# PROJECT TITLE

# CUSTOMER-CHURN-PREDICTION-ANALYSIS-USING-ML-ENSEMBLE-TECHNIQUES

**NAME: ANKIT JAVERI**
**CLASS: SY MSC CS WITH SPECIALIZATION IN AI**
**ROLL NO: 01**
**SEMESTER: 4**

# PROBLEM STATEMENT

- Problem statement :

- Bank 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

(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 etc.

# LITERATURE SURVEY

- Different customer dataset will have different customer churn predictions and based on the
- machine learning model being used it is able to predict the customer churn prediction.

- Customer churn (or customer attrition) is a tendency of customers to abandon a brand and stop
- being a paying client of a particular business. The percentage of customers that discontinue using
- a company's products or services during a particular time period is called a customer churn
- (attrition) rate.

- Recursive feature elimination (RFE) is the process of selecting features sequentially, in which
- features are removed one at a time, or a few at a time, iteration after iteration.

- RFE initial steps:
- Train a machine learning model
- Derive feature importance
- Remove least important feature(s)
- Re-train the machine learning model on the remaining features
- Impact of customer churn on businesses

# METHEDOLOGY

- INSTALL LIBRARIES THROUGH PIP INSTALL:

```
!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==1.20.1
!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
```

# IMPORTING REQUIRED LIBRARIES

```python
### Import required libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

```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)
```

```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)
```

# DATA CLEANING(PERFORMING EXPLORATORY DATA ANALYSIS)

## PERFORMING EXPLORATORY DATA ANALYSIS

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

| | RowNumber | CustomerId | CreditScore | Age | Tenure | Balance | NumOfProducts | HasCrCard | IsActiveMember | EstimatedSal |
|---|---|---|---|---|---|---|---|---|---|---|
| **count** | 10000.00000 | 1.000000e+04 | 10000.000000 | 10000.000000 | 10000.000000 | 10000.000000 | 10000.000000 | 10000.00000 | 10000.000000 | 10000.000 |
| **mean** | 5000.50000 | 1.569094e+07 | 650.528800 | 38.921800 | 5.012800 | 76485.889288 | 1.530200 | 0.70550 | 0.515100 | 100090.239 |
| **std** | 2886.89568 | 7.193619e+04 | 96.653299 | 10.487806 | 2.892174 | 62397.405202 | 0.581654 | 0.45584 | 0.499797 | 57510.492 |
| **min** | 1.00000 | 1.556570e+07 | 350.000000 | 18.000000 | 0.000000 | 0.000000 | 1.000000 | 0.00000 | 0.000000 | 11.580 |
| **25%** | 2500.75000 | 1.562853e+07 | 584.000000 | 32.000000 | 3.000000 | 0.000000 | 1.000000 | 0.00000 | 0.000000 | 51002.110 |
| **50%** | 5000.50000 | 1.569074e+07 | 652.000000 | 37.000000 | 5.000000 | 97198.540000 | 1.000000 | 1.00000 | 1.000000 | 100193.915 |
| **75%** | 7500.25000 | 1.575323e+07 | 718.000000 | 44.000000 | 7.000000 | 127644.240000 | 2.000000 | 1.00000 | 1.000000 | 149388.247 |
| **max** | 10000.00000 | 1.581569e+07 | 850.000000 | 92.000000 | 10.000000 | 250898.090000 | 4.000000 | 1.00000 | 1.000000 | 199992.480 |

# PERFORMING TRAIN-TEST SPLIT

## 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

```
[17] from sklearn.model_selection import train_test_split
```

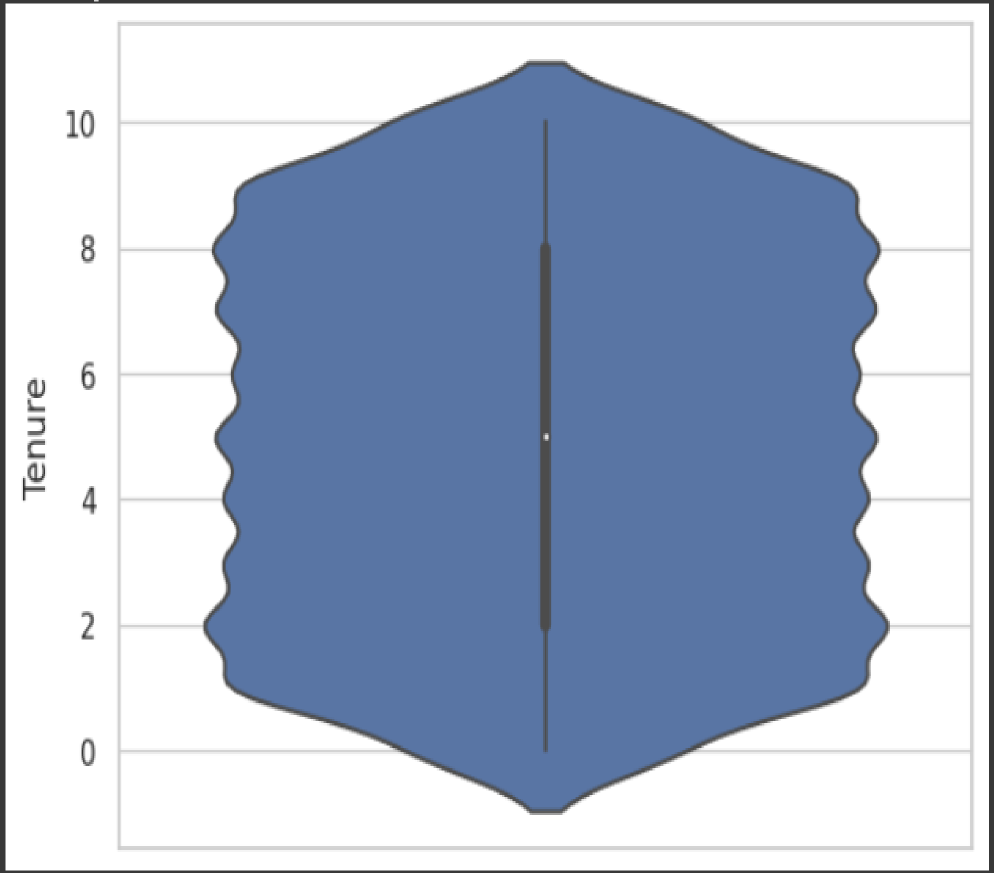
```
[18] ## Keeping aside a test/holdout set
     df_train_val, df_test, y_train_val, y_test = train_test_split(df, y.ravel(), test_size = 0.1, random_state = 42)

     ## 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 = 0.12, random_state = 42)
```

```
[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)

     ((7920, 12), (1080, 12), (1000, 12), (7920,), (1080,), (1000,))(0.20303030303030303, 0.22037037037037038, 0.191)
```

# UNIVARIATE PLOTS OF NUMERICAL VARIABLES IN TRAINING SET

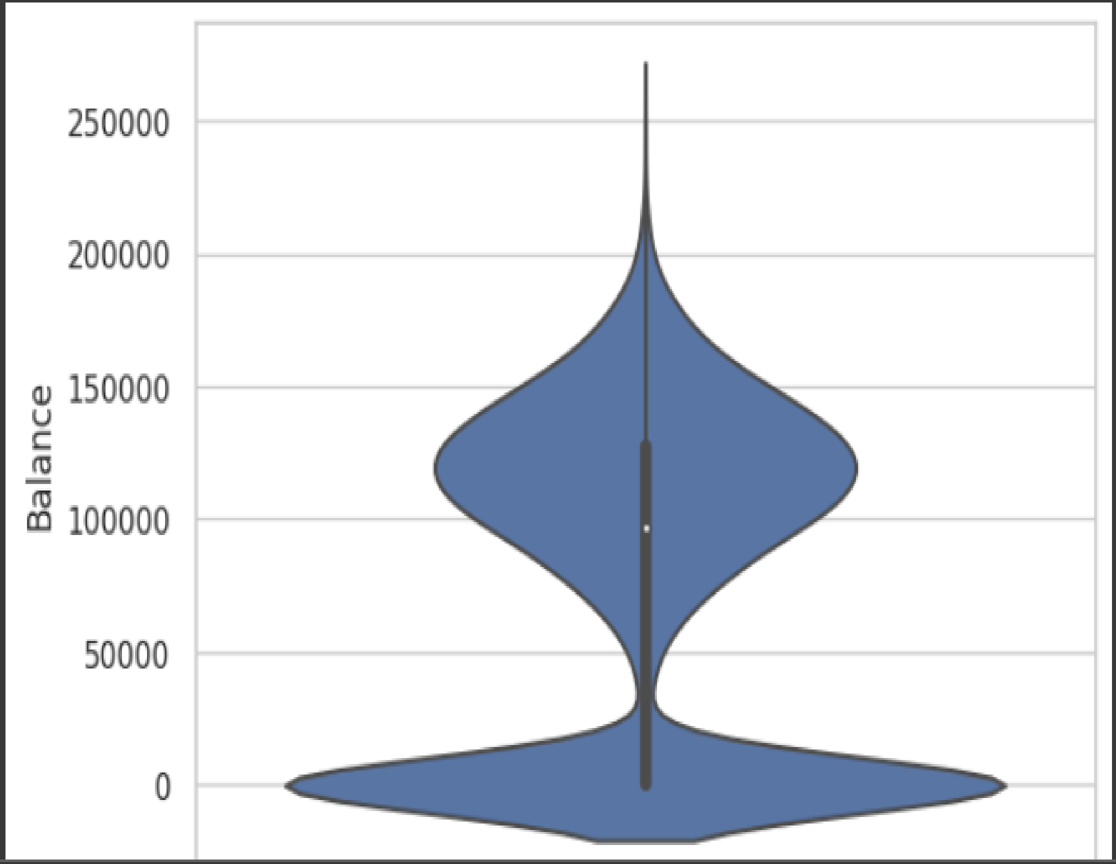
```
[22] ## Tenure
     sns.violinplot(y = df_train.Tenure)
```

<Axes: ylabel='Tenure'>




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

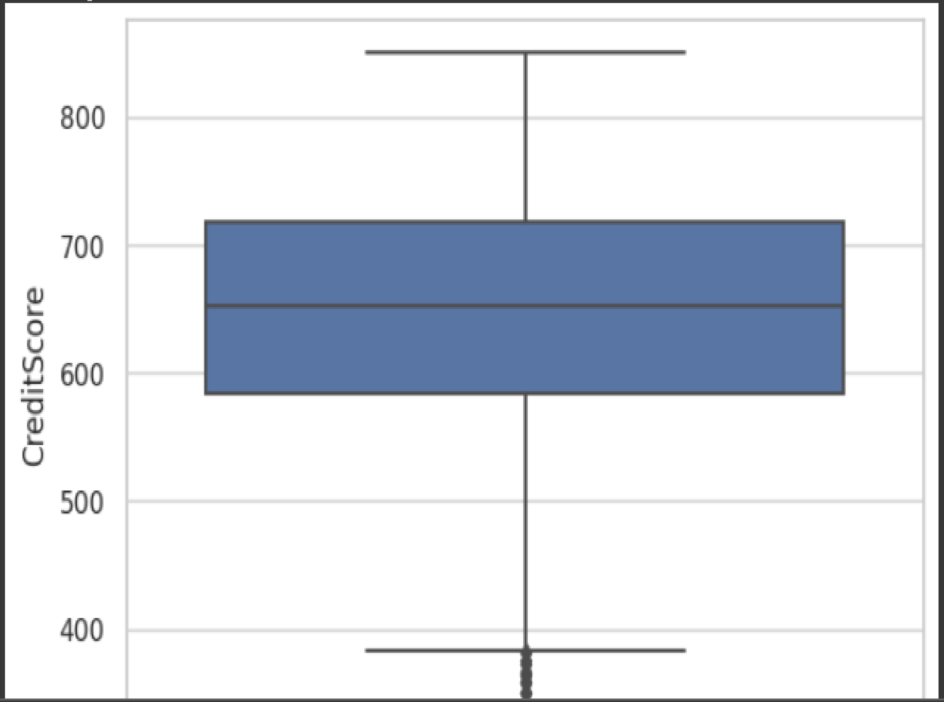
<Axes: ylabel='Balance'>



# UNIVARIATE PLOTS OF NUMERICAL VARIABLES IN TRAINING SET

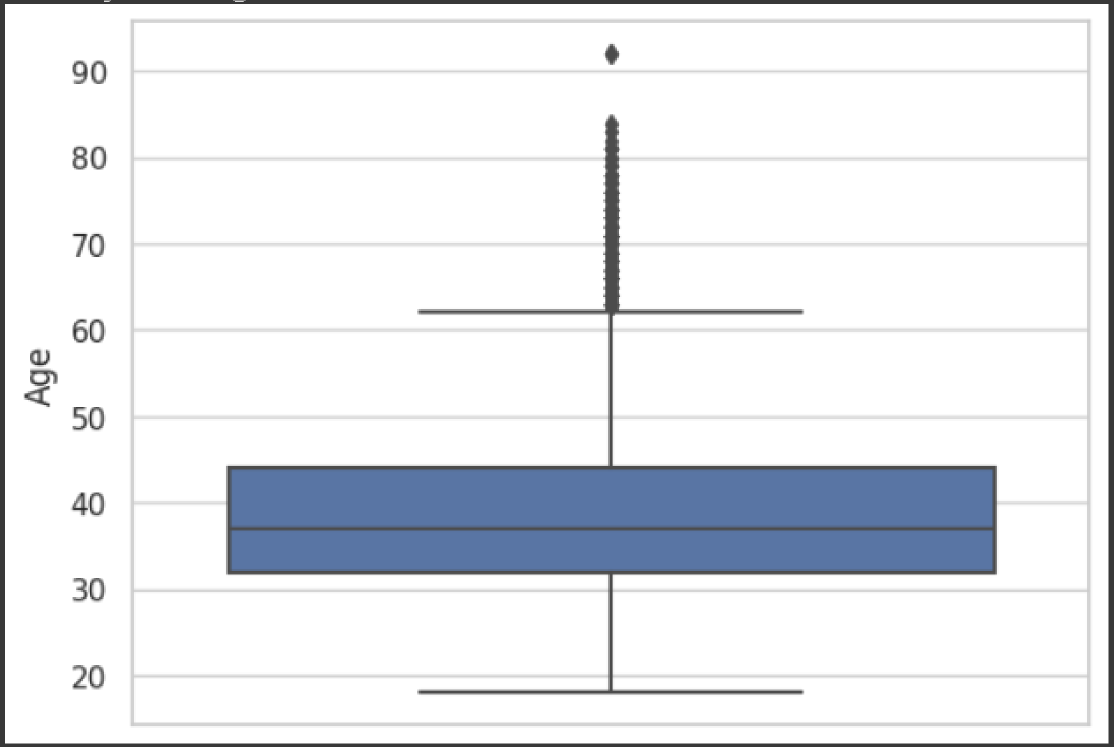
```
[20] ## CreditScore
     sns.set(style="whitegrid")
     sns.boxplot(y = df_train['CreditScore'])
```

<Axes: ylabel='CreditScore'>




```
1] ### Age
   sns.boxplot(y = df_train['Age'])
```

<Axes: ylabel='Age'>



# MISSING VALUES AND OUTLIER TREATMENT

OUTLIERS:

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.

# MISSING VALUES AND OUTLIER TREATMENT

## MISSING VALUES:

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

      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
```

# MISSING VALUES AND OUTLIER TREATMENT

```
### 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
```

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

```
Surname            0.00
CreditScore        0.00
Geography          0.30
Gender             0.00
Age                0.10
Tenure             0.00
Balance            0.00
```

# REMOVING NULL VALUES AND CALCULATING STATISTICAL VALUES

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

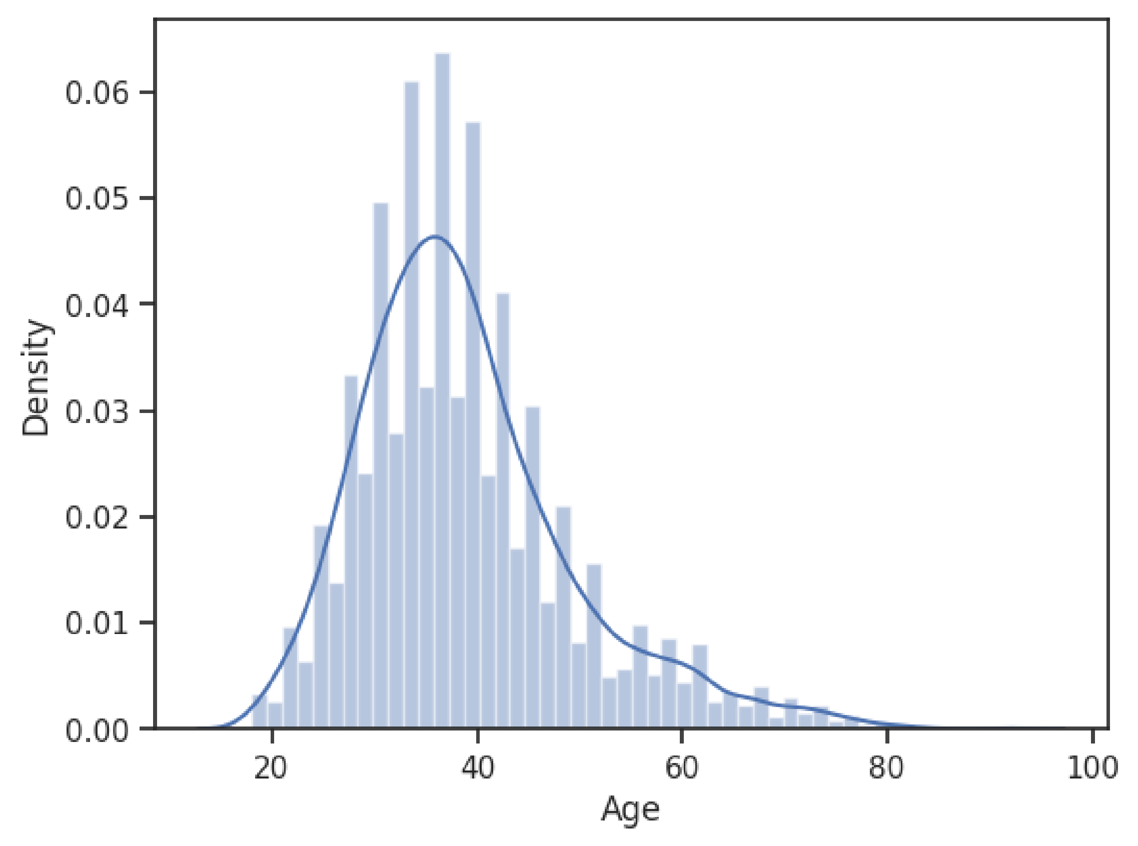
```
[31] age_mean
```

```
38.944725028058365
```

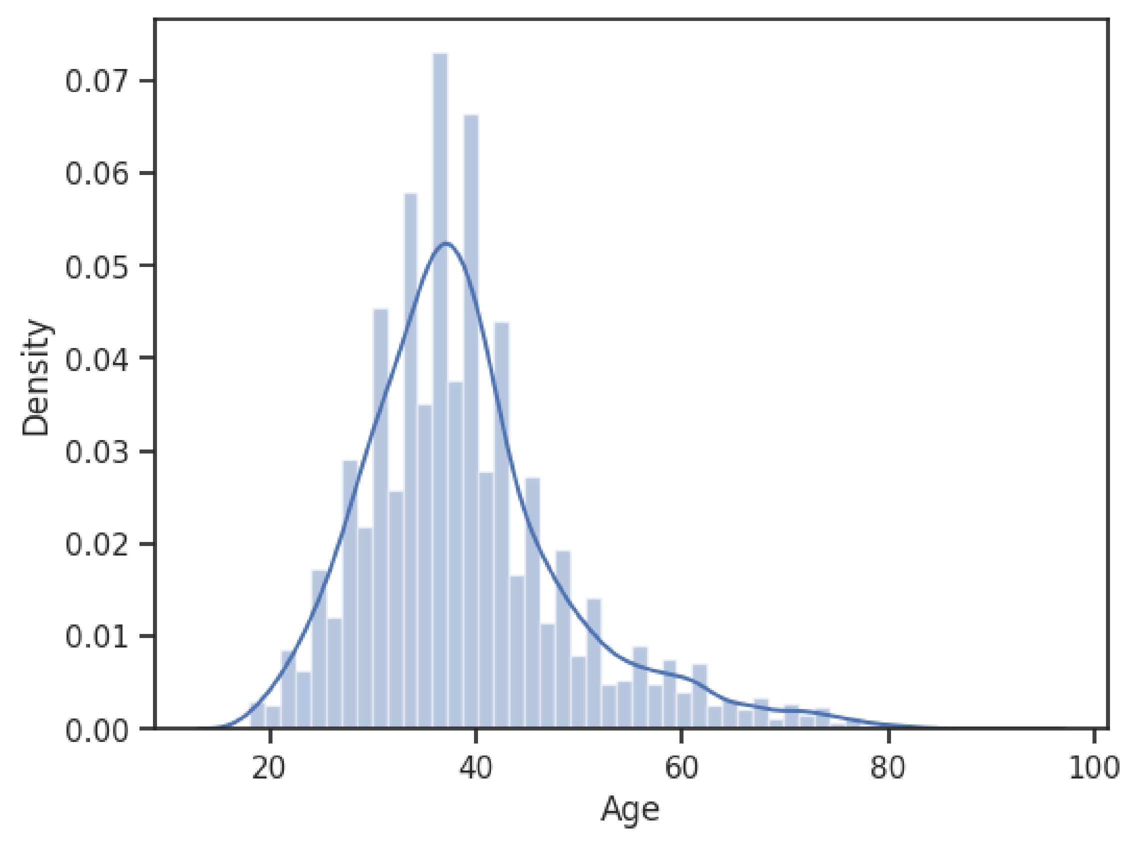
```
[32] # Filling nulls in Age by mean value (numeric column)

     #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.isnan(x) else x)
```

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

# AGE FEATURE BEFORE AND AFTER DATA CLEANING




BEFORE DATA CLEANING




AFTER DATA CLEANING

# NULL VALUES && DATA NORMALIZATION

[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)
```

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

```
France      0.348485
UNK         0.300000
Spain       0.177273
Germany     0.174242
Name: Geography, dtype: float64
```

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

```
Surname            0.0
CreditScore        0.0
Geography          0.0
Gender             0.0
Age                0.0
Tenure             0.0
Balance            0.0
NumOfProducts      0.0
```

# FEATURE ENGINEERING

```
[82]  df_train.columns

      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

```
[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
```

# FEATURE ENGINEERING

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

| | CreditScore | Gender | Age | Tenure | Balance | NumOfProducts | HasCrCard | IsActiveMember | EstimatedSalary | Exited | country_France | country_Germar |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 678 | 1 | 36 | 1 | 117864.85 | 2 | 1 | 0 | 27619.06 | 0 | 0.0 | 1 |
| 1 | 613 | 0 | 27 | 5 | 125167.74 | 1 | 1 | 0 | 199104.52 | 0 | 1.0 | 0 |
| 2 | 628 | 1 | 45 | 9 | 0.00 | 2 | 1 | 1 | 96862.56 | 0 | 1.0 | 0 |
| 3 | 513 | 1 | 30 | 5 | 0.00 | 2 | 1 | 0 | 162523.66 | 0 | 1.0 | 0 |
| 4 | 639 | 1 | 22 | 4 | 0.00 | 2 | 1 | 0 | 28188.96 | 0 | 1.0 | 0 |

```
[85] new_cols = ['bal_per_product','bal_by_est_salary','tenure_age_ratio','age_surname_mean_churn']
```

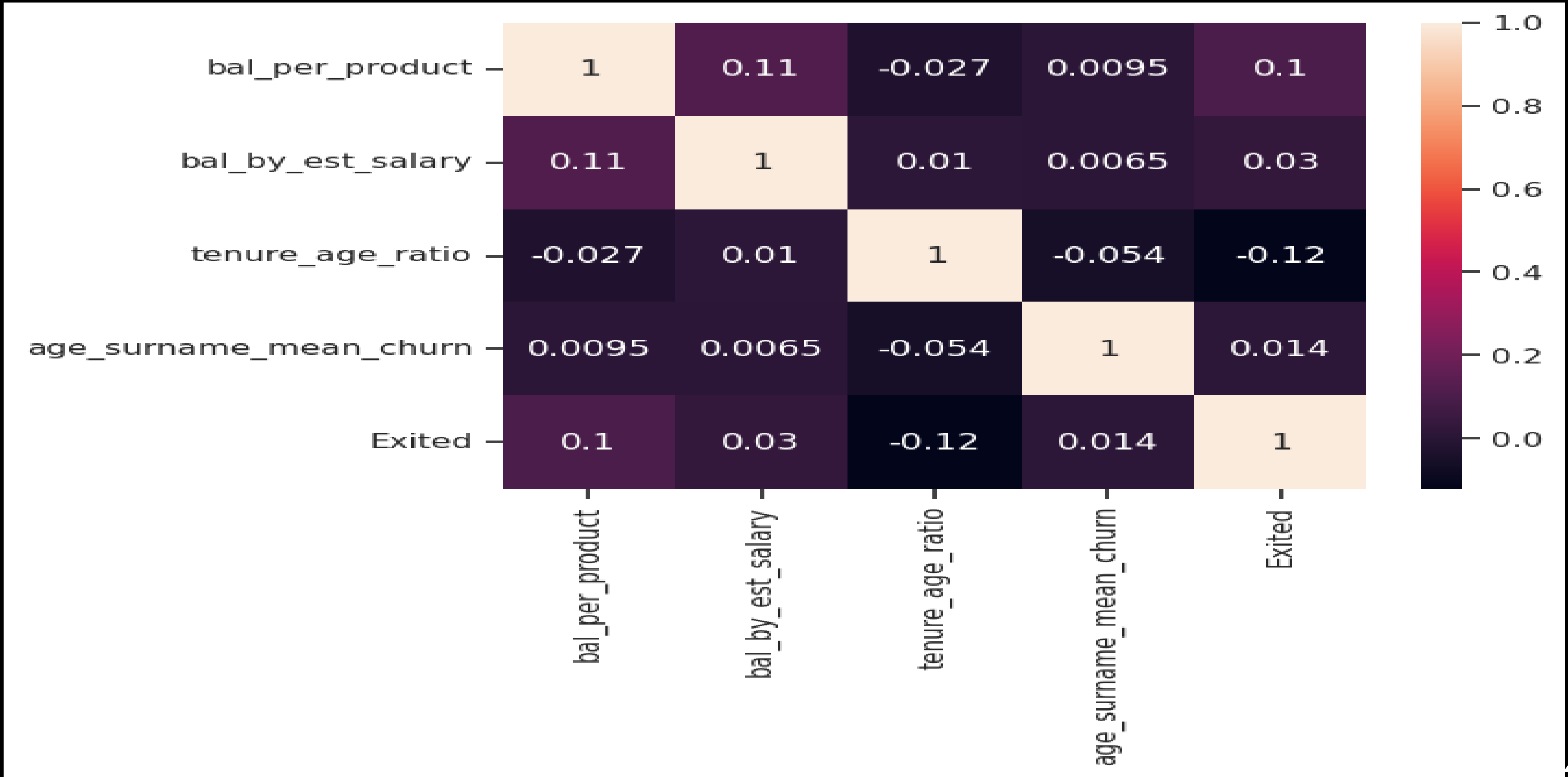
# FEATURE ENGINEERING

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

```
bal_per_product          0
bal_by_est_salary        0
tenure_age_ratio         0
age_surname_mean_churn   0
dtype: int64
```

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

# FEATURE ENGINEERING (HEATMAP)



# FEATURES FOR TESTING AND TRAINING

```
### 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
```

```
### 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
```

# FEATURE SCALING AND NORMALIZATION

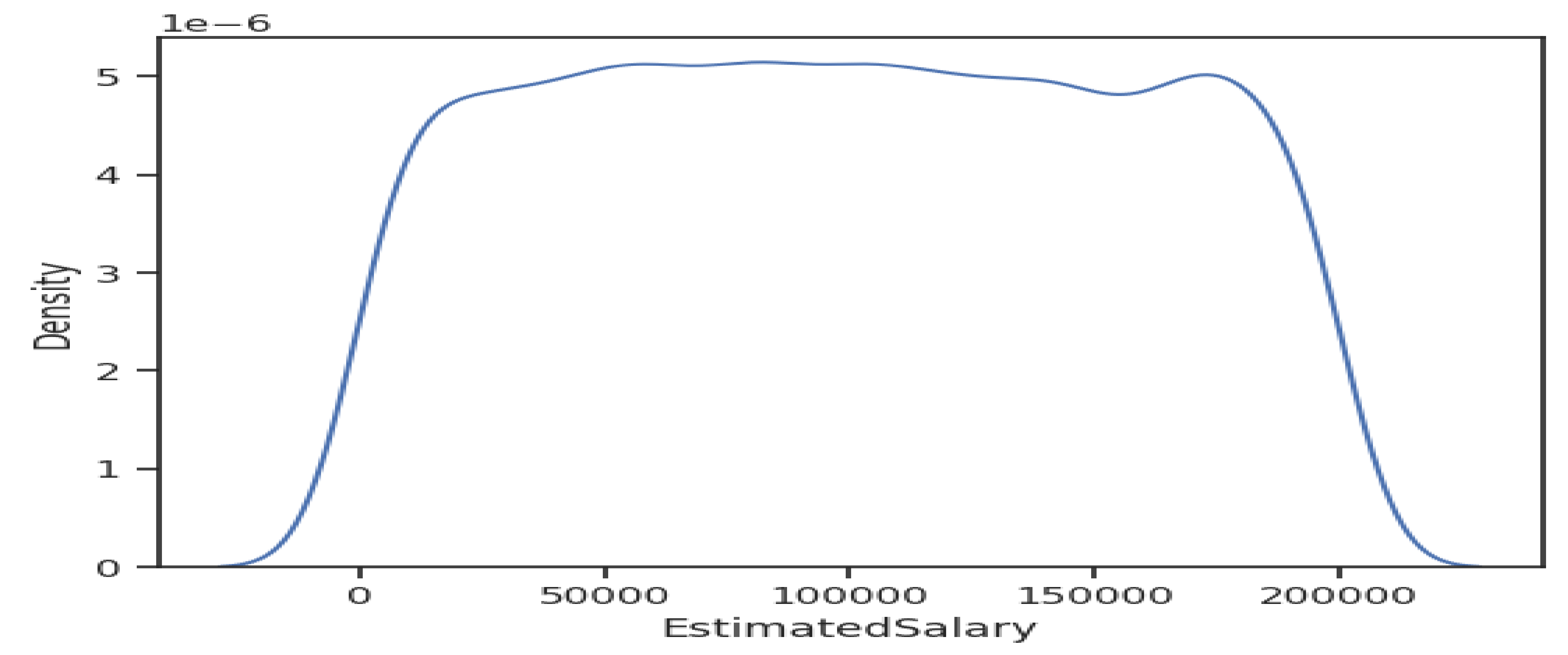
- Different methods :

- 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?

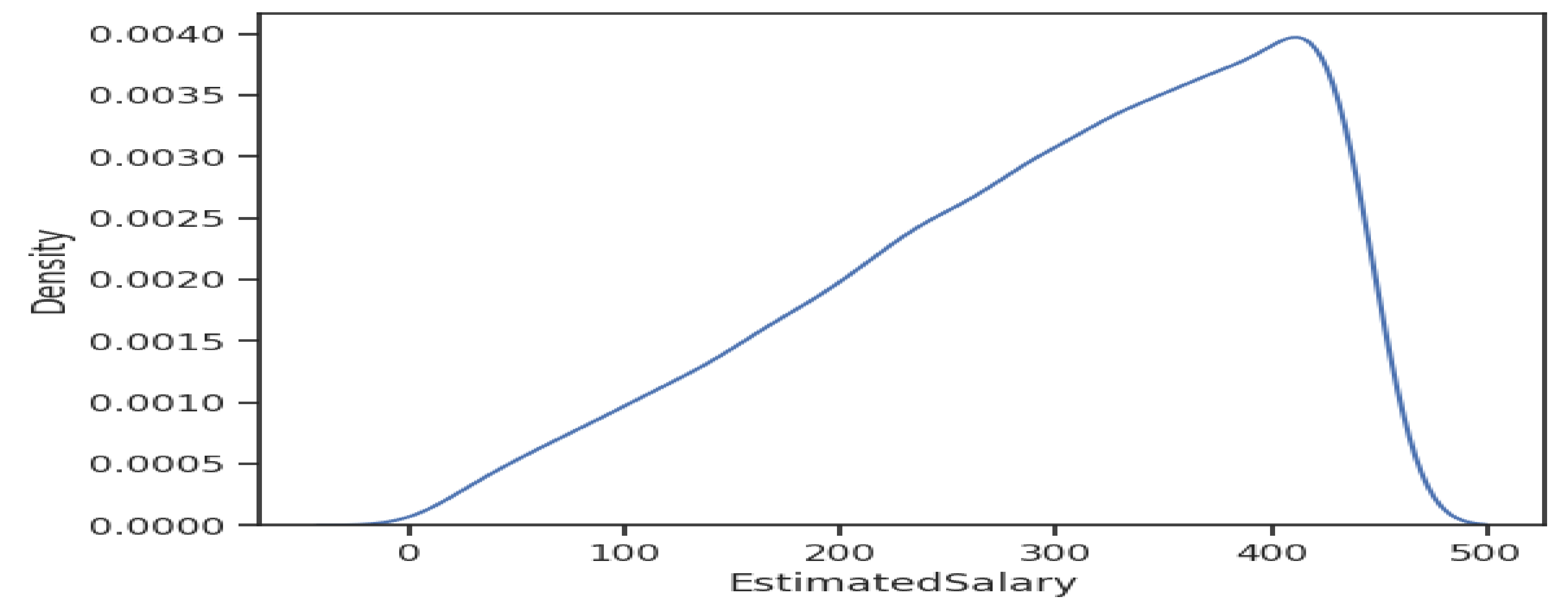
# FEATURE TRANSFORMATIONS

Demo-ing feature transformationssns.distplot(df_train.EstimatedSalary, hist=False)



# FEATURE TRANSFORMATIONS

- sns.distplot(np.sqrt(df_train.EstimatedSalary), hist=False)#sns.distplot(np.log10(1+df_train.EstimatedSalary), hist=False)



# FEATURE SCALING: STANDARDSCALER

```
[92] from sklearn.preprocessing import StandardScaler
     sc = StandardScaler()
```

```
[93] df_train.columns
```

```
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

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

```
[95] ### Scaling only continuous columns
     cols_to_scale = cont_vars
```

# FEATURE SCALING: STANDARDSCALER

[97]
```
## 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()
```

(7920, 11)

| | CreditScore | Age | Tenure | Balance | NumOfProducts | EstimatedSalary | Surname_enc | bal_per_product | bal_by_est_salary | tenure_age_ratio |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0.284761 | -0.274383 | -1.389130 | 0.670778 | 0.804059 | -1.254732 | -1.079210 | -0.062389 | 0.095448 | -1.232035 |
| 1 | -0.389351 | -1.128482 | -0.004763 | 0.787860 | -0.912423 | 1.731950 | -1.079210 | 1.104840 | -0.118834 | 0.525547 |
| 2 | -0.233786 | 0.579716 | 1.379604 | -1.218873 | 0.804059 | -0.048751 | 0.094549 | -1.100925 | -0.155854 | 0.690966 |
| 3 | -1.426446 | -0.843782 | -0.004763 | -1.218873 | 0.804059 | 1.094838 | 0.505364 | -1.100925 | -0.155854 | 0.318773 |
| 4 | -0.119706 | -1.602981 | -0.350855 | -1.218873 | 0.804059 | -1.244806 | 1.561746 | -1.100925 | -0.155854 | 0.487952 |

[98]
```
## Mapping learnt on the continuous features
sc_map = {'mean':sc.mean_, 'std':np.sqrt(sc.var_)}
sc_map
```

# RFE MODEL FOR LOGISTIC REGRESSION AND DECISION TREE

```
02] ### 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
```

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

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

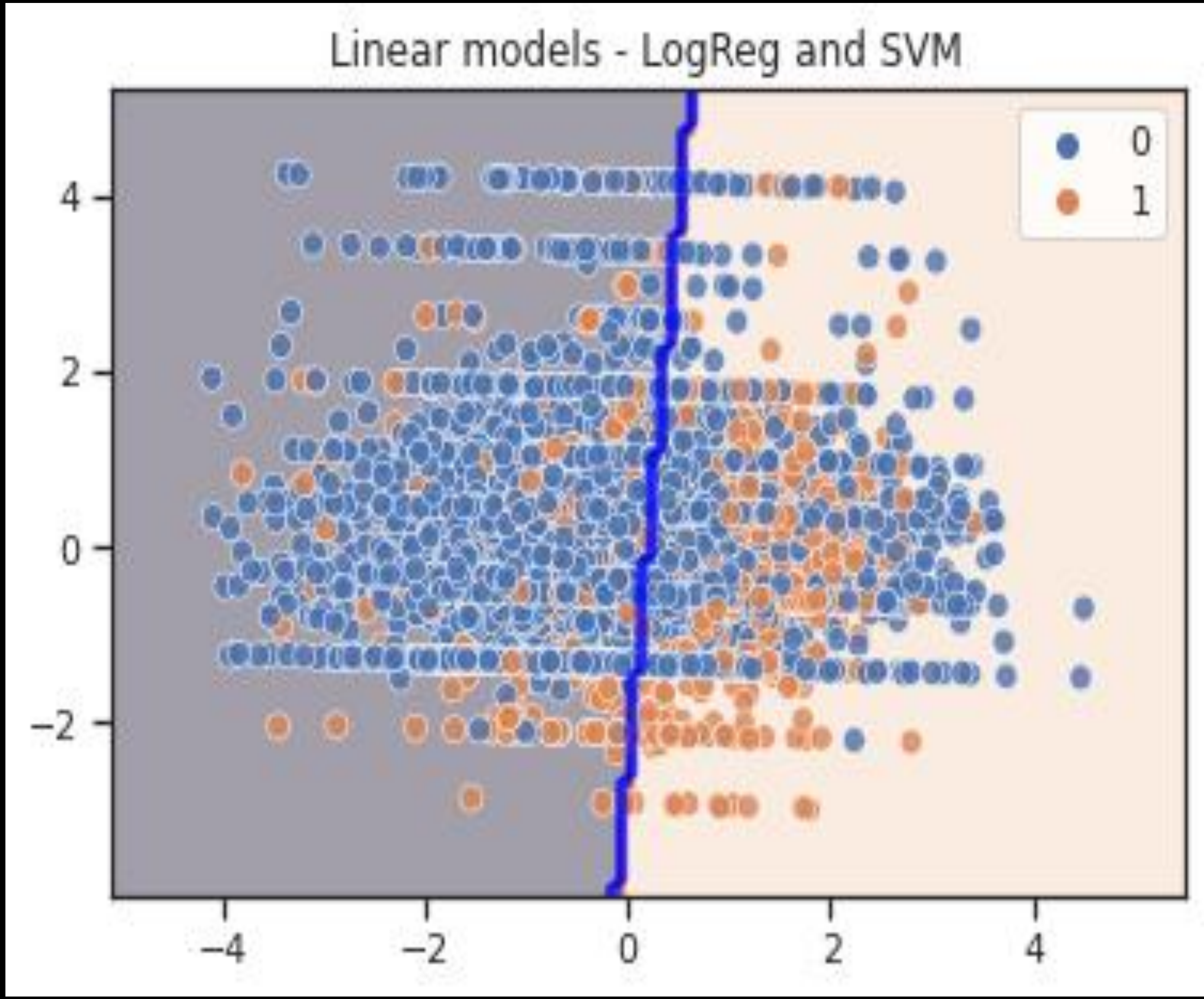
```
05] # for decision trees
    est_dt = DecisionTreeClassifier(max_depth = 4, criterion = 'entropy')
    num_features_to_select = 10
```

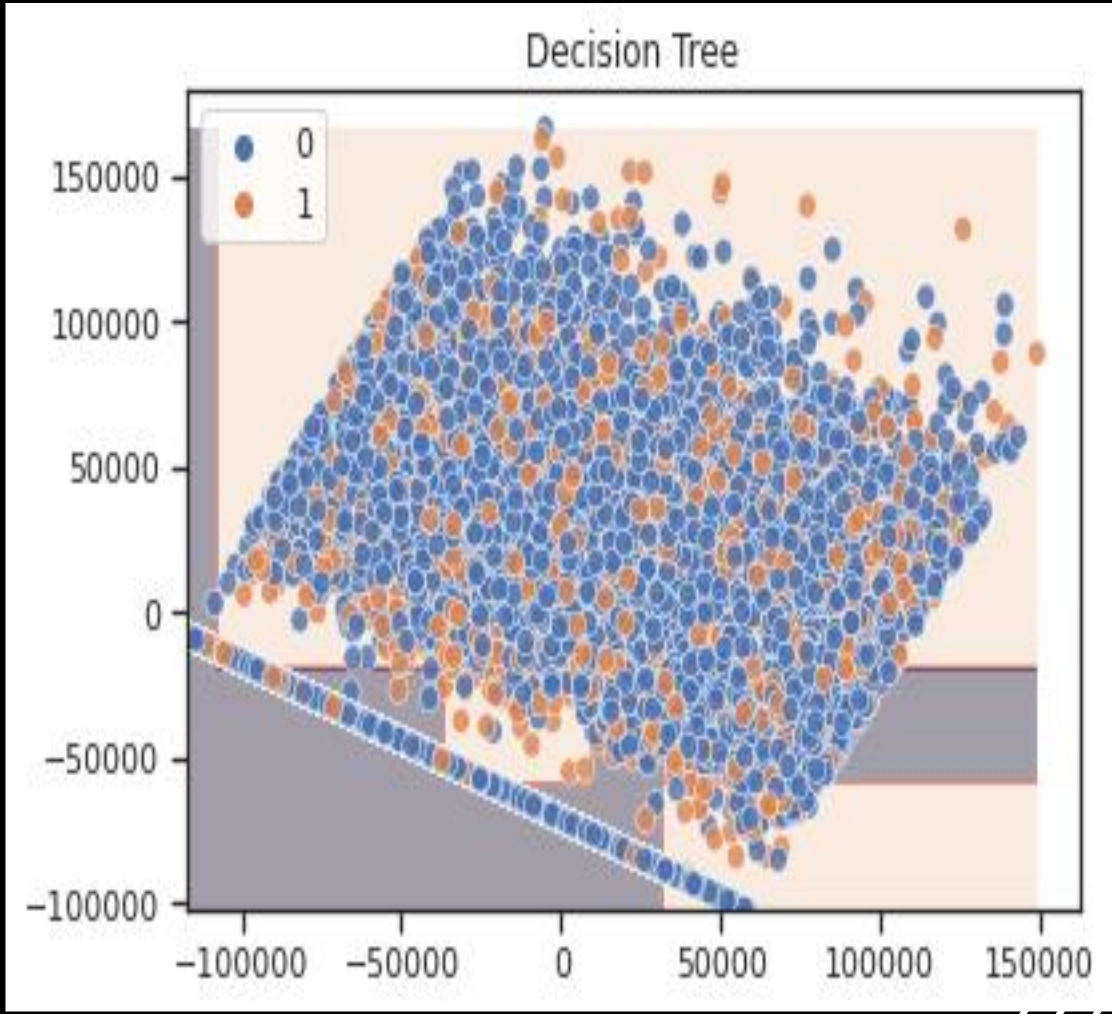
```
06] # for logistics regression
    rfe = RFE(est, num_features_to_select)
    rfe = rfe.fit(X.values, y)
    print(rfe.support_)
```

# PRINCIPAL COMPONENT ANALYSIS(PCA)

```python
from sklearn.decomposition import PCA

pca = PCA(n_components=2)

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

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

# 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))

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

# LOG REG V/S SVM V/S DECISION TREE ANALYSIS



# PIPELINING OF A SINGLE MODEL

Pipeline in action for a single model

```
163] 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, classification_report
```

```
164] 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', 'bal_by_est_salary', 'tenure_age_ratio'
                     ,'age_surname_enc']
```

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

     clf = DecisionTreeClassifier(criterion = 'entropy', class_weight = weights_dict, max_depth = 4, max_features = None
                                 , min_samples_split = 25, min_samples_leaf = 15)
```

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

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

# EVALUATING RECALL AND F1-SCORE METRICS USING KFOLD AND ZOO MODEL

```
### 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 = 0.7493527602020085, std_dev = 0.026176914665796896
Model lgb_21: mean = 0.7866856291480427, std_dev = 0.015745566437193475
Model xgb_21: mean = 0.7506085408564075, std_dev = 0.010966112801395578
Model et_21: mean = 0.7381861806079604, std_dev = 0.009033556110987941
Model rf_1001: mean = 0.7474932760588998, std_dev = 0.024780276266803267
Model lgb_1001: mean = 0.6884232116251622, std_dev = 0.014573973874519829
Model xgb_1001: mean = 0.6753719935759757, std_dev = 0.017567029997772903
Model et_1001: mean = 0.7363150867823766, std_dev = 0.0054959309820837516
Model knn_3: mean = 0.32214933921557243, std_dev = 0.021051639994704833
Model knn_5: mean = 0.2879356049612043, std_dev = 0.006396680440459953
Model knn_11: mean = 0.23568622898163735, std_dev = 0.023099705052575383
Model gauss_nb: mean = 0.0360906329211896, std_dev = 0.01511162576177723
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 = 0.6286545216621772, std_dev = 0.01880933233764158
Model lgb_21: mean = 0.6445713376921776, std_dev = 0.010347896896123705
Model xgb_21: mean = 0.6130509823329311, std_dev = 0.00848890204896738
Model et_21: mean = 0.590474996756568, std_dev = 0.0074631497300233106
Model rf_1001: mean = 0.6284716341377018, std_dev = 0.014863357989071506
Model lgb_1001: mean = 0.677231392541388, std_dev = 0.009841732603586511
Model xgb_1001: mean = 0.683463280904695, std_dev = 0.014982910608582397
Model et_1001: mean = 0.5911873424742697, std_dev = 0.00805199861616842
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
```

# HYPERPARAMTER 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

# HYPERPARAMTER TUNING

```
[177] from sklearn.pipeline import Pipeline
      from sklearn.model_selection import GridSearchCV, RandomizedSearchCV
      from lightgbm import LGBMClassifier
```

```
[178] ## 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
```

```
      ((7920, 17), (7920,))((1080, 17), (1080,))
```

```
[179] lgb = LGBMClassifier(boosting_type = 'dart', min_child_samples = 20, n_jobs = - 1, importance_type = 'gain', num_leaves = 31)
```

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

# HYPERPARAMETER TUNING(GRID SEARCH RESULTS)

```
] grid.cv_results_

{'mean_fit_time': array([2.94769998, 1.95596223, 2.32641468, 2.53946695, 1.93624334,
        2.92211356, 1.80242229, 1.79475646, 2.75336413, 2.03374538,
        2.4281765 , 2.65032048, 2.72008262, 2.95253925, 2.04519801,
        2.56882243, 2.26601057, 1.92016759]),
 'std_fit_time': array([0.52887561, 0.03001612, 0.60437116, 0.67490923, 0.01183468,
        0.52840775, 0.03235945, 0.02422623, 0.49726065, 0.01284962,
        0.57697631, 0.38198122, 0.5527754 , 0.65604198, 0.01946636,
        0.7569963 , 0.38329618, 0.00856377]),
 'mean_score_time': array([0.05595851, 0.0365747 , 0.05523801, 0.04520683, 0.0363625 ,
        0.04027677, 0.03718095, 0.03641295, 0.06008372, 0.03825874,
        0.04899487, 0.04473743, 0.05487528, 0.06271949, 0.037467  ,
        0.06676121, 0.03748217, 0.03675299]),
 'std_score_time': array([0.02320697, 0.00025422, 0.01838449, 0.01762724, 0.00023008,
        0.00310354, 0.00135917, 0.00081168, 0.01884039, 0.00210534,
        0.01485137, 0.00959684, 0.02109968, 0.02173708, 0.00192224,
        0.01825172, 0.00085793, 0.00242326]),
 '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),
```

# ENSEMBLES(MULTIMODEL PIPELINING)

```
] ## 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)
                             ])
```

```
] ## 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())
```

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

# ENSEMBLES(MULTIMODEL PIPELINING PREDICTIONS)

```
] ## 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)
```

```
] ## 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], 'm3_pred': m3_pred_probs_trn[:,1]})
  df_t.shape
  df_t.corr()
```

(7920, 3)

| | 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 |

# ROC_AUC, F1-SCORE, CONFUSION MATRIX AND CLASSFICATION REPORT OF ENSEMBLE MULTIMODEL PIPELINE

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

```
## 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)
```

```
threshold = 0.5
```

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

```
## Model averaging predictions (Weighted average)
m1_m2_preds = np.where(((0.1*m1_pred_probs_val[:,1]) + (0.9*m2_pred_probs_val[:,1])) >= threshold, 1, 0)
```

```
## 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))
```

```
0.74693107646859220.5924369747899160array([[759,  83],
       [ 97, 141]])                  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
```

# LIST OF CUSTOMER WHO ARE MOST LIKELY TO CHURN

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

```
[265] high_churn_list = test[test.pred_probabilities > 0.7].sort_values(by = ['pred_probabilities'], ascending = False
                                                                         ).reset_index().drop(columns = ['index', 'Exited', 'predictions'], axis = 1)
```

```
[266] high_churn_list.shape
      high_churn_list.head()
```

(103, 18)

| | CreditScore | Gender | Age | Tenure | Balance | NumOfProducts | HasCrCard | IsActiveMember | EstimatedSalary | country_France | country_Germany | country_Spain | Surname_enc |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 546 | 0 | 58 | 3 | 106458.31 | 4 | 1 | 0 | 128881.87 | 0.0 | 1.0 | 0.0 | 0.000000 |
| 1 | 479 | 1 | 51 | 1 | 107714.74 | 3 | 1 | 0 | 86128.21 | 0.0 | 1.0 | 0.0 | 0.333333 |
| 2 | 745 | 1 | 45 | 10 | 117231.63 | 3 | 1 | 1 | 122381.02 | 0.0 | 1.0 | 0.0 | 0.250000 |
| 3 | 515 | 1 | 45 | 7 | 120961.50 | 3 | 1 | 1 | 39288.11 | 0.0 | 1.0 | 0.0 | 0.200000 |
| 4 | 481 | 0 | 57 | 9 | 0.00 | 3 | 1 | 1 | 169719.35 | 1.0 | 0.0 | 0.0 | 0.222222 |

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

# CONCLUSION AND FUTURE SCOPE

- CONCLUSION:
- Different Ensemble techniques were used to analyze and determined whether and how much the
- Accurate the customer prediction churn was being performed and in future we can also use this data
- To analyze loan based prediction eligibility for future customer.

- FUTURE SCOPE:
- This will be used to link with analysis of Debit Card Fraud Analysis, Fraud Online Transaction
- It also has application in field of Risk Management, Market Analysis

## Problem statement :

Bank 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 pro
    %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!

```
# !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==1.20.1
# !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
```

```
#!pip install scikit_learn==0.24.1
```

```
%matplotlib inline


## Import required libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns


## 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)


## 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)


## Reading the dataset
# This might be present in S3, or obtained through a query on a database
df = pd.read_csv("https://s3.amazonaws.com/hackerday.datascience/360/Churn_Modelling.cs
```

```
df.shape
```

```
(10000, 14)
```

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

|  | 0 | 1 | 2 | 3 | 4 | 5 |  |
|---|---|---|---|---|---|---|---|
| **RowNumber** | 1 | 2 | 3 | 4 | 5 | 6 |  |
| **CustomerId** | 15634602 | 15647311 | 15619304 | 15701354 | 15737888 | 15574012 | 1559 |
| **Surname** | Hargrave | Hill | Onio | Boni | Mitchell | Chu | B |

## Basic EDA

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

|  | RowNumber | CustomerId | CreditScore | Age | Tenure | Bal |
|---|---|---|---|---|---|---|
| **count** | 10000.00000 | 1.000000e+04 | 10000.000000 | 10000.000000 | 10000.000000 | 10000.00 |
| **mean** | 5000.50000 | 1.569094e+07 | 650.528800 | 38.921800 | 5.012800 | 76485.88 |
| **std** | 2886.89568 | 7.193619e+04 | 96.653299 | 10.487806 | 2.892174 | 62397.40 |
| **min** | 1.00000 | 1.556570e+07 | 350.000000 | 18.000000 | 0.000000 | 0.00 |
| **25%** | 2500.75000 | 1.562853e+07 | 584.000000 | 32.000000 | 3.000000 | 0.00 |
| **50%** | 5000.50000 | 1.569074e+07 | 652.000000 | 37.000000 | 5.000000 | 97198.54 |
| **75%** | 7500.25000 | 1.575323e+07 | 718.000000 | 44.000000 | 7.000000 | 127644.24 |
| **max** | 10000.00000 | 1.581569e+07 | 850.000000 | 92.000000 | 10.000000 | 250898.09 |

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

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

```
(10000, 10000)
```

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

```
df_t.head()
```

| | Surname | RowNumber | Exited |
|---|---|---|---|
| **2473** | Smith | 32 | 0.281250 |
| **1689** | Martin | 29 | 0.310345 |
| **2389** | Scott | 29 | 0.103448 |

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

```
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

```
## 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', 'EstimatedSala
cat_feats = ['Surname', 'Geography', 'Gender', 'HasCrCard', 'IsActiveMember']
```

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

```
## 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

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

```
## 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_siz
```

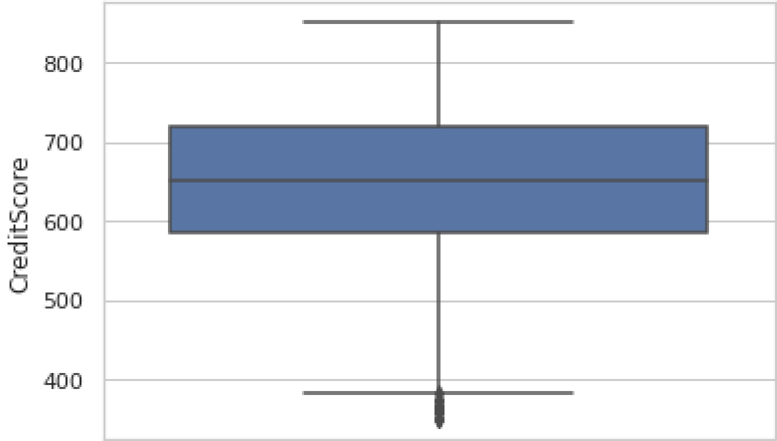
```
df_train.shape, df_val.shape, df_test.shape, y_train.shape, y_val.shape, y_test.shape
np.mean(y_train), np.mean(y_val), np.mean(y_test)
```

```
((7920, 12), (1080, 12), (1000, 12), (7920,), (1080,), (1000,))
(0.20303030303030303, 0.22037037037037038, 0.191)
```

## Univariate plots of numerical variables in training set

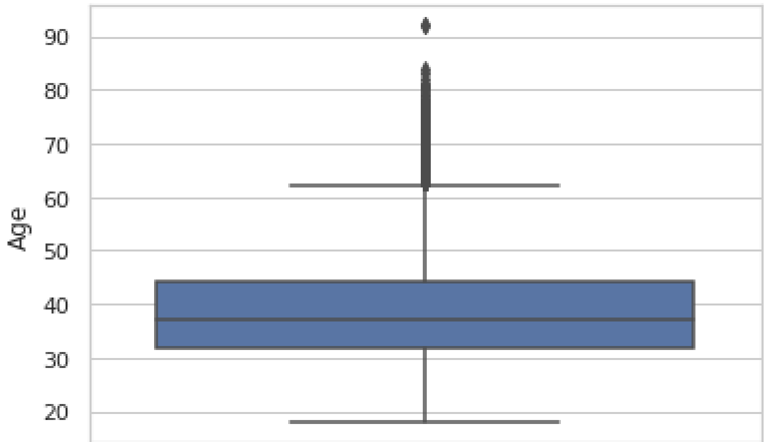
```
## CreditScore
sns.set(style="whitegrid")
sns.boxplot(y = df_train['CreditScore'])
```

<AxesSubplot:ylabel='CreditScore'>



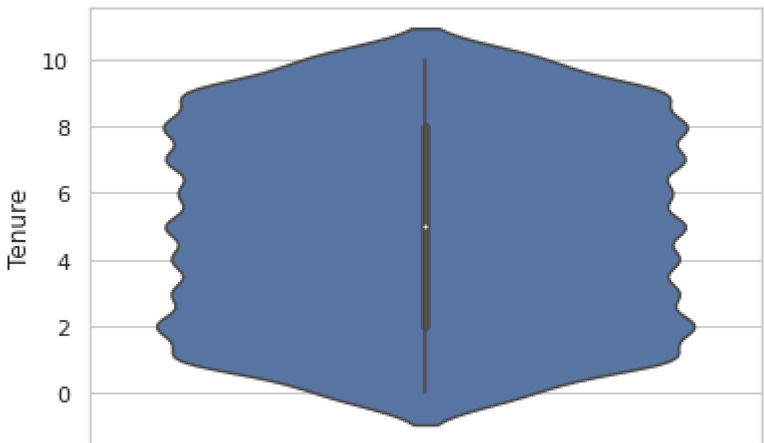

```
## Age
sns.boxplot(y = df_train['Age'])
```

<AxesSubplot:ylabel='Age'>



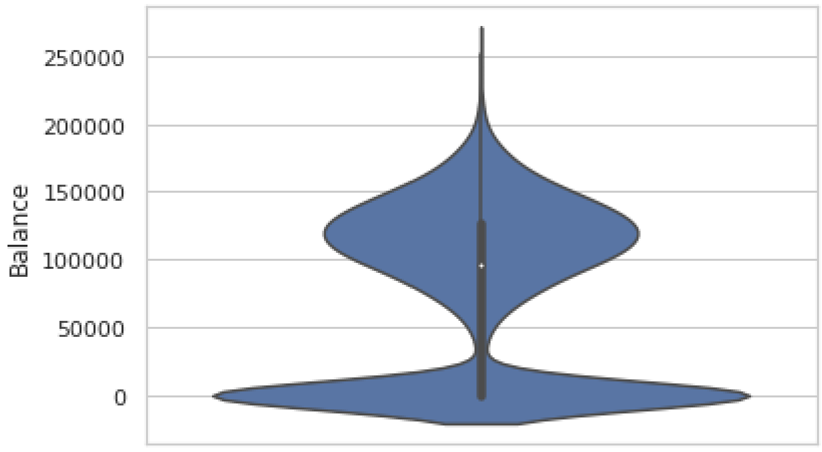

```
## Tenure
sns.violinplot(y = df_train.Tenure)
```

<AxesSubplot:ylabel='Tenure'>



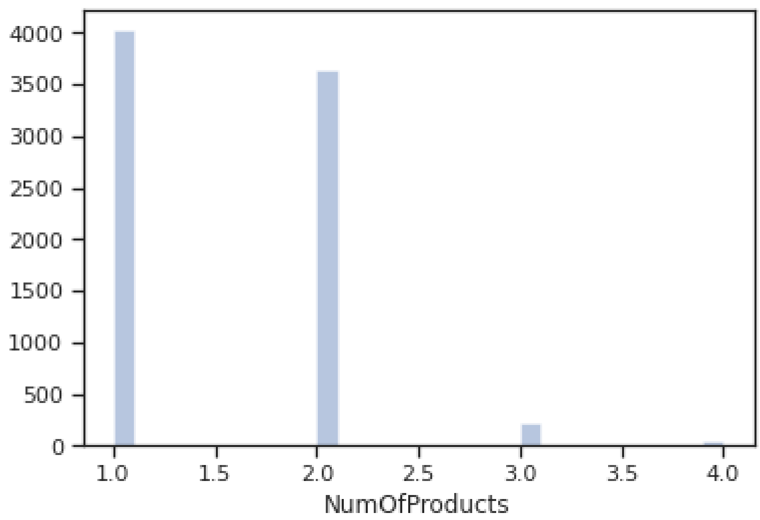
```
## Balance
sns.violinplot(y = df_train['Balance'])
```

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




```
## NumOfProducts
sns.set(style = 'ticks')
sns.distplot(df_train.NumOfProducts, hist=True, kde=False)
```

```
<AxesSubplot:xlabel='NumOfProducts'>
```




```
## EstimatedSalary
sns.kdeplot(df_train.EstimatedSalary)
```

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


- 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

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

```
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.

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


## 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


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

```
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
```

```
## Calculating mean statistics
age_mean = df_missing.Age.mean()
```

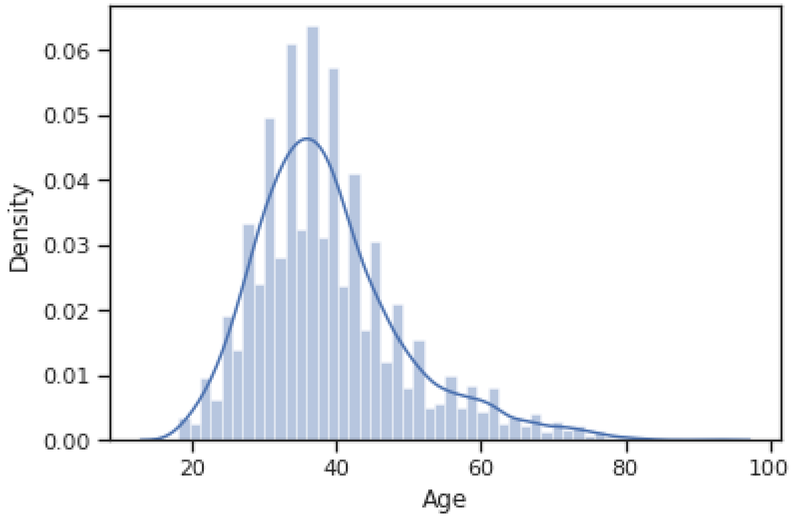
```
age_mean
```

```
38.91442199775533
```

```
# 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
```

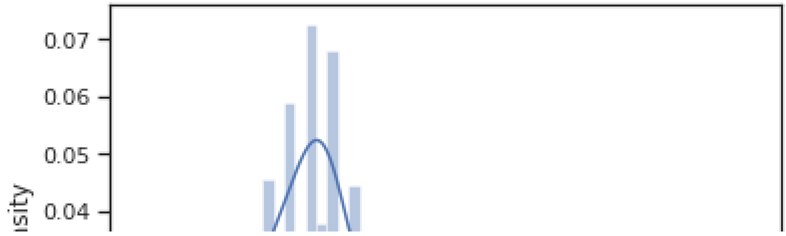
```
## Distribution of "Age" feature before data imputation
sns.distplot(df_train.Age)
```

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




```
## Distribution of "Age" feature after data imputation
sns.distplot(df_missing.Age)
```

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




```
# 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)
```

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

```
France     0.345202
UNK        0.300000
Spain      0.178662
Germany    0.176136
Name: Geography, dtype: float64
```

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

```
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

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

```
df_train.sample(10)
```

| | Surname | CreditScore | Geography | Gender | Age | Tenure | Balance | NumOfProdu |
|---|---|---|---|---|---|---|---|---|
| **5224** | Fleetwood-Smith | 803 | Spain | Male | 43 | 3 | 0.00 | |
| **5925** | Biryukov | 706 | Germany | Female | 39 | 8 | 112889.91 | |
| **4783** | Jennings | 710 | France | Female | 37 | 5 | 0.00 | |
| **2108** | Hay | 593 | Germany | Male | 74 | 5 | 161434.36 | |
| **5757** | T'ang | 681 | France | Male | 32 | 3 | 148884.47 | |
| **8776** | Griffin | 567 | Spain | Male | 44 | 9 | 0.00 | |
| **3849** | Robinson | 646 | Spain | Male | 32 | 1 | 0.00 | |
| **9662** | Gallo | 748 | Spain | Male | 39 | 3 | 0.00 | |
| **5455** | Oliver | 805 | Germany | Female | 45 | 9 | 116585.97 | |
| **2643** | Ni | 632 | France | Male | 27 | 4 | 193125.85 | |

```
df_train.drop('Gender_cat', axis=1, inplace = True)
```

```
## The sklearn method
from sklearn.preprocessing import LabelEncoder
```

```
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.

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

```
le_name_mapping = dict(zip(le.classes_, le.transform(le.classes_)))
le_name_mapping
```

```
{'Female': 0, 'Male': 1}
```

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

```
array([1])
```

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

```
0    NaN
dtype: float64
```

```
## 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)
```

```
df_train.Gender.unique(), df_val.Gender.unique(), df_test.Gender.unique()
```

```
(array([1, 0]), array([1, 0]), array([1, 0]))
```

## One-Hot encoding for categorical variables with multiple levels

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

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

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

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

```
## The sklearn method
from sklearn.preprocessing import LabelEncoder, OneHotEncoder


le_ohe = LabelEncoder()
ohe = OneHotEncoder(handle_unknown = 'ignore', sparse=False)


enc_train = le_ohe.fit_transform(df_train.Geography).reshape(df_train.shape[0],1)
enc_train.shape
np.unique(enc_train)
```

```
(7920, 1)array([0, 1, 2])
```

```
ohe_train = ohe.fit_transform(enc_train)
ohe_train
```

```
array([[0., 1., 0.],
       [1., 0., 0.],
       [1., 0., 0.],
       ...,
       [1., 0., 0.],
       [0., 1., 0.],
       [0., 1., 0.]])
```

```
le_ohe_name_mapping = dict(zip(le_ohe.classes_, le_ohe.transform(le_ohe.classes_)))
le_ohe_name_mapping
```

```
{'France': 0, 'Germany': 1, 'Spain': 2}
```

```
## 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


np.unique(enc_val)
np.unique(enc_test)
```

```
array([0, 1, 2])array([0, 1, 2])
```

```
ohe_val = ohe.transform(enc_val)
ohe_test = ohe.transform(enc_test)
```

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

```
array([[0., 0., 0.]])
```

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

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

```
['country_France', 'country_Germany', 'country_Spain']
```

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

```
print("Training set")
df_train.head()
print("\n\nValidation set")
df_val.head()
print("\n\nTest set")
df_test.head()
```

Training set

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

Validation set

| | Surname | CreditScore | Geography | Gender | Age | Tenure | Balance | NumOfProducts |
|---|---|---|---|---|---|---|---|---|
| **0** | Sun | 757 | France | 1 | 36 | 7 | 144852.06 | 1 |

```
## 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)
```

## Target encoding

Test set

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

```
df_train.head()
```

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

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

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

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

```
0.20303030303030303
```

```
## 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

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

```
Surname
Abazu       2
Abbie       1
Abbott      4
Abdullah    1
Abdulov     1
dtype: int64
```

```
## 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)


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

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

```
## 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) /
df_train.head(10)
```

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

```
## 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)
```

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **7** | Edments | 705 | 1 | 33 | 7 | 68423.89 | 1 | 1 |

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

| | 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 |

```
### 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)


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

| | CreditScore | Gender | Age | Tenure | Balance | NumOfProducts | HasCrCard | IsActiveM |
|---|---|---|---|---|---|---|---|---|
| **0** | 678 | 1 | 36 | 1 | 117864.85 | 2 | 1 | |
| **1** | 613 | 0 | 27 | 5 | 125167.74 | 1 | 1 | |
| **2** | 628 | 1 | 45 | 9 | 0.00 | 2 | 1 | |
| **3** | 513 | 1 | 30 | 5 | 0.00 | 2 | 1 | |
| **4** | 639 | 1 | 22 | 4 | 0.00 | 2 | 1 | |

| | CreditScore | Gender | Age | Tenure | Balance | NumOfProducts | HasCrCard | IsActiveM |
|---|---|---|---|---|---|---|---|---|
| **0** | 757 | 1 | 36 | 7 | 144852.06 | 1 | 0 | |
| **1** | 552 | 1 | 29 | 10 | 0.00 | 2 | 1 | |
| **2** | 619 | 0 | 30 | 7 | 70729.17 | 1 | 1 | |
| **3** | 633 | 1 | 35 | 10 | 0.00 | 2 | 1 | |

*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

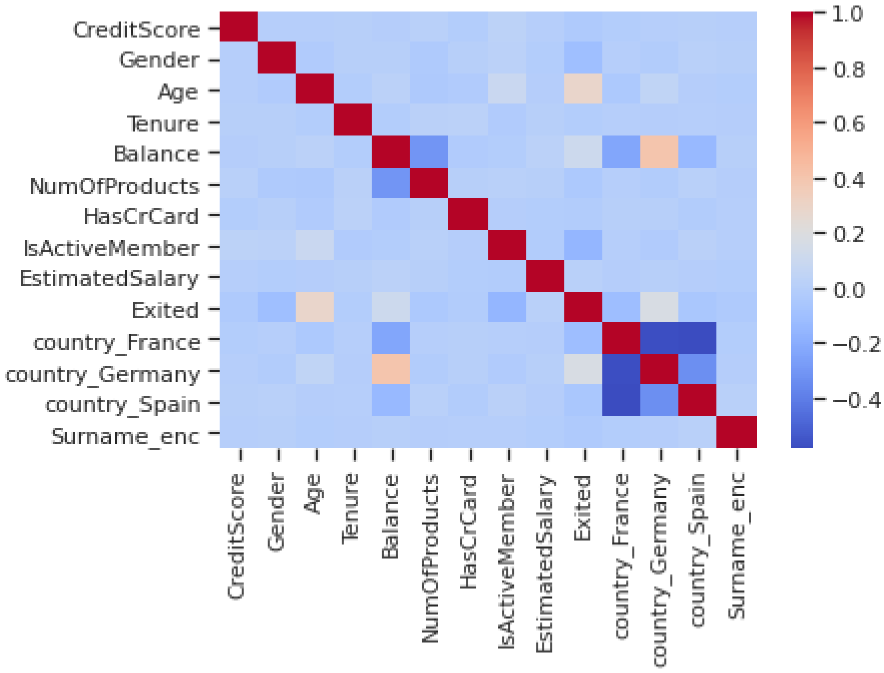
## Bivariate analysis

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

| | CreditScore | Gender | Age | Tenure | Balance | NumOfProduc |
|---|---|---|---|---|---|---|
| **CreditScore** | 1.000000 | 0.000354 | 0.002099 | 0.005994 | -0.001507 | 0.0141 |
| **Gender** | 0.000354 | 1.000000 | -0.024446 | 0.010749 | 0.009380 | -0.0267 |
| **Age** | 0.002099 | -0.024446 | 1.000000 | -0.011384 | 0.027721 | -0.0333 |
| **Tenure** | 0.005994 | 0.010749 | -0.011384 | 1.000000 | -0.013081 | 0.0182 |
| **Balance** | -0.001507 | 0.009380 | 0.027721 | -0.013081 | 1.000000 | -0.3043 |
| **NumOfProducts** | 0.014110 | -0.026795 | -0.033305 | 0.018231 | -0.304318 | 1.0000 |
| **HasCrCard** | -0.011868 | 0.007550 | -0.019633 | 0.026148 | -0.021464 | 0.0072 |
| **IsActiveMember** | 0.035057 | 0.028094 | 0.093573 | -0.021263 | -0.008085 | 0.0148 |
| **EstimatedSalary** | 0.000358 | -0.011007 | -0.006827 | 0.010145 | 0.027247 | 0.0097 |
| **Exited** | -0.028117 | -0.102331 | 0.288221 | -0.010660 | 0.113377 | -0.0392 |
| **country_France** | -0.009481 | 0.000823 | -0.038881 | 0.000021 | -0.231770 | 0.0029 |

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

<AxesSubplot:>



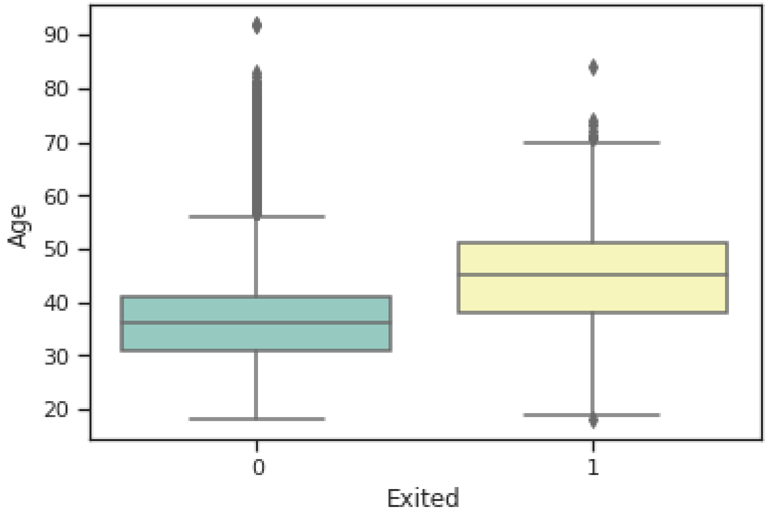

None of the features are highly correlated with the target variable. But some of them have slight linear associations with the target variable.

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

## Individual features versus their distibution across target variable values

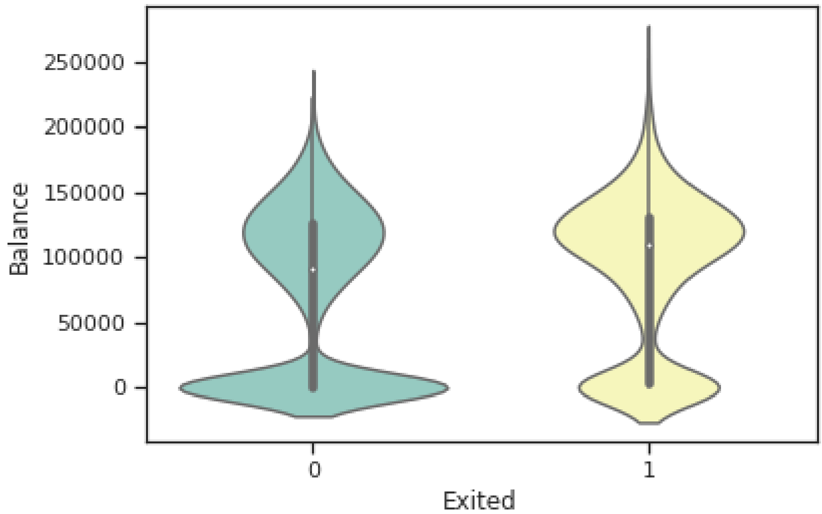
```
sns.boxplot(x = "Exited", y = "Age", data = df_train, palette="Set3")
```

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




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

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




```
# 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()
```

```
Gender
0    0.248191
1    0.165511
Name: Exited, dtype: float64IsActiveMember
0    0.266285
1    0.143557
Name: Exited, dtype: float64country_Germany
0.0    0.163091
1.0    0.324974
Name: Exited, dtype: float64country_France
0.0    0.245877
```

```
1.0    0.160593
Name: Exited, dtype: float64
```

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

```
NumOfProducts
1    0.273428
2    0.076881
3    0.825112
4    1.000000
Name: Exited, dtype: float641    4023
2    3629
3     223
4      45
Name: NumOfProducts, dtype: int64
```

## Some basic feature engineering

```
df_train.columns
```

```
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

```
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
```

```
df_train.head()
```

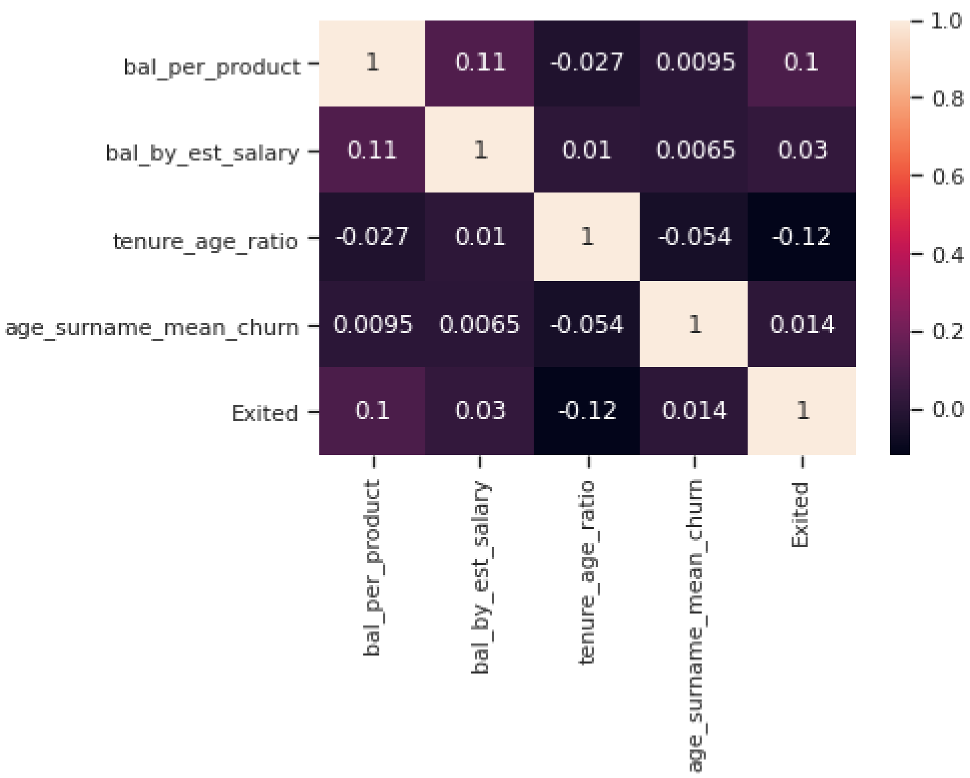
| | CreditScore | Gender | Age | Tenure | Balance | NumOfProducts | HasCrCard | IsActiveM |
|---|---|---|---|---|---|---|---|---|
| **0** | 678 | 1 | 36 | 1 | 117864.85 | 2 | 1 | |
| **1** | 613 | 0 | 27 | 5 | 125167.74 | 1 | 1 | |
| **2** | 628 | 1 | 45 | 9 | 0.00 | 2 | 1 | |
| **3** | 513 | 1 | 30 | 5 | 0.00 | 2 | 1 | |
| **4** | 639 | 1 | 22 | 4 | 0.00 | 2 | 1 | |

```
new_cols = ['bal_per_product','bal_by_est_salary','tenure_age_ratio','age_surname_mean_

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

```
bal_per_product          0
bal_by_est_salary        0
tenure_age_ratio         0
age_surname_mean_churn   0
dtype: int64
```

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

```
<AxesSubplot:>
```




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

```
## 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


## 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
```

## 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?

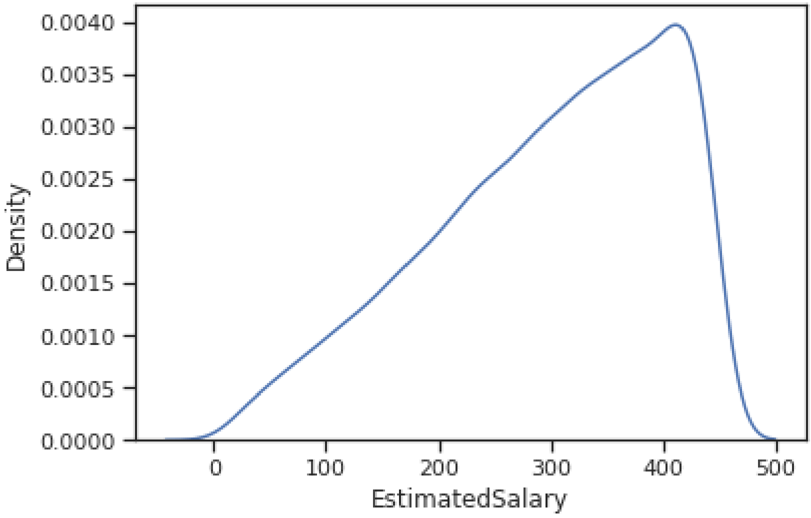
## Feature transformations

```
### Demo-ing feature transformations
sns.distplot(df_train.EstimatedSalary, hist=False)
```

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

```
sns.distplot(np.sqrt(df_train.EstimatedSalary), hist=False)
#sns.distplot(np.log10(1+df_train.EstimatedSalary), hist=False)
```

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




## StandardScaler

```
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
```

```
df_train.columns
```

```
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

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

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

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

```
## 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()
```

```
(7920, 11)
```

| | CreditScore | Age | Tenure | Balance | NumOfProducts | EstimatedSalary | Surr |
|---|---|---|---|---|---|---|---|
| **0** | 0.284761 | -0.274383 | -1.389130 | 0.670778 | 0.804059 | -1.254732 | - |
| **1** | -0.389351 | -1.128482 | -0.004763 | 0.787860 | -0.912423 | 1.731950 | - |
| **2** | -0.233786 | 0.579716 | 1.379604 | -1.218873 | 0.804059 | -0.048751 | |
| **3** | -1.426446 | -0.843782 | -0.004763 | -1.218873 | 0.804059 | 1.094838 | |
| **4** | -0.119706 | -1.602981 | -0.350855 | -1.218873 | 0.804059 | -1.244806 | |

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

```
{'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])}
```

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


## 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

## 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.

```
cont_vars
cat_vars
```

```
     ['CreditScore',
      'Age',
      'Tenure',
      'Balance',
      'NumOfProducts',
      'EstimatedSalary',
      'Surname_enc',
      'bal_per_product',
      'bal_by_est_salary',
      'tenure_age_ratio',
      'age_surname_mean_churn']['Gender',
      'HasCrCard',
      'IsActiveMember',
      'country_France',
      'country_Germany',
      'country_Spain']
```

```
## 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
```

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

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

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

```
# for logistics regression
rfe = RFE(est, num_features_to_select)
```

```
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]
```

```
# for decision trees
rfe_dt = RFE(est_dt, num_features_to_select)
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 False  True  True
  True  True  True  True  True]
[8 7 1 6 1 5 4 1 3 2 1 1 1 1 1 1 1]
```

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

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

```
## 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
```

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

## Baseline model : Logistic Regression

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

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

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

```
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]
```

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

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

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

### Solving class imbalance

```
# 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
```

```
array([1.        , 3.92537313])array([6312, 1608])
```

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

weights_dict
```

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

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

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

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

```
## Fitted model parameters
selected_cat_vars + selected_cont_vars

lr.coef_
lr.intercept_
```

```
['Gender',
 'HasCrCard',
 'IsActiveMember',
 'country_France',
 'country_Germany',
 'country_Spain',
 'Age',
 'NumOfProducts',
 'Surname_enc',
 'tenure_age_ratio']array([[-0.5190172 , -0.06938782, -0.90843476, -0.33748839,
0.58664742,
        -0.24918718,  0.80999582, -0.05061525, -0.0659637 ,
-0.05143544]])array([0.60235927])
```

```
## 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)))
```

```
0.706843633543310.6983830845771144array([[4515, 1797],
       [ 485, 1123]])              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
```

```
## 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)))
```

```
0.70119663067127090.7016806722689075array([[590, 252],
       [ 71, 167]])              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
```

## More linear models - SVM

```
from sklearn.svm import SVC

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


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

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

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

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

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

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

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

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

```
## Fitted model parameters
selected_cat_vars + selected_cont_vars

svm.coef_
svm.intercept_
```

```
['Gender',
 'HasCrCard',
 'IsActiveMember',
 'country_France',
 'country_Germany',
 'country_Spain',
 'Age',
 'NumOfProducts',
 'Surname_enc',
 'tenure_age_ratio']array([[-0.47099449, -0.05292377, -0.73087898, -0.30828289,
0.55377874,
```

```
        -0.24549586,  0.87507442, -0.04736431, -0.0556716 ,
    -0.03867728]])array([0.45499939])
```

```
## 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)))
```

```
    0.71258252463916150.6946517412935324array([[4611, 1701],
           [ 491, 1117]])              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
```

```
## 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)))
```

```
    0.69845705503103850.6890756302521008array([[596, 246],
           [ 74, 164]])              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
```

## 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

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

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

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

```
y = y_train
X_train.shape
X.shape
y.shape
```

```
(7920, 10)(7920, 2)(7920,)
```

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

```
array([0.2602733 , 0.18789887])
```

```
# 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))
```

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

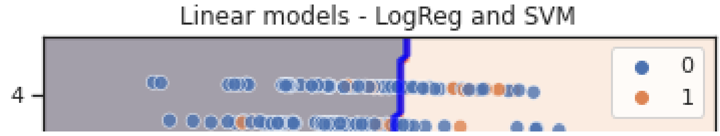
```
LogisticRegression(class_weight={0: 1.0, 1: 3.925373134328358}, n_jobs=-1)
```

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

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

```
## 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')
```

```
<matplotlib.contour.QuadContourSet at 0x7fe3503db700>
<matplotlib.contour.QuadContourSet at 0x7fe3503db730><AxesSubplot:>Text(0.5, 1.0,
'Linear models - LogReg and SVM')
```




## More baseline models (Non-linear) : Decision Tree

```
from sklearn.tree import DecisionTreeClassifier

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

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

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

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

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

```
## Re-defining X_train and X_val to consider original unscaled continuous features. y_t
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
```

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

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

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

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

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

| | features | importance |
|---|---|---|
| 2 | IsActiveMember | 0.476857 |
| 3 | country_France | 0.351836 |
| 0 | Gender | 0.096427 |
| 6 | Age | 0.032250 |
| 1 | HasCrCard | 0.028357 |
| 7 | NumOfProducts | 0.011373 |
| 4 | country_Germany | 0.002900 |
| 5 | country_Spain | 0.000000 |
| 8 | Surname_enc | 0.000000 |
| 9 | tenure_age_ratio | 0.000000 |

## Evaluating the model - Metrics

```
## 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)))
```

```
0.75147078296729290.7369402985074627array([[4835, 1477],
       [ 423, 1185]])              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
```

```
## 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)))
```

```
0.74773947583784110.7436974789915967array([[633, 209],
       [ 61, 177]])              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
```

## Plot decision boundaries of non-linear model

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

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

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

```
(7920, 10)(7920, 2)(7920,)
```

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

```
array([0.51069916, 0.48930078])
```

```
# 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))
```

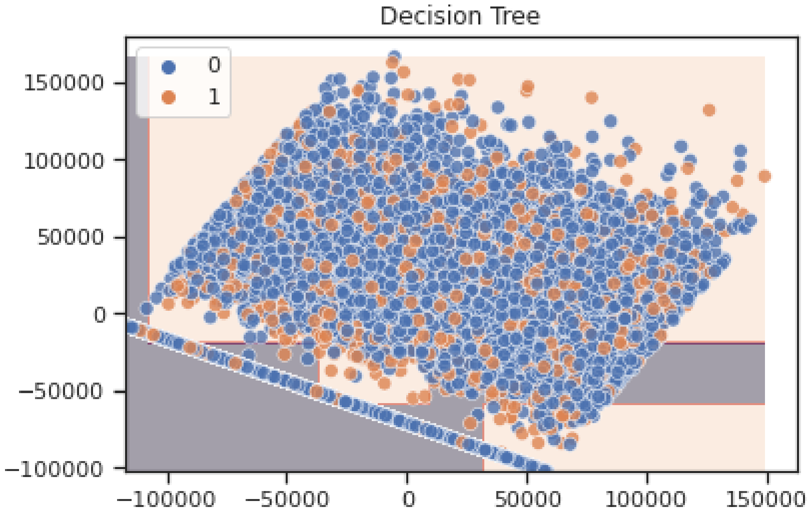
```
## Fitting tree model on 2 features
clf.fit(X, y)
```

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

```
## 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')
```

```
<matplotlib.contour.QuadContourSet at 0x7fe3500b28b0><AxesSubplot:>Text(0.5, 1.0,
'Decision Tree')
```




## Decision tree rule engine visualization

```
from sklearn.tree import export_graphviz
import subprocess


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

clf.fit(X_train, y_train)
```

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

```
## 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)


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


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

## 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

```
from sklearn.base import BaseEstimator, TransformerMixin


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_
        """

        Parameters
        ----------
        cols : list of str
            Columns to encode.  Default is to one-hot/target/label encode all categoric
        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].nunic

        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.coc
```

```
        ## 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


    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 val

        # 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))
                    else :
                        vals[ix] = ((y.sum() - y[ix])/(X.shape[0] - 1)).round(2) # Cate
                                                                                  # cate

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


        ## 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)
```

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

    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 feature
        """
        Xo = X.copy()
        ## Add 4 new columns - bal_per_product, bal_by_est_salary, tenure_age_ratio, ag
        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 feature
        """
        return self.fit(X,y).transform(X)




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

    def __init__(self, scale_cols = None):
```

```
        """
        Parameters
        ----------
        scale_cols : list of str
            Columns on which to perform scaling and normalization. Default is to scale

        """
        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 colum
        if self.scale_cols is None:
            self.scale_cols = [c for c in X if ((str(X[c].dtype).find('float') != -1) c


        ## 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)
```

## Pipeline in action for a single model

```
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, cl


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',
                ,'age_surname_enc']


weights_dict = {0 : 1.0, 1 : 3.92}

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


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


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

```
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))])
```

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


## 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))
```

```
0.74773947583784110.7436974789915967array([[633, 209],
       [ 61, 177]])              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
```

## 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?

```
from sklearn.model_selection import cross_val_score


## 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',
                ,'age_surname_enc']
```

```
3.93
```

```
## 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


## 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, r
                                                              , class_weight = weights_
                                                              , min_samples_split = 30,

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

        models['xgb_' + str(n_trees)] = XGBClassifier(objective='binary:logistic', n_es
                                                      , learning_rate = 0.03, n_jobs =
                                                      , reg_alpha = 0.3, reg_lambda = 0

        models['et_' + str(n_trees)] = ExtraTreesClassifier(n_estimators=n_trees, crite
                                                            , max_features = 0.6, n_job
                                                            , min_samples_split = 30, m

    # 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


## 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 i
    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



## 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 =

        # 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



## 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 = 0.7493527602020085, std_dev = 0.026176914665796896
Model lgb_21: mean = 0.7866856291480427, std_dev = 0.015745566437193475
Model xgb_21: mean = 0.7506085408564075, std_dev = 0.01096611280139578
Model et_21: mean = 0.7381861806079604, std_dev = 0.009033556110987941
Model rf_1001: mean = 0.7474932760588998, std_dev = 0.024780276266803267
Model lgb_1001: mean = 0.6884232116251622, std_dev = 0.014573973874519829
Model xgb_1001: mean = 0.6753719935759757, std_dev = 0.01756702999772903
Model et_1001: mean = 0.7363150867823766, std_dev = 0.0054959309820837516
Model knn_3: mean = 0.32214933921557243, std_dev = 0.021051639994704833
Model knn_5: mean = 0.2879356049612043, std_dev = 0.006396680440459953
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 = 0.6286545216621772, std_dev = 0.01880933233764158
Model lgb_21: mean = 0.6445713376921776, std_dev = 0.010347896896123705
Model xgb_21: mean = 0.6130509823329311, std_dev = 0.00848890204896738
Model et_21: mean = 0.590474996756568, std_dev = 0.0074631497300233106
Model rf_1001: mean = 0.6284716341377018, std_dev = 0.014863357989071506
Model lgb_1001: mean = 0.677231392541388, std_dev = 0.009841732603586511
```

```
Model xgb_1001: mean = 0.683463280904695, std_dev = 0.014982910608582397
Model et_1001: mean = 0.5911873424742697, std_dev = 0.00805199861616842
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 = 0.02449096874076205
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

## 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

```
from sklearn.pipeline import Pipeline
from sklearn.model_selection import GridSearchCV, RandomizedSearchCV
from lightgbm import LGBMClassifier


## 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
```

```
((7920, 17), (7920,))((1080, 17), (1080,))
```

```
lgb = LGBMClassifier(boosting_type = 'dart', min_child_samples = 20, n_jobs = - 1, impc
```

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

```
                                ('classifier', lgb)
                               ])
```

## Randomized Search

```
## 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
             }


search = RandomizedSearchCV(model, parameters, n_iter = 20, cv = 5, scoring = 'f1')


search.fit(X_train, y_train.ravel())
```

```
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')
```

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

```
{'classifier__reg_lambda': 0.1,
 'classifier__reg_alpha': 0.3,
 'classifier__num_leaves': 7,
 'classifier__n_estimators': 51,
 'classifier__max_depth': 4,
 'classifier__learning_rate': 0.5,
 'classifier__colsample_bytree': 0.6,
 'classifier__class_weight': {0: 1, 1: 1.96}}0.6908901259734914
```

```
search.cv_results_
```

```
{'mean_fit_time': array([0.08727965, 0.05784893, 1.93510203, 0.03740792,
0.49507413,
        0.06126742, 0.03452597, 0.54687095, 0.10516443, 7.26537404,
        0.08317885, 0.07134061, 0.11446533, 0.07934232, 0.04723072,
        0.12304249, 0.05450759, 3.67239733, 1.66865702, 0.83372941]),
 'std_fit_time': array([1.53254676e-02, 1.37640763e-03, 1.34075125e+00,
2.50293155e-04,
        4.90542198e-03, 5.88679884e-03, 8.73955289e-04, 1.40339212e-02,
        3.88539545e-02, 1.85160513e+00, 1.46063417e-03, 5.16900092e-03,
        1.71027942e-03, 7.36432179e-03, 2.64858427e-03, 6.81714524e-03,
        2.25822935e-03, 1.42797143e+00, 1.14545071e+00, 9.87372642e-03]),
 'mean_score_time': array([0.01131015, 0.00903459, 0.0297071 , 0.00800943,
0.01609659,
        0.00874305, 0.00813055, 0.01600647, 0.01341586, 0.04727616,
        0.01070886, 0.01015086, 0.01128192, 0.01052709, 0.00886149,
        0.01137843, 0.00854878, 0.04343362, 0.02109904, 0.01851449]),
 'std_score_time': array([1.00678809e-03, 8.03664294e-05, 1.11488620e-02,
8.33615825e-05,
        1.45504263e-04, 1.62825961e-04, 7.94391701e-04, 1.54808031e-04,
        4.33712967e-03, 1.98353342e-03, 8.91531322e-04, 4.03860169e-04,
        2.03547278e-04, 6.48571571e-04, 1.45974364e-04, 1.20196062e-03,
        6.52590283e-05, 1.88524434e-02, 1.03372858e-03, 2.82570843e-04]),
 'param_classifier__reg_lambda': masked_array(data=[0.1, 0.1, 10, 5, 10, 0.1,
1, 5, 5, 10, 1, 10, 0.5, 0.1,
                    0.5, 0.1, 5, 0.1, 1, 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__reg_alpha': masked_array(data=[0.3, 0.3, 0, 1, 0, 0, 1,
0.3, 0, 0.3, 0.3, 0, 0.3, 0.3,
                    1, 5, 1, 0, 0, 0.3],
              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=[7, 31, 7, 15, 31, 31, 15,
7, 7, 31, 31, 7, 31, 7, 15,
                    15, 15, 31, 31, 15],
              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=[51, 21, 350, 10, 201, 21,
10, 201, 51, 501, 51, 51, 51,
                     51, 21, 51, 21, 501, 201, 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=[4, 6, 9, 9, 3, 6, 4, 6, 4,
9, 4, 3, 6, 9, 6, 9, 6, 4,
                     6, 6],
```

## Grid Search

```
## 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}]
             }
```

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

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

```
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')
```

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

```
{'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}0.6827227378996369
```

```
grid.cv_results_
```

```
{'mean_fit_time': array([2.94769998, 1.95596223, 2.32641468, 2.53946695,
1.93624334,
        2.92211356, 1.80242229, 1.79475646, 2.75336413, 2.03374538,
        2.4281765 , 2.65032048, 2.72008262, 2.95253925, 2.04519801,
        2.56882243, 2.26601057, 1.92016759]),
 'std_fit_time': array([0.52887561, 0.03001612, 0.60437116, 0.67490923,
0.01183468,
        0.52840775, 0.03235945, 0.02422623, 0.49726065, 0.01284962,
        0.57697631, 0.38198122, 0.5527754 , 0.65604198, 0.01946636,
        0.7569963 , 0.38329618, 0.00856377]),
 'mean_score_time': array([0.05595851, 0.0365747 , 0.05523801, 0.04520683,
0.0363625 ,
        0.04027677, 0.03718095, 0.03641295, 0.06008372, 0.03825874,
        0.04899487, 0.04473743, 0.05487528, 0.06271949, 0.037467  ,
        0.06676121, 0.03748217, 0.03675299]),
 'std_score_time': array([0.02320697, 0.00025422, 0.01838449, 0.01762724,
0.00023008,
        0.00310354, 0.00135917, 0.00081168, 0.01884039, 0.00210534,
        0.01485137, 0.00959684, 0.02109968, 0.02173708, 0.00192224,
        0.01825172, 0.00085793, 0.00242326]),
 '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, 201, 201, 201, 201,
```

## Ensembles

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

```
## 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
```

```
((7920, 17), (7920,))((1080, 17), (1080,))
```

```
## 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_sa
                     , importance_type = 'gain', max_depth = 4, num_leaves = 31, colsam
                     , n_estimators = 21, reg_alpha = 0, reg_lambda = 0.5)

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

```
## 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)
                           ])


## 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())
```

```
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))])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))])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))])
```

```
## 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)


## 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[:,
df_t.shape
df_t.corr()
```

(7920, 3)

| | m1_pred | m2_pred | m3_pred |
|---|---|---|---|
| **m1_pred** | 1.000000 | 0.894747 | 0.911251 |
| **m2_pred** | 0.894747 | 1.000000 | 0.994593 |

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

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


## 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)


threshold = 0.5


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


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


## 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))
```

```
0.74693107646859220.592436974789916array([[759,  83],
       [ 97, 141]])              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
```

```
## 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))
```

```
0.75866783768139080.6218487394957983array([[754,  88],
       [ 90, 148]])              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
```

## 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

```
from sklearn.linear_model import LogisticRegression


## 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)
```

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

```
## 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)
```

```
## 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))
```

```
0.74633725224056380.592436974789916array([[758,  84],
       [ 97, 141]])              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
```

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

```
array([[-6.06252409, 12.94656529]])array([-5.65280526])
```

## Error analysis

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

```
## 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
```

```
((7920, 17), (7920,))((1080, 17), (1080,))
```

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

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

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

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

```
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))])
```

```
## 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]
```

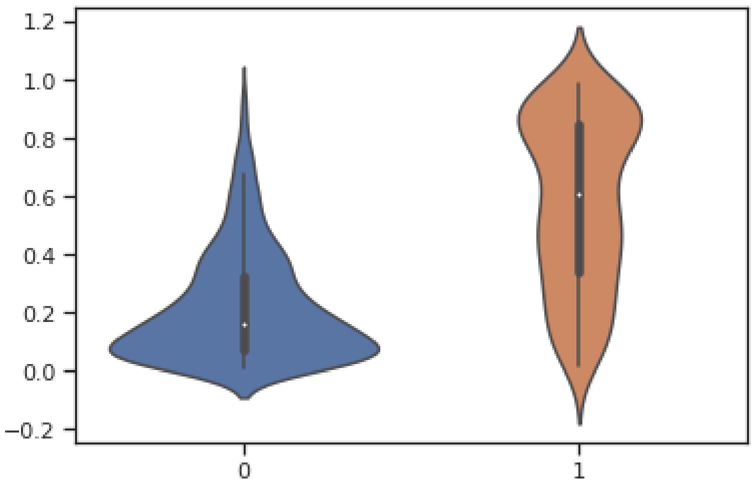
```
df_ea.shape
df_ea.sample(5)
```

```
(1080, 20)
```

| | CreditScore | Gender | Age | Tenure | Balance | NumOfProducts | HasCrCard | IsActiv |
|---|---|---|---|---|---|---|---|---|
| **187** | 677 | 0 | 44 | 4 | 148770.61 | 2 | 1 | |
| **430** | 709 | 0 | 36 | 7 | 0.00 | 1 | 0 | |
| **803** | 716 | 1 | 31 | 8 | 109578.04 | 2 | 1 | |
| **507** | 526 | 0 | 33 | 8 | 114634.63 | 2 | 1 | |
| **720** | 727 | 1 | 28 | 5 | 0.00 | 2 | 0 | |

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

```
<AxesSubplot:>
```

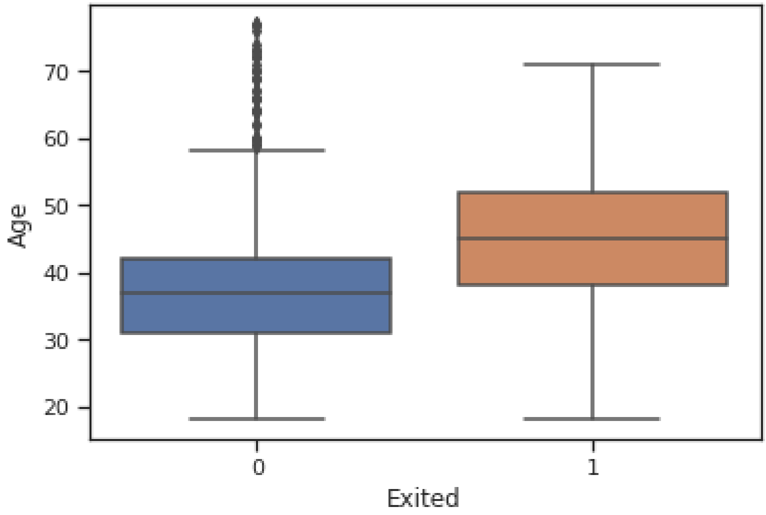


## Revisiting bivariate plots of important features

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

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

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




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

```
0    0.77963
1    0.22037
Name: Exited, dtype: float640    0.560185
1    0.439815
Name: Exited, dtype: float640    0.481481
1    0.518519
Name: y_pred, dtype: float64
```

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

| | y_pred | Exited |
|---|---|---|
| **CreditScore** | -0.016600 | -0.026118 |

## Extracting the subset of incorrect predictions

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

| | | |
|---|---|---|
| **NumOfProducts** | -0.150982 | -0.125494 |

```
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()
```

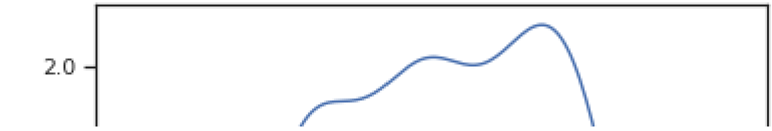
```
(97, 20)(83, 20)
```

| | CreditScore | Gender | Age | Tenure | Balance | NumOfProducts | HasCrCard | IsActive |
|---|---|---|---|---|---|---|---|---|
| **5** | 706 | 0 | 23 | 5 | 0.00 | 1 | 0 | |
| **21** | 611 | 1 | 35 | 10 | 0.00 | 1 | 1 | |
| **38** | 491 | 0 | 68 | 1 | 95039.12 | 1 | 0 | |
| **58** | 637 | 1 | 43 | 1 | 135645.29 | 2 | 0 | |
| **92** | 717 | 0 | 36 | 2 | 99472.76 | 2 | 1 | |

| | CreditScore | Gender | Age | Tenure | Balance | NumOfProducts | HasCrCard | IsActive |
|---|---|---|---|---|---|---|---|---|
| **48** | 512 | 1 | 39 | 3 | 0.00 | 1 | 1 | |
| **49** | 736 | 1 | 43 | 4 | 202443.47 | 1 | 1 | |
| **57** | 505 | 1 | 43 | 6 | 127146.68 | 1 | 0 | |
| **75** | 648 | 1 | 41 | 5 | 123049.21 | 1 | 0 | |
| **99** | 631 | 1 | 51 | 8 | 100654.80 | 1 | 1 | |

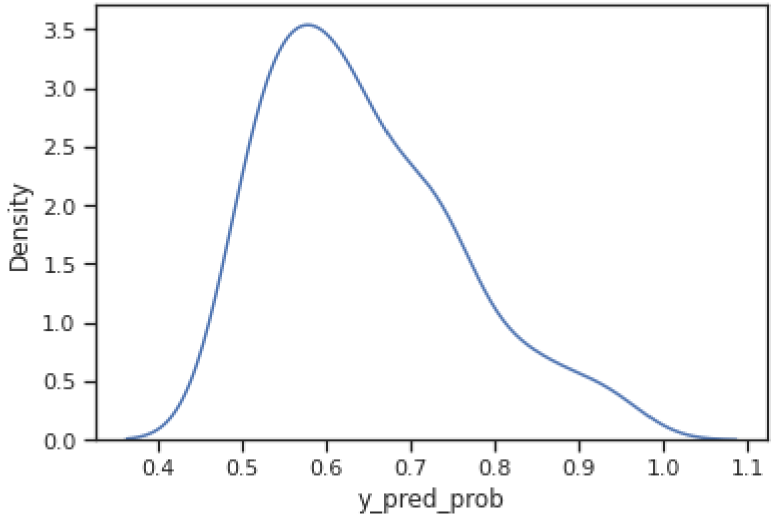
```
## Prediction probabilty distribution of errors causing low recall
sns.distplot(low_recall.y_pred_prob, hist=False)
```

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




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

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




## Tweaking the threshold of classifier

```
threshold = 0.55
```

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

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

```
array([[778,  64],
       [110, 128]])              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
```

```
## Tweaking threshold between 0.4 and 0.6
confusion_matrix(y_val, val_preds)
print(classification_report(y_val, val_preds))
```

```
array([[778,  64],
       [110, 128]])              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

```
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()
```

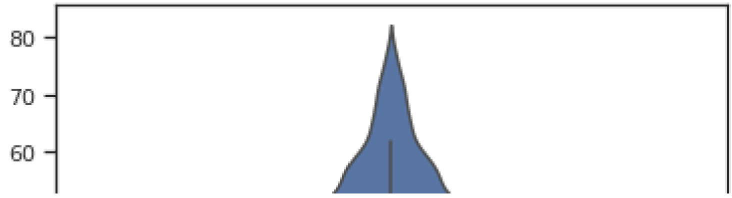
```
1    0.506481
2    0.467593
3    0.020370
4    0.005556
Name: NumOfProducts, dtype: float641    0.701031
2    0.288660
3    0.010309
Name: NumOfProducts, dtype: float641    0.819277
2    0.156627
3    0.024096
Name: NumOfProducts, dtype: float64
```

```
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()
```

```
0    0.481481
1    0.518519
Name: IsActiveMember, dtype: float640    0.556701
1    0.443299
Name: IsActiveMember, dtype: float640    0.626506
1    0.373494
Name: IsActiveMember, dtype: float64
```

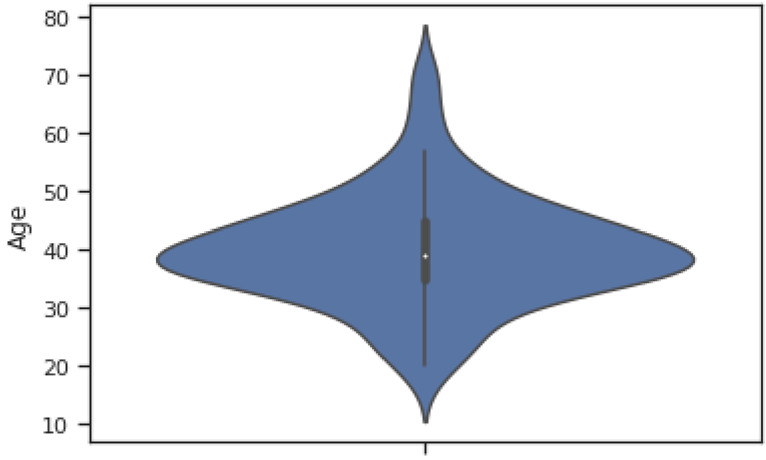
```
sns.violinplot(y = df_ea.Age)
```

<AxesSubplot:ylabel='Age'>



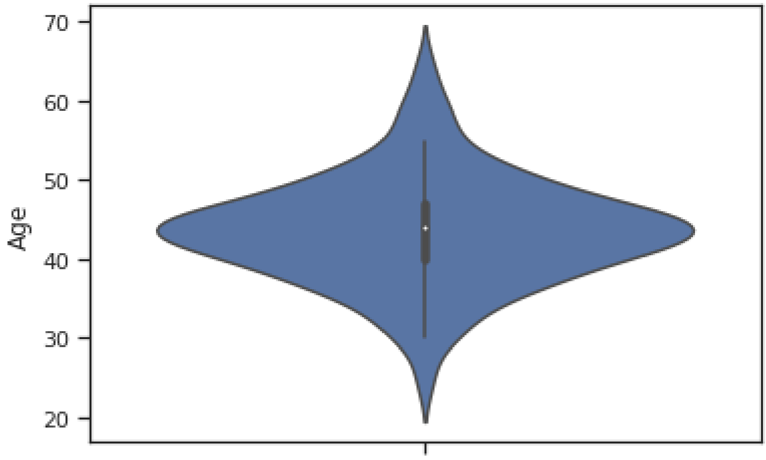

```
sns.violinplot(y = low_recall.Age)
```

<AxesSubplot:ylabel='Age'>




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

<AxesSubplot:ylabel='Age'>

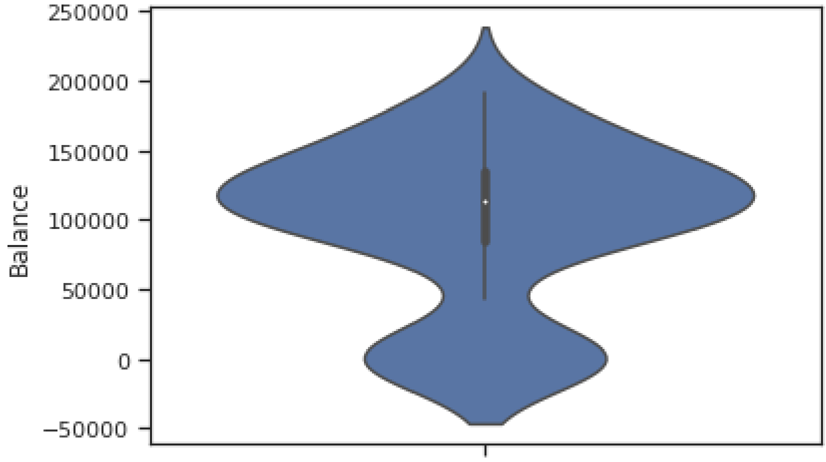



```
sns.violinplot(y = df_ea.Balance)
```

<AxesSubplot:ylabel='Balance'>

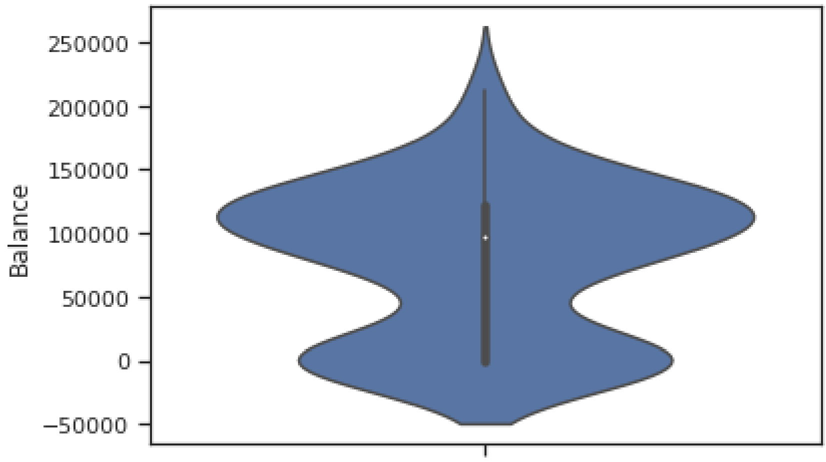

```
sns.violinplot(y = low_recall.Balance)
```

<AxesSubplot:ylabel='Balance'>




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

<AxesSubplot:ylabel='Balance'>




## Train final, best model ; Save model and its parameters

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

```
## Re-defining X_train and X_val to consider original unscaled continuous features. y_t
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
```

```
((7920, 17), (7920,))((1080, 17), (1080,))
```

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

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

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

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

```
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))])
```

```
# 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 tweakec
```

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

```
<AxesSubplot:>
```

```
## 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))
```

```
0.75875765983352970.6386554621848739array([[740, 102],
       [ 86, 152]])              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
```

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

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

## Load saved model and make predictions on unseen/future data

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

```
import joblib
```

```
## Load model object
model = joblib.load('final_churn_model_f1_0_45.sav')
```

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

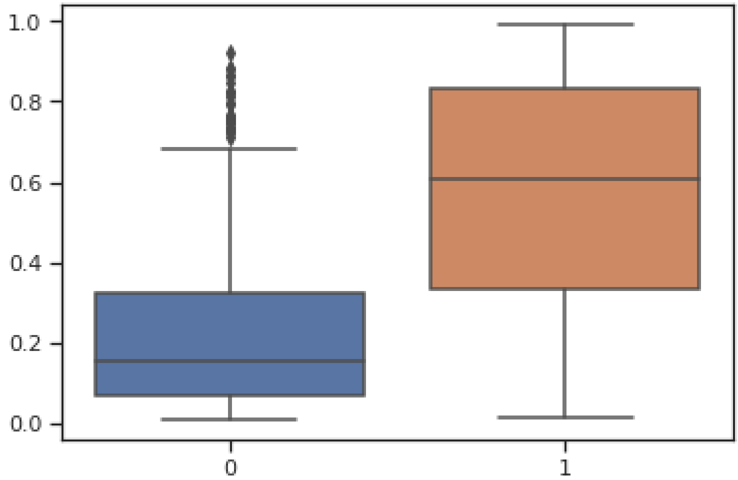
```
(1000, 17)(1000,)
```

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

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

```
sns.boxplot(y_test.ravel(), test_probs)
```

<AxesSubplot:>




```
## 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))
```

```
0.76785702729114210.675392670157068array([[696, 113],
       [ 62, 129]])              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
```

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

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

| | CreditScore | Gender | Age | Tenure | Balance | NumOfProducts | HasCrCard | IsActiv |
|---|---|---|---|---|---|---|---|---|
| **615** | 850 | 0 | 31 | 3 | 51293.47 | 1 | 0 | |
| **233** | 745 | 0 | 36 | 9 | 0.00 | 1 | 1 | |
| **263** | 518 | 0 | 41 | 5 | 110624.99 | 1 | 1 | |
| **424** | 521 | 0 | 41 | 2 | 0.00 | 2 | 1 | |

## 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

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

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

```
(103, 18)
```

| | CreditScore | Gender | Age | Tenure | Balance | NumOfProducts | HasCrCard | IsActiveM |
|---|---|---|---|---|---|---|---|---|
| **0** | 546 | 0 | 58 | 3 | 106458.31 | 4 | 1 | |
| **1** | 479 | 1 | 51 | 1 | 107714.74 | 3 | 1 | |