# Supervised Machine Learning Project

## *Churn prediction model for a bank*

# Table of Contents

## Goal

Develop a binary classification model for Beta Bank that would analyze data on clients' past behavior and termination of contracts with the bank and predict whether a customer will leave the bank soon.

## Data description

### Features

- *RowNumber* — data string index
- *CustomerId* — unique customer identifier
- *Surname* — surname
- *CreditScore* — credit score
- *Geography* — country of residence
- *Gender* — gender
- *Age* — age
- *Tenure* — years with the bank
- *Balance* — account balance
- *NumOfProducts* — number of banking products used by the customer
- *HasCrCard* — customer has a credit card
- *IsActiveMember* — customer's activeness
- *EstimatedSalary* — estimated salary

### Target

- *Exited* — customer has left

## Imports

In [1369]:

```
import pandas as pd
import matplotlib
import numpy as np
import seaborn as sns
import re
from sklearn.utils import shuffle

from sklearn.preprocessing import StandardScaler as ss
from sklearn.model_selection import train_test_split
from sklearn.metrics import f1_score
from sklearn.metrics import roc_curve
from sklearn.metrics import roc_auc_score

from sklearn.dummy import DummyClassifier

from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression

import matplotlib.pyplot as plt
%matplotlib inline

import sys
import warnings
if not sys.warnoptions:
       warnings.simplefilter("ignore")

pd.set_option('display.max_rows', None)

print("Setup Complete")
```

```
Setup Complete
```

## Input data

In [1308]:

```
try:
    df = pd.read_csv('Churn.csv')

except:
    df = pd.read_csv('/datasets/Churn.csv')
```

## Descriptive statistics

In [1309]:

```
df.head()
```

Out[1309]:

| | RowNumber | CustomerId | Surname | CreditScore | Geography | Gender | Age | Tenure | Bala |
|---|---|---|---|---|---|---|---|---|---|
| **0** | 1 | 15634602 | Hargrave | 619 | France | Female | 42 | 2.0 | ( |
| **1** | 2 | 15647311 | Hill | 608 | Spain | Female | 41 | 1.0 | 83807 |
| **2** | 3 | 15619304 | Onio | 502 | France | Female | 42 | 8.0 | 159660 |
| **3** | 4 | 15701354 | Boni | 699 | France | Female | 39 | 1.0 | ( |
| **4** | 5 | 15737888 | Mitchell | 850 | Spain | Female | 43 | 2.0 | 125510 |

Notes for preprocessing:

- The `RowNumber` and `CustomerId` columns are basically indexes, `RowNumber` starts with 1 but it's better to have index start with 0, so we'll drop these two columns and keep the automatic index created when our data frame was loaded;
- Check if `Tenure` has floats, if not, change to integer;
- Change column names to lower case.

In [1310]:

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10000 entries, 0 to 9999
Data columns (total 14 columns):
 #   Column           Non-Null Count  Dtype  
---  ------           --------------  -----  
 0   RowNumber        10000 non-null  int64  
 1   CustomerId       10000 non-null  int64  
 2   Surname          10000 non-null  object 
 3   CreditScore      10000 non-null  int64  
 4   Geography        10000 non-null  object 
 5   Gender           10000 non-null  object 
 6   Age              10000 non-null  int64  
 7   Tenure           9091 non-null   float64
 8   Balance          10000 non-null  float64
 9   NumOfProducts    10000 non-null  int64  
 10  HasCrCard        10000 non-null  int64  
 11  IsActiveMember   10000 non-null  int64  
 12  EstimatedSalary  10000 non-null  float64
 13  Exited           10000 non-null  int64  
dtypes: float64(3), int64(8), object(3)
memory usage: 1.1+ MB
```

Notes for preprocessing:

- missing values in the `Tenure` variable;
- data types seem fine (except for maybe `Tenure` but it needs to be checked).

In [1311]:

```
df.describe()
```

Out[1311]:

| | RowNumber | CustomerId | CreditScore | Age | Tenure | Balance | I |
|---|---|---|---|---|---|---|---|
| **count** | 10000.00000 | 1.000000e+04 | 10000.000000 | 10000.000000 | 9091.000000 | 10000.000000 | |
| **mean** | 5000.50000 | 1.569094e+07 | 650.528800 | 38.921800 | 4.997690 | 76485.889288 | |
| **std** | 2886.89568 | 7.193619e+04 | 96.653299 | 10.487806 | 2.894723 | 62397.405202 | |
| **min** | 1.00000 | 1.556570e+07 | 350.000000 | 18.000000 | 0.000000 | 0.000000 | |
| **25%** | 2500.75000 | 1.562853e+07 | 584.000000 | 32.000000 | 2.000000 | 0.000000 | |
| **50%** | 5000.50000 | 1.569074e+07 | 652.000000 | 37.000000 | 5.000000 | 97198.540000 | |
| **75%** | 7500.25000 | 1.575323e+07 | 718.000000 | 44.000000 | 7.000000 | 127644.240000 | |
| **max** | 10000.00000 | 1.581569e+07 | 850.000000 | 92.000000 | 10.000000 | 250898.090000 | |

Notes for data preprocessing:

- most features seem to be normally distributed as their mean and median values are close to each other. Maximum values are mostly within 3 std from the mean value. Maybe these are a few outliers, we'll check that in the EDA section.
- `Balance` and `EstimatedSalary` variables have expectedly high variance and they are slightly positively skewed. The minimum of `EstimatedSalary` is very low, we need to check it out as well;
- the target variable is binary, it has values: 1 when the customer left and 0 when he stayed. Most customers stayed, the classes are imbalanced, so we will try to apply some techniques to correct that.

## Preprocessing

### Lower case column names

In [1312]:

```
columns = []
for name in df.columns.values:
    name = re.sub('([A-Z])', r' \1', name).lower().replace(' ', '_')[1:]
    columns.append(name)
```

In [1313]:

```
df.columns = columns
```

In [1314]:

```
df.head()
```

Out[1314]:

| | row_number | customer_id | surname | credit_score | geography | gender | age | tenure | balan |
|---|---|---|---|---|---|---|---|---|---|
| **0** | 1 | 15634602 | Hargrave | 619 | France | Female | 42 | 2.0 | 0 |
| **1** | 2 | 15647311 | Hill | 608 | Spain | Female | 41 | 1.0 | 83807 |
| **2** | 3 | 15619304 | Onio | 502 | France | Female | 42 | 8.0 | 159660 |
| **3** | 4 | 15701354 | Boni | 699 | France | Female | 39 | 1.0 | 0 |
| **4** | 5 | 15737888 | Mitchell | 850 | Spain | Female | 43 | 2.0 | 125510 |

### Row number and Customer ID

In [1315]:

```
df = df.drop(['customer_id','row_number'], axis=1)
```

### Tenure

In [1316]:

```
df['tenure'].value_counts()
```

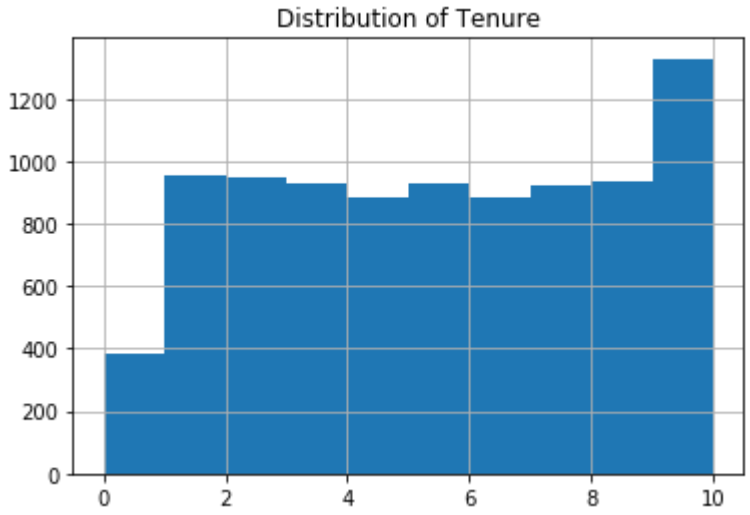
Out[1316]:

```
1.0     952
2.0     950
8.0     933
3.0     928
5.0     927
7.0     925
4.0     885
9.0     882
6.0     881
10.0    446
0.0     382
Name: tenure, dtype: int64
```

All values are integers in essence, so let's first fill missing values and then convert them to the proper data type.

In [1317]:

```
df['tenure'].hist()
plt.title('Distribution of Tenure');
```




In [1318]:

```
df['tenure'].isnull().sum()
```

Out[1318]:

```
909
```

There are 909 missing values in this column. The distribution of `tenure` is close to uniform. We will fill the missing values with the mean per `num_of_products` group.

In [1319]:

```
df['tenure'] = df.groupby('num_of_products')['tenure'].apply(lambda x: x.fillna(
x.mean()))
```

In [1320]:

```
df['tenure'].isnull().sum()
```

Out[1320]:

```
0
```

Finally, let's convert them to integers.

In [1321]:

```
df['tenure'] = df['tenure'].astype(int)
```

### Categorical features encoding

First, let's analyze our categorical variables and decide which ones to convert and with which method

In [1322]:

```
df['geography'].value_counts()
```

Out[1322]:

```
France     5014
Germany    2509
Spain      2477
Name: geography, dtype: int64
```

`geography` has only 3 category, so it will be easy to encode it using OHE.

In [1323]:

```
df['surname'].value_counts().count()
```

Out[1323]:

```
2932
```

This is a very high-cardinality variable but it doesn't give any useful information about the target, so we will simply drop it.

In [1324]:

```
df = df.drop('surname', axis=1)
```

In [1325]:

```
df['gender'].value_counts()
```

Out[1325]:

```
Male      5457
Female    4543
Name: gender, dtype: int64
```

`gender` is a binary variable, both values are equally represented in the data frame. We'll use OHE method for it as well. We will drop the first column for each encoded feature to avoid the dummy trap.

In [1326]:

```
df_OHE = pd.get_dummies(df, drop_first=True)
df_OHE.head()
```

Out[1326]:

| | credit_score | age | tenure | balance | num_of_products | has_cr_card | is_active_member | est |
|---|---|---|---|---|---|---|---|---|
| **0** | 619 | 42 | 2 | 0.00 | 1 | 1 | 1 | |
| **1** | 608 | 41 | 1 | 83807.86 | 1 | 0 | 1 | |
| **2** | 502 | 42 | 8 | 159660.80 | 3 | 1 | 0 | |
| **3** | 699 | 39 | 1 | 0.00 | 2 | 0 | 0 | |
| **4** | 850 | 43 | 2 | 125510.82 | 1 | 1 | 1 | |

### Duplicates

Let's check if any rows are duplicated in any data frames.

In [1327]:

```
df.duplicated().sum()
```

Out[1327]:

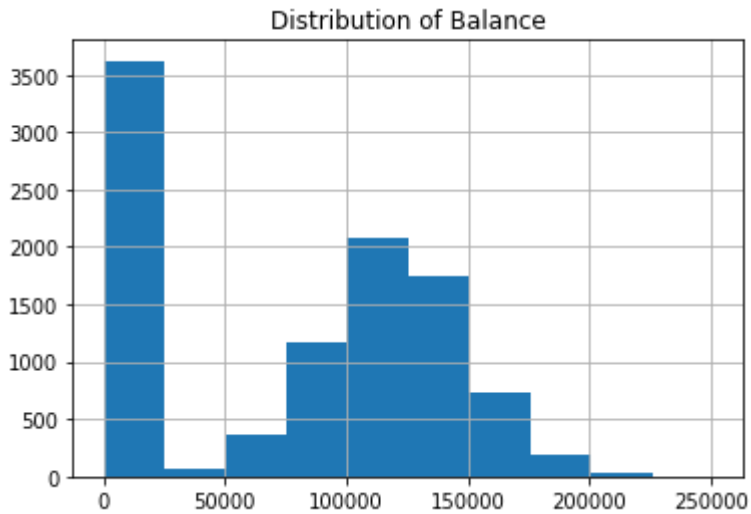
0

## EDA

### Balance

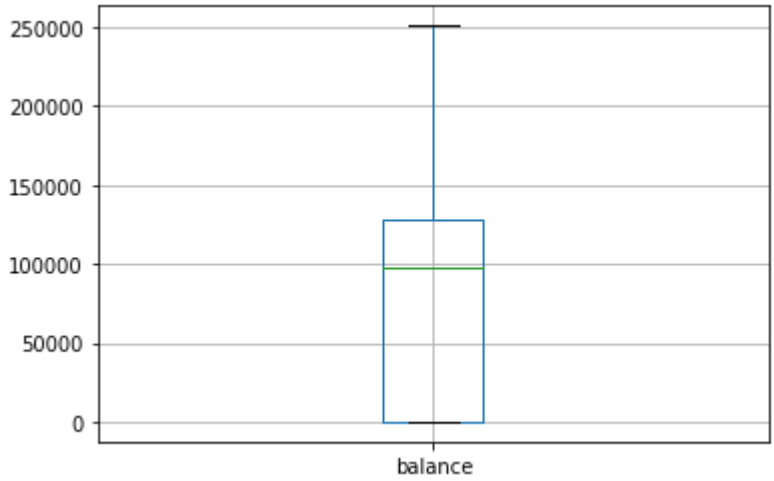
In [1328]:

```
df['balance'].hist()
plt.title('Distribution of Balance');
```



In [1329]:

```
df.boxplot('balance');
```




In [1330]:

```
df[df['balance']==0].head()
```

Out[1330]:

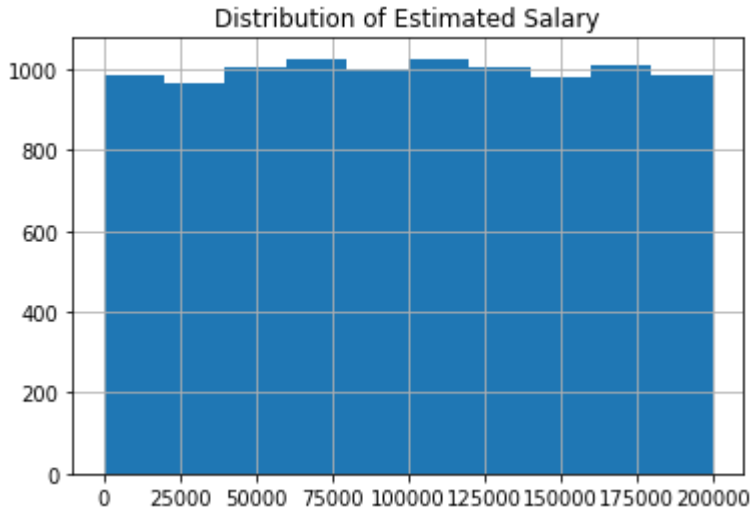
| | credit_score | geography | gender | age | tenure | balance | num_of_products | has_cr_card | is |
|---|---|---|---|---|---|---|---|---|---|
| **0** | 619 | France | Female | 42 | 2 | 0.0 | 1 | 1 | |
| **3** | 699 | France | Female | 39 | 1 | 0.0 | 2 | 0 | |
| **6** | 822 | France | Male | 50 | 7 | 0.0 | 2 | 1 | |
| **11** | 497 | Spain | Male | 24 | 3 | 0.0 | 2 | 1 | |
| **12** | 476 | France | Female | 34 | 10 | 0.0 | 2 | 1 | |

This distribution is close to normal except for a high number of 0 values. Some of them are observations when a customer left the bank, so 0 balance might be a good predictor for our target. However, we see that customers who stayed (at least at the moment when the data was taken) also sometimes have 0 balances. Probably more information is needed to understand the reason behind.

### Estimated Salary

In [1331]:

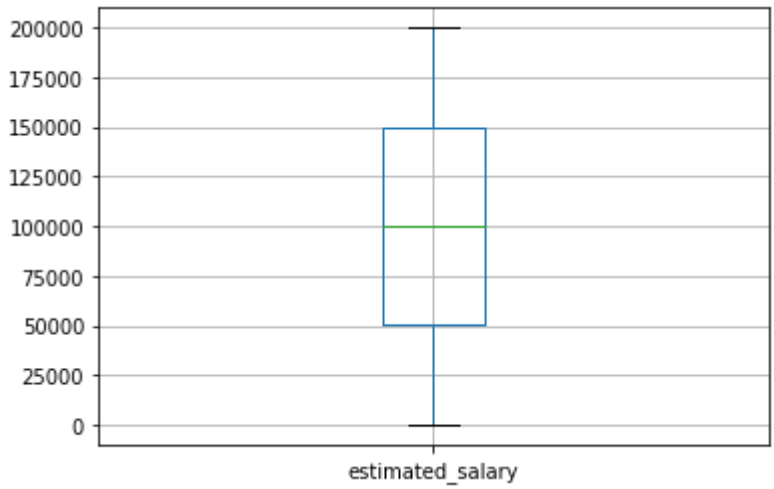
```
df['estimated_salary'].hist()
plt.title('Distribution of Estimated Salary');
```




It looks like a uniform distribution. It implies that each range of values that has the same length on the distributions support has equal probability of occurrence.

In [1332]:

```
df.boxplot('estimated_salary');
```

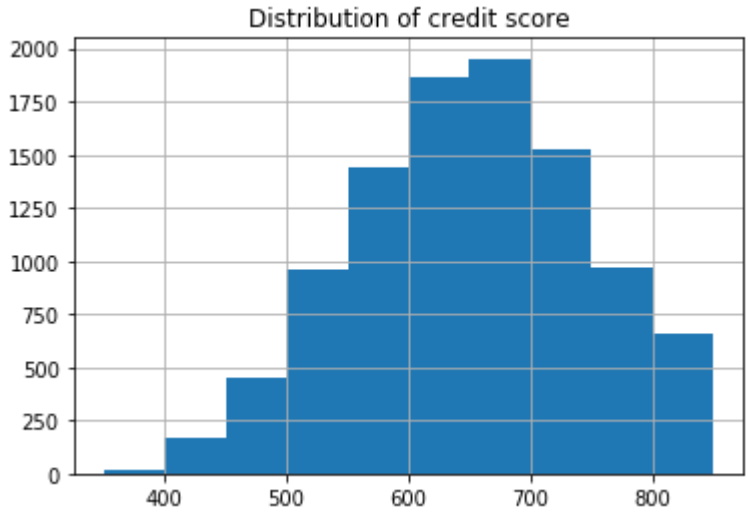



No visible outliers here.

### Credit Score

In [1333]:

```
df['credit_score'].hist()
plt.title('Distribution of credit score');
```

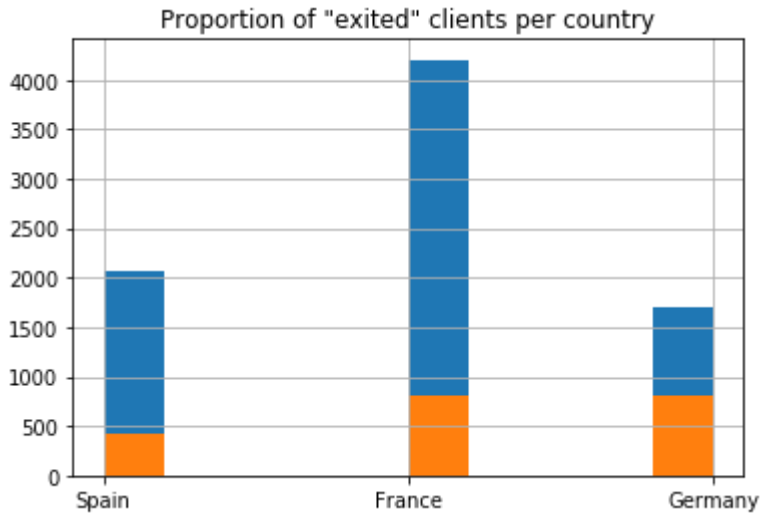



Distribution close to normal.

### Geography

In [1334]:

```
df.groupby('exited')['geography'].hist()
plt.title('Proportion of "exited" clients per country');
```

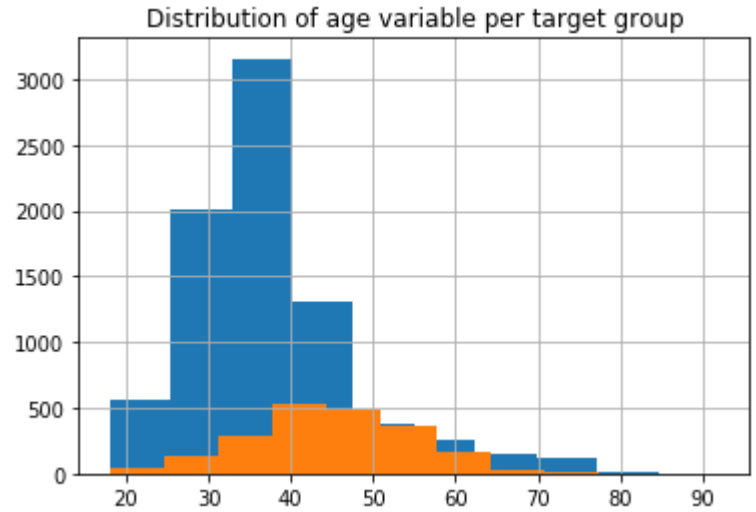



Most clients are located in France and they are the most loyal to the bank among all countries, on average. The highest churn rate is in Germany. This features might be quite a good predictor for our model.

### Age

In [1335]:

```
df.groupby('exited')['age'].hist()
plt.title('Distribution of age variable per target group');
```

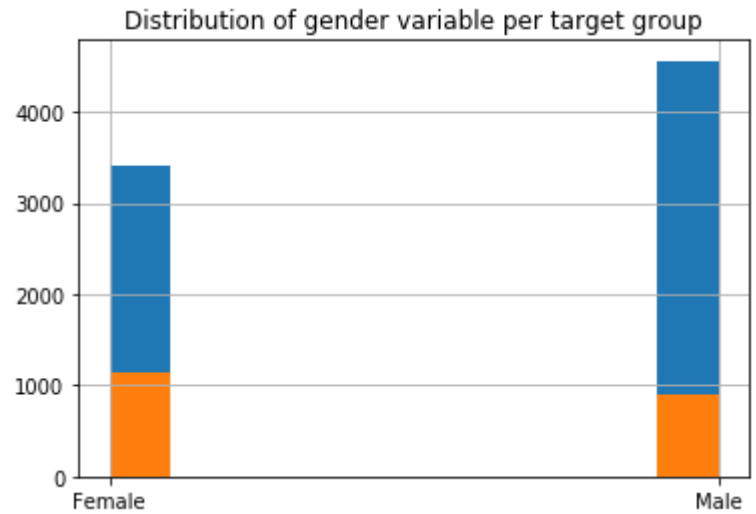



Most customers are in their 30s, the distribution is positively skewed as less elder than younger people have bank accounts. Interestingly that the distribution of "exited" customers (in orange) is closer to normal than that of loyal clients.

### Gender

In [1336]:

```
df.groupby('exited')['gender'].hist()
plt.title('Distribution of gender variable per target group');
```

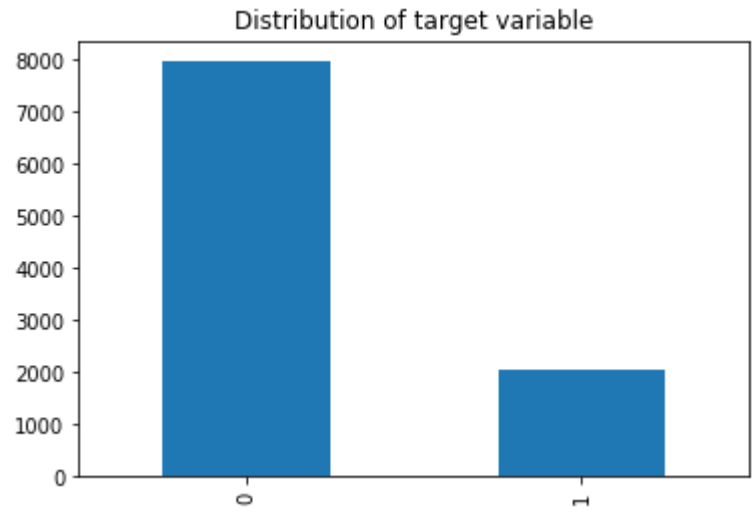



Most clients are male. Female clients leave the bank more often.

### Target analysis

In [1337]:

```
df['exited'].value_counts().plot(kind='bar')
plt.title('Distribution of target variable');
```




We see that our classes are indeed imbalanced: there are more than twice as many observations when a client stayed with the bank than of those who left.

First, let's calculate the baseline F1 score for this dataset and try to run models without taking into account the imbalance.

## Splitting data into train, validation and test sets

First, let's split data into train and test sets with the 80/20 proportion, respectively.

In [1338]:

```
X = df_OHE.drop('exited', axis=1)
y = df_OHE['exited']
X_train_all, X_test, y_train_all, y_test = train_test_split(X, y, test_size = 0.
2, stratify=y, random_state=12345)
```

This stratify parameter makes a split so that the proportion of values in the sample produced will be the same as the proportion of values on the target variable.

For example, if variable y is a binary categorical variable with values 0 and 1 and there are 25% of zeros and 75% of ones, stratify=y will make sure that your random split has 25% of 0's and 75% of 1's.

Next, we'll further split the train set into train and validation with the 80/20 proportion, respectively.

In [1339]:

```
X_train, X_valid, y_train, y_valid = train_test_split(X_train_all, y_train_all,
test_size = 0.2, random_state=12345)
```

In [1340]:

```
X_test.name = 'X_test'
X_valid.name = 'X_valid'
y_train.name = 'y_train'
y_valid.name = 'y_valid'
y_test.name = 'y_test'
X_train.name = 'X_train'
```

In [1341]:

```
for part in [X_train, y_train, X_valid, y_valid, X_test, y_test]:
    print("Size of", part.name, ":", part.shape[0]/df.shape[0])
```

```
Size of X_train : 0.64
Size of y_train : 0.64
Size of X_valid : 0.16
Size of y_valid : 0.16
Size of X_test : 0.2
Size of y_test : 0.2
```

## Standard Scaling

Finally, we will scale our features with Standard Scaler. It will convert our data frames into numpy arrays, so after they are transformed, let's convert them back to data frames.

In [1342]:

```
sc = ss()
X_train_scaled = sc.fit_transform(X_train)
X_valid_scaled = sc.transform(X_valid)
X_test_scaled = sc.transform(X_test)
```

In [1343]:

```
X_train = pd.DataFrame(data=X_train_scaled,
                       index=X_train.index,
                       columns=X_train.columns)
```

In [1344]:

```
X_valid = pd.DataFrame(data=X_valid_scaled,
                       index=X_valid.index,
                       columns=X_valid.columns)
```

In [1345]:

```
X_test = pd.DataFrame(data=X_test_scaled,
                       index=X_test.index,
                       columns=X_test.columns)
```

## Models with class imbalance

### Preliminary F1 score (baseline)

In our case it is more important that a model correctly predicts the minority class - whether a customer left the bank, that's why we will choose the strategy `constant` for the dummy classifier.

In [1346]:

```
base_model = DummyClassifier(strategy='constant', constant=1, random_state=12345
)
base_model.fit(X_train, y_train)
y_prelim_pred = base_model.predict(X_valid)
f1_baseline = round(f1_score(y_valid,y_prelim_pred) * 100, 2)
print("Baseline F1 score:", f1_baseline)
```

```
Baseline F1 score: 34.37
```

It means that to pass the sanity check our model must do better than 34% of the F1 metric.

### Decision Tree

Default model F1 score:

In [1347]:

```
decision_tree = DecisionTreeClassifier(random_state=12345)
decision_tree.fit(X_train, y_train)
y_pred = decision_tree.predict(X_valid)

f1_DT_imbalance = round(f1_score(y_valid,y_pred) * 100, 2)
f1_DT_imbalance
```

Out[1347]:

```
46.91
```

### Random Forest

Default model F1 score:

In [1348]:

```
rfc = RandomForestClassifier(random_state=12345)
rfc.fit(X_train, y_train)

y_pred = rfc.predict(X_valid)

f1_rfc_imbalance = round(f1_score(y_valid,y_pred) * 100, 2)
f1_rfc_imbalance
```

Out[1348]:

```
55.32
```

### Logistic Regression

Default model F1 score:

In [1349]:

```
LR = LogisticRegression(random_state=12345)
LR.fit(X_train, y_train)
y_pred = LR.predict(X_valid)

f1_LR_imbalance = round(f1_score(y_valid,y_pred) * 100, 2)
f1_LR_imbalance
```

Out[1349]:

```
32.52
```

As we can see, all 3 algorithms gave F1 score lower than our target number - 59%. Default Logistic Regression was even worse than the dummy classifier. Let's try to correct class imbalance for this model run it again.

## Class imbalance correction and hyper parameter tuning

In this section we will try to correct class imbalance using the following methods:

- Class Weight Adjustment;
- Upsampling;
- Downsampling.

We will also tune each of the 3 chosen algorithms and find the best hyperparameters.

#### Method 1: Class Weight Adjustment

***Logistic Regression***

In [1350]:

```
d = []
for penalty in ['l1', 'l2']:
    for C in np.arange(0.01,1,0.01):
        LR = LogisticRegression(solver="liblinear", penalty=penalty, class_weigh
t='balanced', C=C, random_state=12345)
        LR.fit(X_train, y_train)
        y_pred = LR.predict(X_valid)
        f1_LR_balanced = round(f1_score(y_valid,y_pred) * 100, 2)
        d.append(
            {
                'penalty': penalty,
                'C': C,
                'f1_LR_balanced':  f1_LR_balanced
            }
        )
best_param = pd.DataFrame(d).nlargest(1, ['f1_LR_balanced'], keep='first')
f1_LR_balanced = best_param['f1_LR_balanced'].values
best_param
```

Out[1350]:

| | penalty | C | f1_LR_balanced |
|---|---|---|---|
| **0** | l1 | 0.01 | 50.57 |

### *Random Forest*

In [1351]:

```
d = []
for estim in range(1,51,9):
    for depth in range(1,10):
        rfc = RandomForestClassifier(random_state=12345, n_estimators=estim, max
_depth=depth, class_weight='balanced')
        rfc.fit(X_train, y_train)
        y_pred = rfc.predict(X_valid)
        f1_rfc_balanced = round(f1_score(y_valid,y_pred) * 100, 2)
        d.append(
            {
                'n_estimators': estim,
                'max_depth': depth,
                'f1_rfc_balanced':  f1_rfc_balanced
            }
        )

best_param = pd.DataFrame(d).nlargest(1, ['f1_rfc_balanced'], keep='first')
f1_rfc_balanced = best_param['f1_rfc_balanced'].values
best_param
```

Out[1351]:

| | n_estimators | max_depth | f1_rfc_balanced |
|---|---|---|---|
| **51** | 46 | 7 | 62.33 |

### *Decision Tree*

In [1352]:

```
d = []
for depth in range(1,10):
    decision_tree = DecisionTreeClassifier(class_weight='balanced', random_state
=12345, max_depth=depth)
    decision_tree.fit(X_train, y_train)
    y_pred = decision_tree.predict(X_valid)
    f1_DT_balanced = round(f1_score(y_valid,y_pred) * 100, 2)
    d.append(
        {
            'max_depth': depth,
            'f1_DT_balanced':  f1_DT_balanced
        }
     )

best_param = pd.DataFrame(d).nlargest(1, ['f1_DT_balanced'], keep='first')
f1_DT_balanced = best_param['f1_DT_balanced'].values
best_param
```

Out[1352]:

| | max_depth | f1_DT_balanced |
|---|---|---|
| **5** | 6 | 61.95 |

### Method 2: Upsampling

In [1353]:

```
def upsample(features, target, repeat):
    features_zeros = features[target == 0]
    features_ones = features[target == 1]
    target_zeros = target[target == 0]
    target_ones = target[target == 1]

    features_upsampled = pd.concat([features_zeros] + [features_ones] * repeat)
    target_upsampled = pd.concat([target_zeros] + [target_ones] * repeat)

    features_upsampled, target_upsampled = shuffle(
        features_upsampled, target_upsampled, random_state=12345)

    return features_upsampled, target_upsampled

X_train_upsampled, y_train_upsampled = upsample(X_train, y_train, 10)
```

### *Logistic Regression*

In [1354]:

```
d = []
for penalty in ['l1', 'l2']:
    for C in np.arange(0.01,1,0.01):
        LR = LogisticRegression(solver="liblinear", penalty=penalty, C=C, random
_state=12345)
        LR.fit(X_train_upsampled, y_train_upsampled)
        y_pred = LR.predict(X_valid)
        f1_LR_upsampled = round(f1_score(y_valid,y_pred) * 100, 2)
        d.append(
            {
                'penalty': penalty,
                'C': C,
                'f1_LR_upsampled':  f1_LR_upsampled
            }
        )
best_param = pd.DataFrame(d).nlargest(1, ['f1_LR_upsampled'], keep='first')
f1_LR_upsampled = best_param['f1_LR_upsampled'].values
best_param
```

Out[1354]:

| | penalty | C | f1_LR_upsampled |
|---|---|---|---|
| **30** | l1 | 0.31 | 41.73 |

### *Random Forest*

In [1355]:

```
d = []
for estim in range(1,51,9):
    for depth in range(1,10):
        rfc = RandomForestClassifier(random_state=12345, n_estimators=estim, max
_depth=depth)
        rfc.fit(X_train_upsampled, y_train_upsampled)
        y_pred = rfc.predict(X_valid)
        f1_rfc_upsampled = round(f1_score(y_valid,y_pred) * 100, 2)
        d.append(
            {
                'n_estimators': estim,
                'max_depth': depth,
                'f1_rfc_upsampled':  f1_rfc_upsampled
            }
        )

best_param = pd.DataFrame(d).nlargest(1, ['f1_rfc_upsampled'], keep='first')
f1_rfc_upsampled = best_param['f1_rfc_upsampled'].values
best_param
```

Out[1355]:

| | n_estimators | max_depth | f1_rfc_upsampled |
|---|---|---|---|
| **35** | 28 | 9 | 55.54 |

### *Decision Tree*

In [1356]:

```
d = []
for depth in range(1,10):
    decision_tree = DecisionTreeClassifier(random_state=12345, max_depth=depth)
    decision_tree.fit(X_train_upsampled, y_train_upsampled)
    y_pred = decision_tree.predict(X_valid)
    f1_DT_upsampled = round(f1_score(y_valid,y_pred) * 100, 2)
    d.append(
        {
            'max_depth': depth,
            'f1_DT_upsampled':  f1_DT_upsampled
          }
     )

best_param = pd.DataFrame(d).nlargest(1, ['f1_DT_upsampled'], keep='first')
f1_DT_upsampled = best_param['f1_DT_upsampled'].values
best_param
```

Out[1356]:

| | max_depth | f1_DT_upsampled |
|---|---|---|
| **4** | 5 | 54.94 |

### Method 3: Downsampling

In [1357]:

```
def downsample(features, target, fraction):
    features_zeros = features[target == 0]
    features_ones = features[target == 1]
    target_zeros = target[target == 0]
    target_ones = target[target == 1]

    features_downsampled = pd.concat(
        [features_zeros.sample(frac=fraction, random_state=12345)] + [features_o
nes])
    target_downsampled = pd.concat(
        [target_zeros.sample(frac=fraction, random_state=12345)] + [target_ones
])

    features_downsampled, target_downsampled = shuffle(
        features_downsampled, target_downsampled, random_state=12345)

    return features_downsampled, target_downsampled

X_train_downsampled, y_train_downsampled = downsample(X_train, y_train, 0.1)
```

### *Logistic Regression*

In [1358]:

```
d = []
for penalty in ['l1', 'l2']:
    for C in np.arange(0.01,1,0.01):
        LR = LogisticRegression(solver="liblinear", penalty=penalty, C=C, random
_state=12345)
        LR.fit(X_train_downsampled, y_train_downsampled)
        y_pred = LR.predict(X_valid)
        f1_LR_downsampled = round(f1_score(y_valid,y_pred) * 100, 2)
        d.append(
            {
                'penalty': penalty,
                'C': C,
                'f1_LR_downsampled':  f1_LR_downsampled
            }
        )
best_param = pd.DataFrame(d).nlargest(1, ['f1_LR_downsampled'], keep='first')
f1_LR_downsampled = best_param['f1_LR_downsampled'].values
best_param
```

Out[1358]:

| | penalty | C | f1_LR_downsampled |
|---|---|---|---|
| **25** | l1 | 0.26 | 41.42 |

### *Random Forest*

In [1359]:

```
d = []
for estim in range(1,51,9):
    for depth in range(1,10):
        rfc = RandomForestClassifier(random_state=12345, n_estimators=estim, max
_depth=depth)
        rfc.fit(X_train_downsampled, y_train_downsampled)
        y_pred = rfc.predict(X_valid)
        f1_rfc_downsampled = round(f1_score(y_valid,y_pred) * 100, 2)
        d.append(
            {
                'n_estimators': estim,
                'max_depth': depth,
                'f1_rfc_downsampled':  f1_rfc_downsampled
            }
        )

best_param = pd.DataFrame(d).nlargest(1, ['f1_rfc_downsampled'], keep='first')
f1_rfc_downsampled = best_param['f1_rfc_downsampled'].values
best_param
```

Out[1359]:

| | n_estimators | max_depth | f1_rfc_downsampled |
|---|---|---|---|
| **5** | 1 | 6 | 50.22 |

***Decision Tree***

In [1360]:

```
d = []
for depth in range(1,10):
    decision_tree = DecisionTreeClassifier(random_state=12345, max_depth=depth)
    decision_tree.fit(X_train_downsampled, y_train_downsampled)
    y_pred = decision_tree.predict(X_valid)
    f1_DT_downsampled = round(f1_score(y_valid,y_pred) * 100, 2)
    d.append(
        {
            'max_depth': depth,
            'f1_DT_downsampled':  f1_DT_downsampled
          }
     )

best_param = pd.DataFrame(d).nlargest(1, ['f1_DT_downsampled'], keep='first')
f1_DT_downsampled = best_param['f1_DT_downsampled'].values
best_param
```

Out[1360]:

| | max_depth | f1_DT_downsampled |
|---|---|---|
| **4** | 5 | 54.14 |

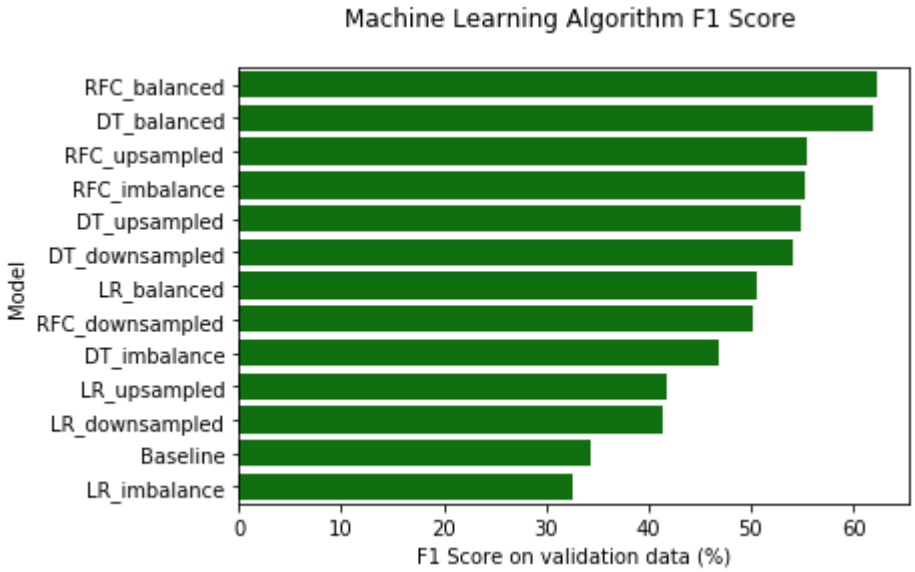
## Model selection

In [1361]:

```
models = pd.DataFrame({
    'Model': ['Baseline', 'DT_imbalance', 'LR_imbalance', 'RFC_imbalance', 'DT_b
alanced','LR_balanced',
              'RFC_balanced', 'DT_upsampled','LR_upsampled','RFC_upsampled', 'DT
_downsampled','LR_downsampled',
              'RFC_downsampled'],
    'Score': [f1_baseline, f1_DT_imbalance, f1_LR_imbalance, f1_rfc_imbalance, f
1_DT_balanced, f1_LR_balanced,
              f1_rfc_balanced, f1_DT_upsampled, f1_LR_upsampled, f1_rfc_upsample
d, f1_DT_downsampled,
              f1_LR_downsampled, f1_rfc_downsampled]})
sorted_by_score = models.sort_values(by='Score', ascending=False)
```

In [1362]:

```
sns.barplot(x='Score', y = 'Model', data = sorted_by_score, color = 'g')
plt.title('Machine Learning Algorithm F1 Score \n')
plt.xlabel('F1 Score on validation data (%)')
plt.ylabel('Model');
```




We see that the two algorithms that gave by far the best F1 score are **RFC_balanced** and **DT_balanced**. Both are using Class Weight Adjustment method to deal with class imbalance.

Random forest leverages the power of multiple decision trees. It does not rely on the feature importance given by a single decision tree. The decision tree model gives high importance to a particular set of features. But the random forest chooses features randomly during the training process, so it does not depend highly on any specific set of features. Therefore, the random forest can generalize over the data in a better way. We will choose the balanced RFC to be our final model.

## Retrain the best tuned model on the whole training set and test it on the test set

In [1363]:

```
X_train_all_scaled = sc.fit_transform(X_train_all)
```

In [1364]:

```
rfc = RandomForestClassifier(n_estimators=46, max_depth=7, random_state=12345, c
lass_weight='balanced')
rfc.fit(X_train_all_scaled, y_train_all)
y_pred = rfc.predict(X_test)

f1_rfc = round(f1_score(y_test,y_pred) * 100, 2)
f1_rfc
```

Out[1364]:

```
63.19
```

## Sanity check

The final F1 score of our model is much higher (28.82%) than the baseline F1 score that we would get if instead of classifying we simply predicted minority class target value for each new observation.

## AUC-ROC

To find how much our model differs from the random model, let's calculate the AUC-ROC value (Area Under Curve ROC) This is an evaluation metric with values in the range from 0 to 1. The AUC-ROC value for a random model is 0.5.

Unlike other metrics, it takes class "1" probabilities instead of predictions.

In [1371]:

```
probabilities_test = rfc.predict_proba(X_test)
probabilities_one_test = probabilities_test[:, 1]
```

In [1373]:

```
auc_roc = round(roc_auc_score(y_test, probabilities_one_test) * 100, 2)
auc_roc
```

Out[1373]:

```
86.66
```

The AUC-ROC value is very high, it means that our model's predictions are correct in more than 80% of cases.

The ROC curve takes the target values and the positive class probabilities, goes over different thresholds, and returns three lists: FPR values, TPR values, and the thresholds it went over.

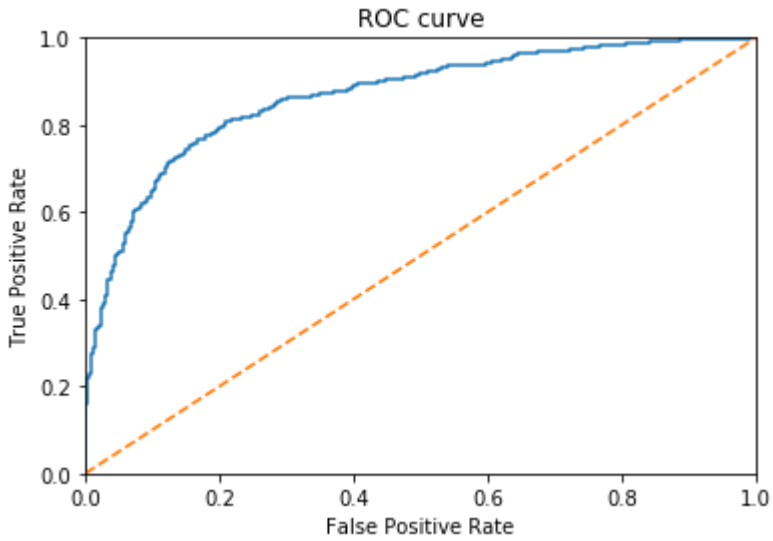
For a model that always answers randomly, the ROC curve is a diagonal line going from the lower left to the upper right. The higher the curve, the greater the TPR value and the better the model's quality.

In [1375]:

```
fpr, tpr, thresholds = roc_curve(y_test, probabilities_one_test)
```

In [1376]:

```
plt.figure()
plt.plot(fpr, tpr)
# ROC curve for random model (looks like a straight line)
plt.plot([0, 1], [0, 1], linestyle='--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.ylim([0.0, 1.0])
plt.xlim([0.0, 1.0])
plt.title('ROC curve')
plt.show()
```




We see that our final model is performing quite well - its ROC curve is much higher than the diagonal.

# Conclusion

In this project we have **developed a binary classification model that analyzes data on clients' past behavior and termination of contracts with the bank and predicts whether a customer will leave the bank soon**.

First of all, we have familiarized ourselves with the data by performing the descriptive statistics. We found missing values in the `Tenure` variable.

In the **preprocessing step** we have converted column names to lower case, dropped 3 uninformative columns, filled missing `tenure` values with the mean per `num_of_products` group, OHE encoded categorical features and checked for duplicated values.

In the following section we have performed an **exploratory data analysis** and reached the following conclusions**:

- There are no visible outliers in our data. We decided to keep the data as is and if necessary revisit this section after modeling;
- Most features' distribution is close to normal, there is only slight deviation from it (except for `messages`). We tried to correct this by applying standard scaling before modeling;
- Most clients are located in France and they are the most loyal to the bank among all countries, on average. The highest churn rate is in Germany. This features might be quite a good predictor for our model;
- Most customers are in their 30s, the distribution is positively skewed as less elder than younger people have bank accounts. Interestingly that the distribution of "exited" customers (in orange) is closer to normal than that of loyal clients;
- Most clients are male. Female clients leave the bank more often.
- We noticed that our target classes are imbalanced: there are more than twice as many observations when a client stayed with the bank than of those who left. As one way to correct it, we used the `stratify` parameter while splittig data into train and test sets. It makes a split so that the proportion of values in the sample produced will be the same as the proportion of values in the target variable.

In the next step we have tried to correct **imbalanced classes** using 3 methods:

- class weight adjustment;
- upsampling;
- downsampling.

We have also tuned each of the 3 chosen algorithms (Decision Tree, Random Forest and Logistic Regression) and searched for the best **hyperparameters** in order to select the best model.

**Random Forest model using Class Weight Adjustment method** showed the highest score (62.33). Then we have retrained this model on the whole training set (including validation set) and tested it with the test set that our model didn't see before. We have reached **63.19% F1 score on the test set**.

Next, we have checked our model for **sanity** by comparing the final score to the baseline F1 score. The final F1 score of our model is much higher (28.82%) than the baseline F1 score that we would get if instead of classifying we simply predicted minority class target value for each new observation.

Finally, we have calculated the AUC-ROC value and it turned out to be **86.66%**, which means that our model's predictions are correct in more than 80% of cases. We have also plotted the ROC curve. It visualized well the fact that our final model is performing quite well - its ROC curve is much higher than the diagonal.