m2_preds))
```

Out[203…

```
0.7586678376813908
```

Out[203…

```
0.6218487394957983
```

Out[203…

```
array([[754,  88],
       [ 90, 148]], dtype=int64)
              precision    recall  f1-score   support

           0       0.89      0.90      0.89       842
           1       0.63      0.62      0.62       238

    accuracy                           0.84      1080
   macro avg       0.76      0.76      0.76      1080
weighted avg       0.83      0.84      0.83      1080
```

In [ ]:

In [ ]:

Model stacking

The base models are the 2 LightGBM models with different class_weights parameters. They are stacked on top by a logistic regression model. Other models like linear SVM/Decision Trees can also be used. But since there are only 2 features for the model at stacking layer, it's better to use the simplest model available.

For training, we have the predictions from the 2 models on the train set. They go in as the input to the next layer of the Ensemble, which is the logistic regression model, and train the LogReg model

For prediction, we first predict using the 2 LGBM models on the validation set. The predictions from the two models go as inputs to the logistic regression which gives out the final prediction

In [204…

```
from sklearn.linear_model import LogisticRegression
```

In [205…

```
## Training
lr = LogisticRegression(C = 1.0, class_weight =  {0:1, 1:2.0})

# Concatenating the probability predictions of the 2 models on train set
X_t = np.c_[m1_pred_probs_trn[:,1],m2_pred_probs_trn[:,1]]

# Fit stacker model on top of outputs of base model
lr.fit(X_t, y_train)
```

Out[205…

```
LogisticRegression(class_weight={0: 1, 1: 2.0})
```

In [206…

```
## Prediction
# Concatenating outputs from both the base models on the validation set
X_t_val = np.c_[m1_pred_probs_val[:,1],m2_pred_probs_val[:,1]]

# Predict using the stacker model
m1_m2_preds = lr.predict(X_t_val)
```

In [207…

```
## Ensemble model prediction on validation set
roc_auc_score(y_val, m1_m2_preds)
recall_score(y_val, m1_m2_preds)
confusion_matrix(y_val, m1_m2_preds)
print(classification_report(y_val, m1_m2_preds))
```

Out[207…

```
0.7463372522405638
```

Out[207…

```
0.592436974789916
```

Out[207…

```
array([[758,  84],
       [ 97, 141]], dtype=int64)
              precision    recall  f1-score   support

           0       0.89      0.90      0.89       842
           1       0.63      0.59      0.61       238

    accuracy                           0.83      1080
   macro avg       0.76      0.75      0.75      1080
weighted avg       0.83      0.83      0.83      1080
```

In [208…

```
# Model weights learnt by the stacker LogReg model
lr.coef_
lr.intercept_
```

Out[208…

```
array([[-6.06252409, 12.94656529]])
```

Out[208…

```
array([-5.65280526])
```

In [ ]:

In [ ]:

Error analysis

In [209…

```
from lightgbm import LGBMClassifier
from sklearn.pipeline import Pipeline
```

In [210…

```
## Preparing data for error analysis
# Unscaled features will be used since it's a tree model

X_train = df_train.drop(columns = ['Exited'], axis = 1)
X_val = df_val.drop(columns = ['Exited'], axis = 1)

X_train.shape, y_train.shape
X_val.shape, y_val.shape
```

Out[210…

```
((7920, 17), (7920,))
```

Out[210…

```
((1080, 17), (1080,))
```

In [211…

```
## Final model with best params for F1-score metric

lgb = LGBMClassifier(boosting_type = 'dart', class_weight = {0: 1, 1: 3.0}, min_child_samp
                     , importance_type = 'gain', max_depth = 6, num_leaves = 63, colsample
                     , n_estimators = 201, reg_alpha = 1, reg_lambda = 1)


model = Pipeline(steps = [('categorical_encoding', CategoricalEncoder()),
                          ('add_new_features', AddFeatures()),
                          ('classifier', lgb)
                         ])
```

In [212…

```
## Fit best model
model.fit(X_train, y_train.ravel())
```

Out[212…

```
Pipeline(steps=[('categorical_encoding',
                 CategoricalEncoder(cols=[], lcols=[], ohecols=[], tcols=[])),
                ('add_new_features', AddFeatures()),
                ('classifier',
                 LGBMClassifier(boosting_type='dart',
                                class_weight={0: 1, 1: 3.0},
                                colsample_bytree=0.6, importance_type='gain',
                                max_depth=6, n_estimators=201, num_leaves=63,
                                reg_alpha=1, reg_lambda=1))])
```

In [213…

```
## Making predictions on a copy of validation set
df_ea = df_val.copy()
df_ea['y_pred'] = model.predict(X_val)
df_ea['y_pred_prob'] = model.predict_proba(X_val)[:,1]
```

In [214…

```
df_ea.shape
df_ea.sample(5)
```

Out[214…

```
(1080, 20)
```

Out[214…

	CreditScore	Gender	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsActiveMember	EstimatedSalary
181	779	0	42	5	0.00	2	0	0	25951.91
1029	569	1	35	10	124525.52	1	1	1	193793.78
516	662	0	72	7	140301.72	1	0	1	179258.67
570	709	0	32	2	87814.89	1	1	0	138578.37

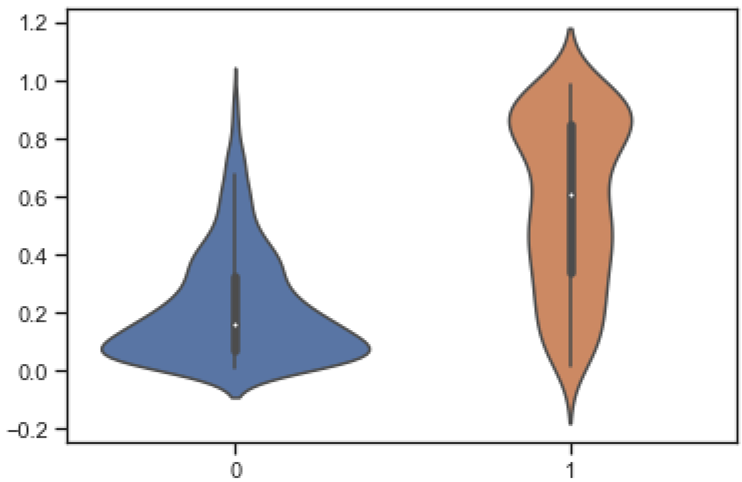
	CreditScore	Gender	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsActiveMember	EstimatedSalary
471	492	1	45	9	170295.04	2	0	0	164741.81

In [215…

```
## Visualizing distribution of predicted probabilities
sns.violinplot(y_val.ravel(), df_ea['y_pred_prob'].values)
```

Out[215…

```
<AxesSubplot:>
```



Revisiting bivariate plots of important features

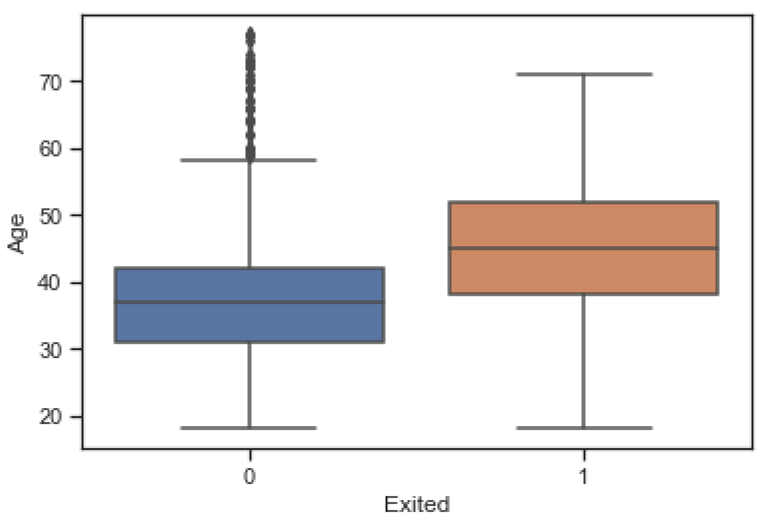
The difference in distribution of these features across the two classes help us to test a few hypotheses

In [216…

```
sns.boxplot(x = 'Exited', y = 'Age', data = df_ea)
```

Out[216…

```
<AxesSubplot:xlabel='Exited', ylabel='Age'>
```



In [217…

```
## Are we able to correctly identify pockets of high-churn customer regions in feature spa
df_ea.Exited.value_counts(normalize=True).sort_index()
df_ea[(df_ea.Age > 42) & (df_ea.Age < 53)].Exited.value_counts(normalize=True).sort_index
df_ea[(df_ea.Age > 42) & (df_ea.Age < 53)].y_pred.value_counts(normalize=True).sort_index
```

Out[217…

```
0    0.77963
1    0.22037
Name: Exited, dtype: float64
```

Out[217…

```
0    0.560185
1    0.439815
Name: Exited, dtype: float64
```

Out[217…
```
0    0.481481
1    0.518519
Name: y_pred, dtype: float64
```

In [218…
```
## Checking correlation between features and target variable vs predicted variable
x = df_ea[num_feats + ['y_pred', 'Exited']].corr()
x[['y_pred','Exited']]
```

Out[218…

	y_pred	Exited
CreditScore	-0.016600	-0.026118
Age	0.364415	0.290853
Tenure	-0.015095	-0.011182
Balance	0.065750	0.128656
NumOfProducts	-0.150982	-0.125494
EstimatedSalary	0.006502	-0.007971
y_pred	1.000000	0.504881
Exited	0.504881	1.000000

Extracting the subset of incorrect predictions

All incorrect predictions are extracted and categorized into false positives (low precision) and false negatives (low recall)

In [219…
```
low_recall = df_ea[(df_ea.Exited == 1) & (df_ea.y_pred == 0)]
low_prec = df_ea[(df_ea.Exited == 0) & (df_ea.y_pred == 1)]
low_recall.shape
low_prec.shape
low_recall.head()
low_prec.head()
```

Out[219…
```
(97, 20)
```

Out[219…
```
(83, 20)
```

Out[219…

	CreditScore	Gender	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsActiveMember	EstimatedSalary	Ex
5	706	0	23	5	0.00	1	0	0	164128.41	
21	611	1	35	10	0.00	1	1	1	23598.23	
38	491	0	68	1	95039.12	1	0	1	116471.14	
58	637	1	43	1	135645.29	2	0	1	101382.86	
92	717	0	36	2	99472.76	2	1	0	94274.72	

Out[219…

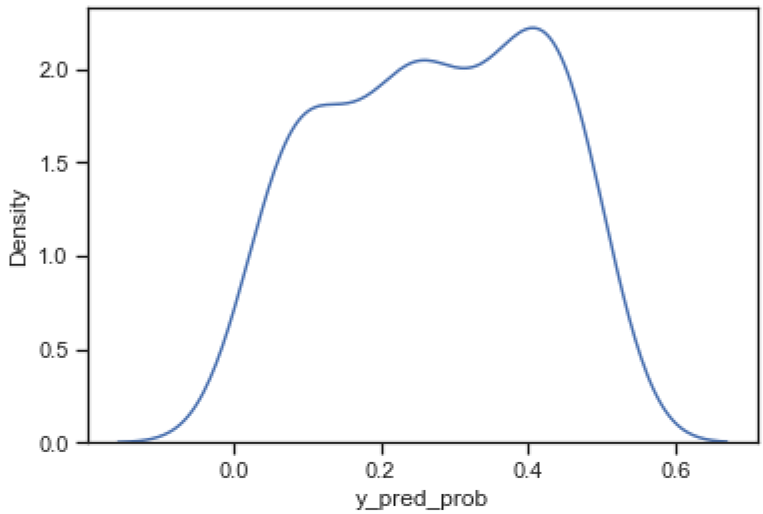
	CreditScore	Gender	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsActiveMember	EstimatedSalary	Ex
48	512	1	39	3	0.00	1	1	0	134878.19	
49	736	1	43	4	202443.47	1	1	0	72375.03	
57	505	1	43	6	127146.68	1	0	0	137565.87	
75	648	1	41	5	123049.21	1	0	1	5066.76	
99	631	1	51	8	100654.80	1	1	0	171587.90	

In [220…

```
## Prediction probabilty distribution of errors causing low recall
sns.distplot(low_recall.y_pred_prob, hist=False)
```

Out[220…

```
<AxesSubplot:xlabel='y_pred_prob', ylabel='Density'>
```

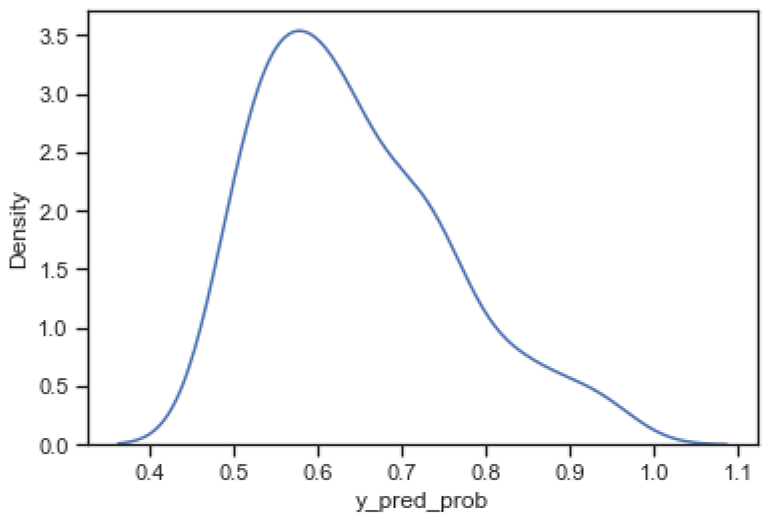


In [221…

```
## Prediction probabilty distribution of errors causing low precision
sns.distplot(low_prec.y_pred_prob, hist=False)
```

Out[221…

```
<AxesSubplot:xlabel='y_pred_prob', ylabel='Density'>
```



In [ ]:

Tweaking the threshold of classifier

In [222…

```
threshold = 0.55
```

In [223…

```
## Predict on validation set with adjustable decision threshold
probs = model.predict_proba(X_val)[:,1]
val_preds = np.where(probs > threshold, 1, 0)
```

In [224…

```
## Default params : 0.5 threshold
confusion_matrix(y_val, val_preds)
print(classification_report(y_val, val_preds))
```

Out[224…
```
array([[778,  64],
       [110, 128]], dtype=int64)
              precision    recall  f1-score   support

           0       0.88      0.92      0.90       842
           1       0.67      0.54      0.60       238

    accuracy                           0.84      1080
   macro avg       0.77      0.73      0.75      1080
weighted avg       0.83      0.84      0.83      1080
```

In [225…
```python
## Tweaking threshold between 0.4 and 0.6
confusion_matrix(y_val, val_preds)
print(classification_report(y_val, val_preds))
```

Out[225…
```
array([[778,  64],
       [110, 128]], dtype=int64)
              precision    recall  f1-score   support

           0       0.88      0.92      0.90       842
           1       0.67      0.54      0.60       238

    accuracy                           0.84      1080
   macro avg       0.77      0.73      0.75      1080
weighted avg       0.83      0.84      0.83      1080
```

Checking whether there's too much dependence on certain features

We'll compare a few important features : NumOfProducts, IsActiveMember, Age, Balance

In [226…
```python
df_ea.NumOfProducts.value_counts(normalize=True).sort_index()
low_recall.NumOfProducts.value_counts(normalize=True).sort_index()
low_prec.NumOfProducts.value_counts(normalize=True).sort_index()
```

Out[226…
```
1    0.506481
2    0.467593
3    0.020370
4    0.005556
Name: NumOfProducts, dtype: float64
```

Out[226…
```
1    0.701031
2    0.288660
3    0.010309
Name: NumOfProducts, dtype: float64
```

Out[226…
```
1    0.819277
2    0.156627
3    0.024096
Name: NumOfProducts, dtype: float64
```

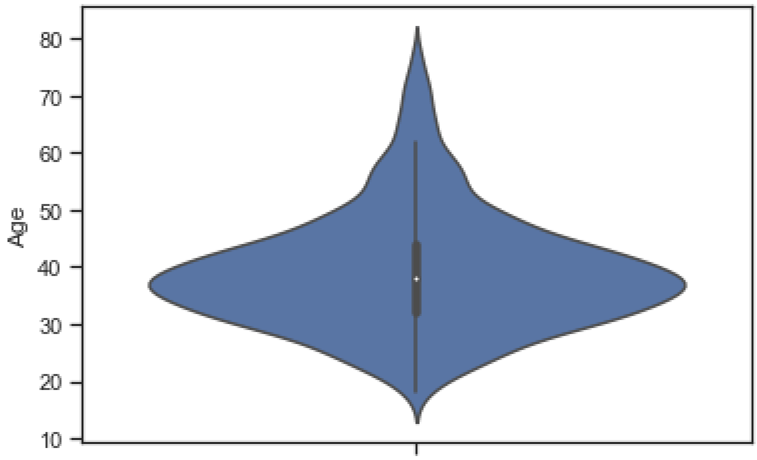
In [227…
```python
df_ea.IsActiveMember.value_counts(normalize=True).sort_index()
low_recall.IsActiveMember.value_counts(normalize=True).sort_index()
low_prec.IsActiveMember.value_counts(normalize=True).sort_index()
```

Out[227…
```
0    0.481481
1    0.518519
Name: IsActiveMember, dtype: float64
```

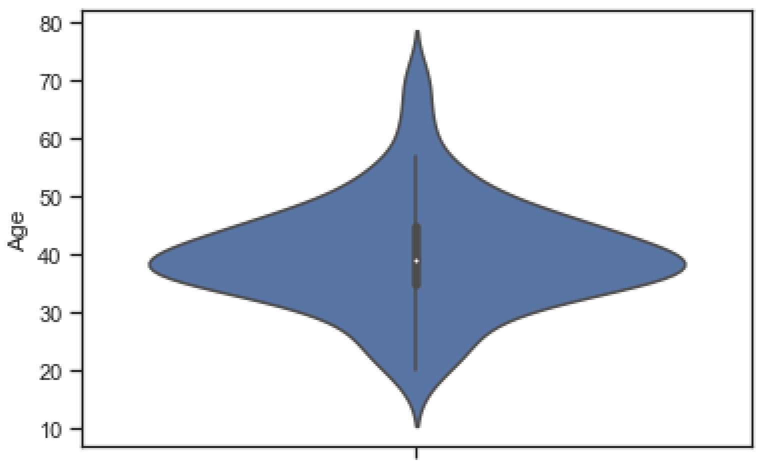
Out[227…
```
0    0.556701
1    0.443299
Name: IsActiveMember, dtype: float64
```

Out[227…
```
0    0.626506
1    0.373494
Name: IsActiveMember, dtype: float64
```

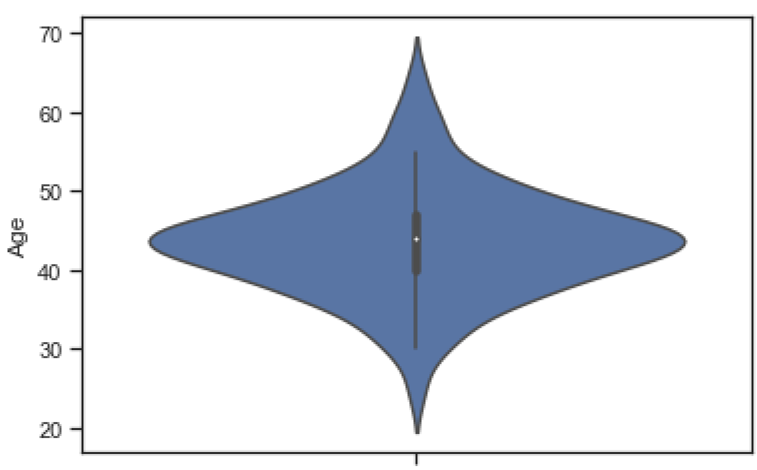
In [228…
```
sns.violinplot(y = df_ea.Age)
```
Out[228… <AxesSubplot:ylabel='Age'>



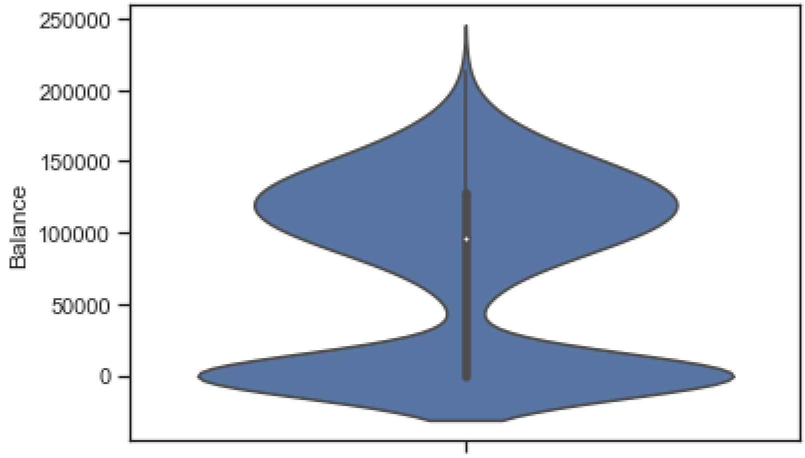
In [229…
```
sns.violinplot(y = low_recall.Age)
```
Out[229… <AxesSubplot:ylabel='Age'>



In [230…
```
sns.violinplot(y = low_prec.Age)
```
Out[230… <AxesSubplot:ylabel='Age'>



In [231…
```
sns.violinplot(y = df_ea.Balance)
```
Out[231… <AxesSubplot:ylabel='Balance'>

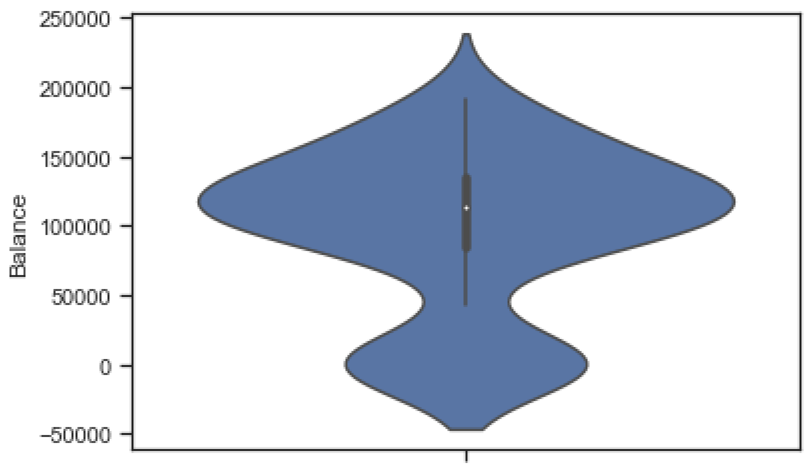


In [232…

```
sns.violinplot(y = low_recall.Balance)
```

Out[232…

```
<AxesSubplot:ylabel='Balance'>
```

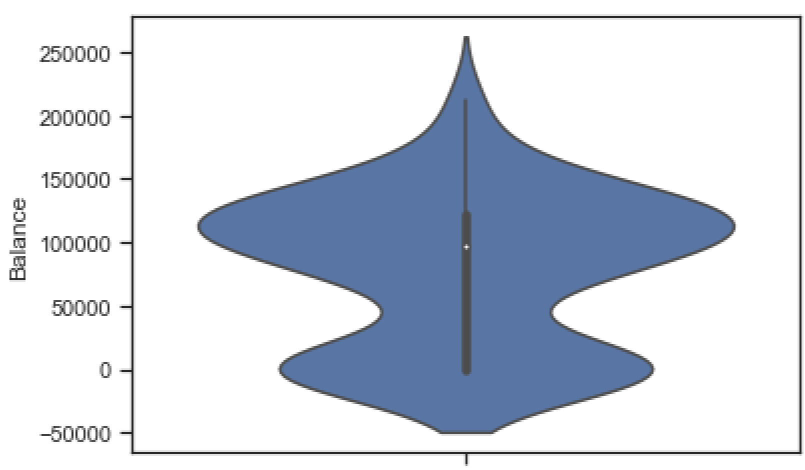


In [233…

```
sns.violinplot(y = low_prec.Balance)
```

Out[233…

```
<AxesSubplot:ylabel='Balance'>
```



In [ ]:

In [ ]:

Train final, best model ; Save model and its parameters

In [234…

```
from sklearn.pipeline import Pipeline
from lightgbm import LGBMClassifier
from sklearn.metrics import roc_auc_score, f1_score, recall_score, confusion_matrix, class
import joblib
```

In [235…

```
## Re-defining X_train and X_val to consider original unscaled continuous features. y_trai
X_train = df_train.drop(columns = ['Exited'], axis = 1)
X_val = df_val.drop(columns = ['Exited'], axis = 1)

X_train.shape, y_train.shape
X_val.shape, y_val.shape
```

Out[235…

```
((7920, 17), (7920,))
```

Out[235…

```
((1080, 17), (1080,))
```

In [236…

```
best_f1_lgb = LGBMClassifier(boosting_type = 'dart', class_weight = {0: 1, 1: 3.0}, min_ch
                       , importance_type = 'gain', max_depth = 6, num_leaves = 63, colsample
                       , n_estimators = 201, reg_alpha = 1, reg_lambda = 1)
```

In [237…

```
best_recall_lgb = LGBMClassifier(boosting_type='dart', num_leaves=31, max_depth= 6, learni
                                 , class_weight= {0: 1, 1: 3.93}, min_child_samples=2, col
                                 , reg_lambda=1.0, n_jobs=- 1, importance_type = 'gain')
```

In [238…

```
model = Pipeline(steps = [('categorical_encoding', CategoricalEncoder()),
                          ('add_new_features', AddFeatures()),
                          ('classifier', best_f1_lgb)
                         ])
```

In [239…

```
## Fitting final model on train dataset
model.fit(X_train, y_train)
```

Out[239…

```
Pipeline(steps=[('categorical_encoding',
                 CategoricalEncoder(cols=[], lcols=[], ohecols=[], tcols=[])),
                ('add_new_features', AddFeatures()),
                ('classifier',
                 LGBMClassifier(boosting_type='dart',
                                class_weight={0: 1, 1: 3.0},
                                colsample_bytree=0.6, importance_type='gain',
                                max_depth=6, n_estimators=201, num_leaves=63,
                                reg_alpha=1, reg_lambda=1))])
```

In [240…

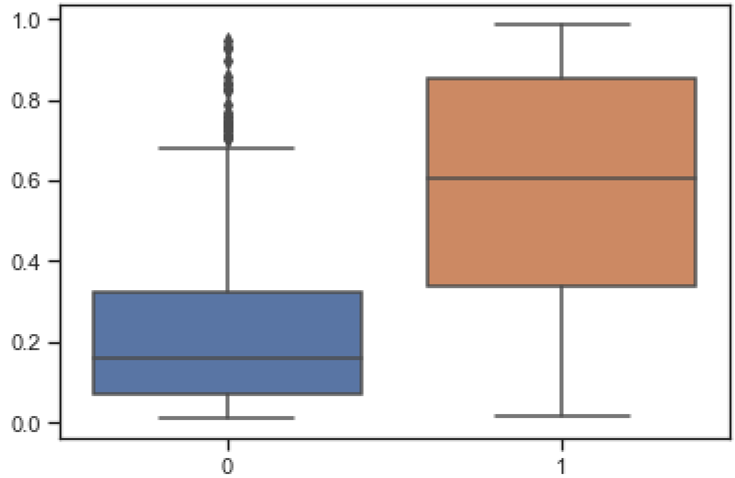
```
# Predict target probabilities
val_probs = model.predict_proba(X_val)[:,1]

# Predict target values on val data
val_preds = np.where(val_probs > 0.45, 1, 0) # The probability threshold can be tweaked
```

In [241…

```
sns.boxplot(y_val.ravel(), val_probs)
```

Out[241…

```
<AxesSubplot:>
```



In [242…

```
## Validation metrics
roc_auc_score(y_val, val_preds)
recall_score(y_val, val_preds)
confusion_matrix(y_val, val_preds)
print(classification_report(y_val, val_preds))
```

Out[242…

```
0.7587576598335297
```

Out[242…

```
0.6386554621848739
```

Out[242…

```
array([[740, 102],
       [ 86, 152]], dtype=int64)
              precision    recall  f1-score   support

           0       0.90      0.88      0.89       842
           1       0.60      0.64      0.62       238

    accuracy                           0.83      1080
   macro avg       0.75      0.76      0.75      1080
weighted avg       0.83      0.83      0.83      1080
```

In [243…

```
## Save model object
joblib.dump(model, 'final_churn_model_f1_0_45.sav')
```

Out[243…

```
['final_churn_model_f1_0_45.sav']
```

SHAP

SHAP paper : https://papers.nips.cc/paper/7062-a-unified-approach-to-interpreting-model-predictions.pdf

In [ ]:

```
```

In [244…

```
import shap

shap.initjs()
```

js

In [245…

```
ce = CategoricalEncoder()
af = AddFeatures()
```

```
X = ce.fit_transform(X_train, y_train)
X = af.transform(X)
```

In [246…

```
X.shape
X.sample(5)
```

Out[246…

```
(7920, 18)
```

Out[246…

	CreditScore	Gender	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsActiveMember	EstimatedSalary
5248	821	1	42	3	87807.29	2	1	1	64613.81
5693	659	0	38	9	0.00	2	1	1	132809.18
3859	511	1	45	5	68375.27	1	1	0	193160.25
7815	668	1	42	8	187534.79	1	1	1	32900.41
5928	690	1	47	2	0.00	2	1	0	151375.73

In [247…

```
best_f1_lgb.fit(X, y_train)
```

Out[247…

```
LGBMClassifier(boosting_type='dart', class_weight={0: 1, 1: 3.0},
               colsample_bytree=0.6, importance_type='gain', max_depth=6,
               n_estimators=201, num_leaves=63, reg_alpha=1, reg_lambda=1)
```

In [248…

```
explainer = shap.TreeExplainer(best_f1_lgb)
```

In [249…

```
X.head(10)
```

Out[249…

	CreditScore	Gender	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsActiveMember	EstimatedSalary	cou
0	678	1	36	1	117864.85	2	1	0	27619.06	
1	613	0	27	5	125167.74	1	1	0	199104.52	
2	628	1	45	9	0.00	2	1	1	96862.56	
3	513	1	30	5	0.00	2	1	0	162523.66	
4	639	1	22	4	0.00	2	1	0	28188.96	
5	562	1	30	3	111099.79	2	0	0	140650.19	
6	635	1	43	5	78992.75	2	0	0	153265.31	
7	705	1	33	7	68423.89	1	1	1	64872.55	
8	694	1	42	8	133767.19	1	1	0	36405.21	
9	711	1	26	9	128793.63	1	1	0	19262.05	

In [250…

```
row_num = 7
shap_vals = explainer.shap_values(X.iloc[row_num].values.reshape(1,-1))
```

In [251…

```
#base value
explainer.expected_value
```

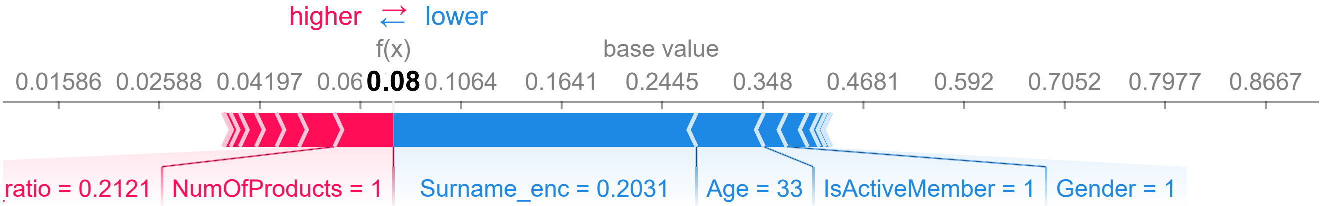
Out[251…

```
[1.1279613498396024, -1.1279613498396024]
```

In [252…

```
## Explain single prediction
shap.force_plot(explainer.expected_value[1], shap_vals[1], X.iloc[row_num], link = 'logit'
```

Out[252…



In [253…

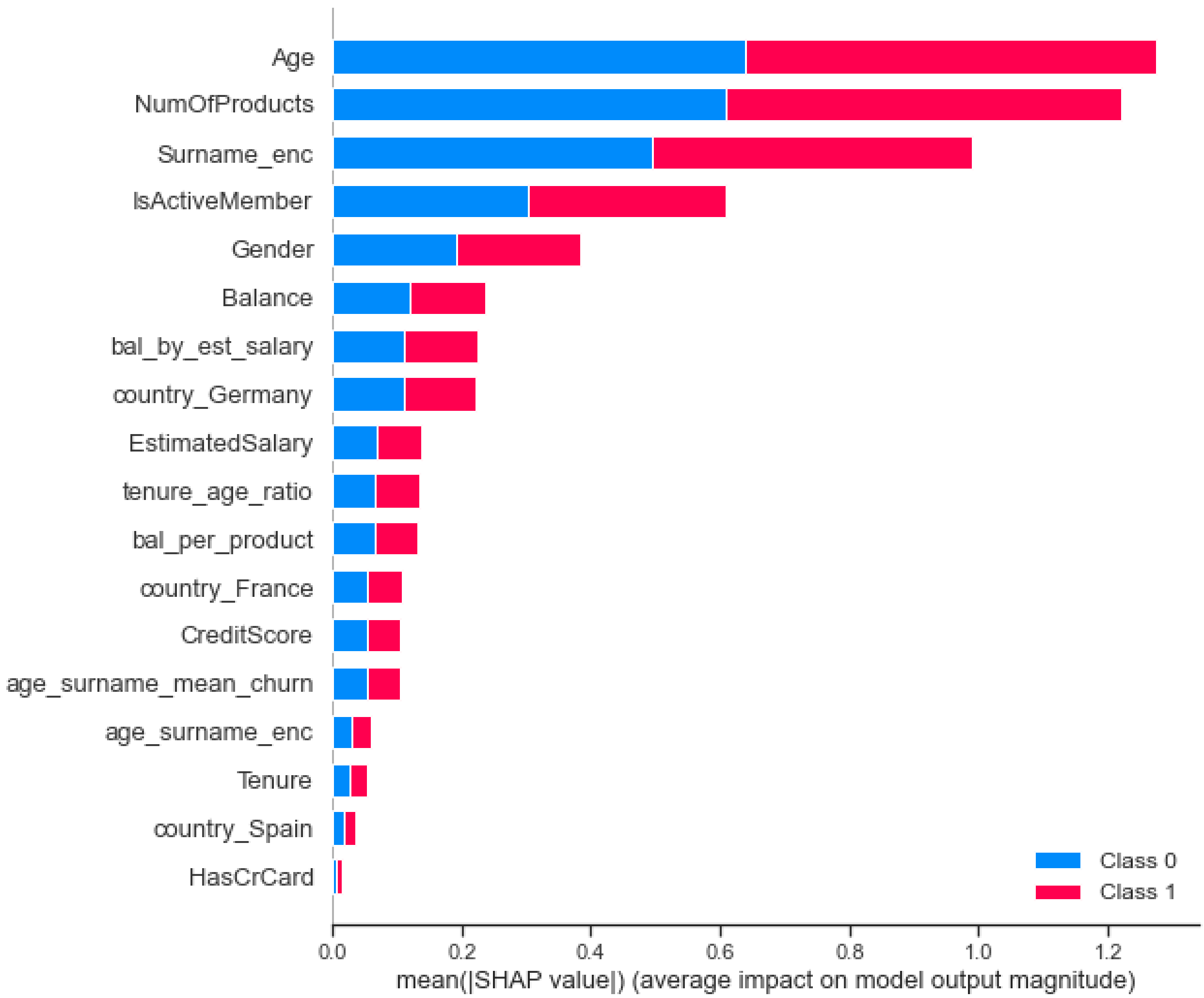
```
## Check probability predictions through the model
pred_probs = best_f1_lgb.predict_proba(X)[:,1]
pred_probs[row_num]
```

Out[253…

```
0.07878111194117235
```

In [254…

```
## Explain global patterns/ summary stats
shap_values = explainer.shap_values(X)
shap.summary_plot(shap_values, X)
```



Load saved model and make predictions on unseen/future data

Here, we'll use df_test as the unseen, future data

In [255…
```python
import joblib
```

In [256…
```python
## Load model object
model = joblib.load('final_churn_model_f1_0_45.sav')
```

In [257…
```python
X_test = df_test.drop(columns = ['Exited'], axis = 1)
X_test.shape
y_test.shape
```

Out[257…
```
(1000, 17)
```

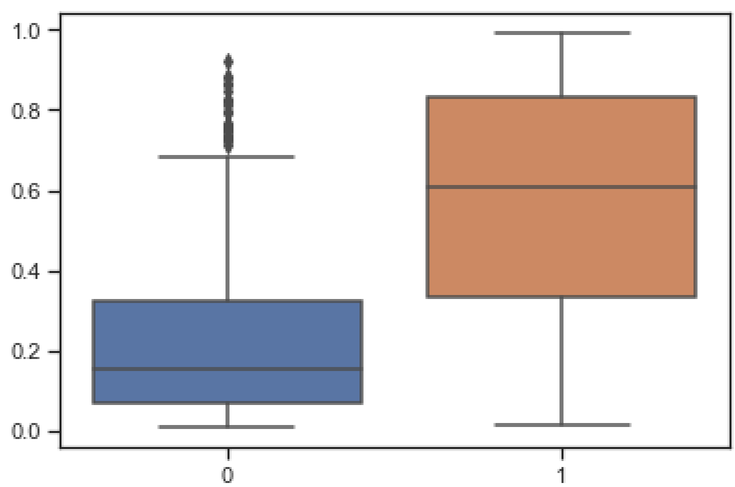
Out[257…
```
(1000,)
```

In [258…
```python
## Predict target probabilities
test_probs = model.predict_proba(X_test)[:,1]
```

In [259…
```python
## Predict target values on test data
test_preds = np.where(test_probs > 0.45, 1, 0) # Flexibility to tweak the probability thre
#test_preds = model.predict(X_test)
```

In [260…
```python
sns.boxplot(y_test.ravel(), test_probs)
```

Out[260…
```
<AxesSubplot:>
```



In [261…
```python
## Test set metrics
roc_auc_score(y_test, test_preds)
recall_score(y_test, test_preds)
confusion_matrix(y_test, test_preds)
print(classification_report(y_test, test_preds))
```

Out[261…
```
0.7678570272911421
```

Out[261…
```
0.675392670157068
```

Out[261…
```
array([[696, 113],
       [ 62, 129]], dtype=int64)
              precision    recall  f1-score   support

           0       0.92      0.86      0.89       809
           1       0.53      0.68      0.60       191
```

```
    accuracy                           0.82      1000
   macro avg       0.73      0.77      0.74      1000
weighted avg       0.84      0.82      0.83      1000
```

In [262…

```
## Adding predictions and their probabilities in the original test dataframe
test = df_test.copy()
test['predictions'] = test_preds
test['pred_probabilities'] = test_probs
```

In [263…

```
test.sample(10)
```

Out[263…

	CreditScore	Gender	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsActiveMember	EstimatedSalary	E
869	629	0	44	6	125512.98	2	0	0	79082.76	
650	559	1	49	2	147069.78	1	1	0	120540.83	
228	692	1	66	4	159732.02	1	1	1	118188.15	
281	646	0	46	4	0.00	3	1	0	93251.42	
59	505	1	40	6	47869.69	2	1	1	155061.97	
692	670	0	42	1	115961.58	2	0	1	29483.87	
662	521	0	40	9	134504.78	1	1	0	18082.06	
800	480	0	42	1	152160.21	2	1	0	101778.90	
167	485	0	39	2	75339.64	1	1	1	70665.16	
338	643	1	34	6	0.00	2	1	1	116046.22	

Creating a list of customers who are the most likely to churn

Listing customers who have a churn probability higher than 70%. These are the ones who can be targeted immediately

In [264…

```
high_churn_list = test[test.pred_probabilities > 0.7].sort_values(by = ['pred_probabilitie
                                                                    ).reset_index().drop(colu
```

In [265…

```
high_churn_list.shape
high_churn_list.head()
```

Out[265…

```
(103, 18)
```

Out[265…

	CreditScore	Gender	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsActiveMember	EstimatedSalary	cou
0	546	0	58	3	106458.31	4	1	0	128881.87	
1	479	1	51	1	107714.74	3	1	0	86128.21	
2	745	1	45	10	117231.63	3	1	1	122381.02	
3	515	1	45	7	120961.50	3	1	1	39288.11	
4	481	0	57	9	0.00	3	1	1	169719.35	

In [266…

```
high_churn_list.to_csv('high_churn_list.csv', index = False)
```

Feature-based user segments from the above list

Based on business requirements, a prioritization matrix can be defined, wherein certain segments of customers are targeted first. These segments can be defined based on insights through data or the business teams' requirements. E.g. Males who are an ActiveMember, have a CreditCard and are from Germany can be prioritized first because the business potentially sees the max. ROI from them

Ending notes

Note on common issues with a model in production

- Data drift / Covariate shift
- Importance of incremental training
- Ensure parity between training and testing environments (model and library versions etc.)
- Tracking core business metrics
- Creation and monitoring of metrics of specific user segments
- Highlight impact to business folks : Through visualizations, Model can potentially reduce the Churn rate by 30-40% etc.

Future steps

- The model can be expanded to predict when will a customer churn. This will further help sales/customer service teams to reduce churn rate by targeting the right customers at the right time

In [ ]:

```

```