Home Credit Default Risk

1. Business/Real-world Problem

1.1. Problem Statement

- The problem is that there are a lot of people who apply for loans in Banks and similar financial institutions whereas only a few of them get approved. This is primarily because of insufficient or non-existent credit histories of the applicant, whereas this population is taken advantage of by untrustworthy lenders.
- In order to make sure that these applicants have a positive loan taking experience, Home Credit uses a lot of data (including telco data and transactional data) to predict the applicants' loan repayment abilities.
- Improving on this overall process basically ensures that the clients capable of loan repayment do not have their applications rejected.

1.2. Source/Useful Links

Our Task is that basically, given a 'SK_ID_CURR' (Applicant's Primary Key Information), we have to determine whether the applicant will be capable of loan repayment or not.

- Source: https://www.kaggle.com/c/home-credit-default-risk/overview

1.3. Real-world/Business objectives and constraints.

No Strict Latency Constraints

- This is not exactly a Low-Latency Requirement because in a Low Latency Requirement Problem, such as for Internet Companies, low latency refers to a few Milliseconds.
- Our algorithm over here can take sometime to run in order to ensure high accuracy in predicting repayment capabilities. The Bank/Financial Institution doesn't need to deliver the results in a very quick time.

High Misclassification Cost

- This is a very important real world metric that needs to be considered because our cost of misclassification can be very high.
- If a loan applicant who is not capable of loan repayment is classified as capable and he/she is granted a loan, and in case he/she is unable to repay the loan, the bank or financial institution runs into delinquencies and may suffer losses, which could even have to be Written Off.
- Similarly, if a capable applicant is classified as non-capable, the person has his/her application rejected and the Bank loses out on a customer, which affects their profits.

Interpretability is Important

- This means that we should be able to generate the Probability Estimates, of an applicant being capable or not capable, rather than strictly classifying the applicant as either of them.
- Eg: If probability is 0.5 for an applicant's capability and 0.9 in the other case, we can very well conclude that we are much more sure of the capability when the value is 0.9 (and classified as 1) rather than when the value is 0.5 (and then classified as 1).

2. Machine Learning Problem

2.1. Data

2.1.1. Data Overview

Source : https://www.kaggle.com/c/home-credit-default-risk/data

application_{train|test}.csv

- This is the main table, broken into two files for Train (with TARGET) and Test (without TARGET).
- Static data for all applications. One row represents one loan in our data sample.

bureau.csv

- All client's previous credits provided by other financial institutions that were reported to Credit Bureau (for clients who have a loan in our sample).
- For every loan in our sample, there are as many rows as number of credits the client had in Credit Bureau before the application date.

bureau_balance.csv

- Monthly balances of previous credits in Credit Bureau.
- This table has one row for each month of history of every previous credit reported to Credit Bureau – i.e the table has (#loans in sample *# of relative previous credits* # of months where we have some history observable for the previous credits) rows.

POS_CASH_balance.csv

- Monthly balance snapshots of previous POS (point of sales) and cash loans that the applicant had with Home Credit.

- This table has one row for each month of history of every previous credit in Home Credit (consumer credit and cash loans) related to loans in our sample – i.e. the table has (#loans in sample # *of relative previous credits* # of months in which we have some history observable for the previous credits) rows.

credit_card_balance.csv

- Monthly balance snapshots of previous credit cards that the applicant has with Home Credit.
- This table has one row for each month of history of every previous credit in Home Credit (consumer credit and cash loans) related to loans in our sample – i.e. the table has (#loans in sample # *of relative previous credit cards* # of months where we have some history observable for the previous credit card) rows.

previous_application.csv

- All previous applications for Home Credit loans of clients who have loans in our sample.
- There is one row for each previous application related to loans in our data sample.

installments_payments.csv

- Repayment history for the previously disbursed credits in Home Credit related to the loans in our sample.
- There is one row for every payment that was made plus one row each for missed payment.
- One row is equivalent to one payment of one installment OR one installment corresponding to one payment of one previous Home Credit credit related to loans in our sample.

HomeCredit_columns_description.csv

- This file contains descriptions for the columns in the various data files.

Database Schema

2.2. Mapping the real-world problem to an ML problem

2.2.1. Type of Machine Learning Problem

This is a Simple Binary Classification Problem because 0 means that an applicant is capable of loan repayment whereas 1 means that an applicant is not capable of loan repayment.

2.2.2. Performance Metric

Source: https://www.kaggle.com/c/home-credit-default-risk/overview/evaluation

*** AUC or Area Under Curve (Because we have a Binary Classification Problem)**

*** Confusion matrix**

- We will build a Confusion Matrix for Binary Classes. Along with this, we also have a look at the corresponding Precision as well as Recall Matrices.

2.3. Useful Blogs, Videos and Reference Papers

- https://www.kaggle.com/rinnqd/reduce-memory-usage
- Seventh place solution in Kaggle Competition : https://www.kaggle.com/jsaguiar/lightgbm-7th-place-solution
- https://www.linkedin.com/pulse/winning-9th-place-kaggles-biggest-competition-yet-home-levinson/
- https://www.kaggle.com/c/home-credit-default-risk/discussion/64593
- https://www.kaggle.com/c/home-credit-default-risk/discussion/64821
- http://mlexplained.com/2018/01/05/lightgbm-and-xgboost-explained/

- https://www.analyticsvidhya.com/blog/2017/06/which-algorithm-takes-the-crown-light-gbm-vs-xgboost/
- https://www.youtube.com/watch?v=OAl6eAyP-yo
- https://www.youtube.com/watch?v=4jRBRDbJemM

3. Importing Necessary Libraries

In [5]:
```
import warnings
warnings.filterwarnings("ignore")

import pandas as pd
import matplotlib.pyplot as plt
import re
import time
import numpy as np
import gc
import xgboost as xgb
import lightgbm as lgb
import seaborn as sns
import math
import pickle
import os

from lightgbm import LGBMClassifier
from sklearn.metrics import roc_auc_score
from scipy.stats import randint as sp_randint
from sklearn.model_selection import KFold, StratifiedKFold
from prettytable import PrettyTable
from sklearn.metrics import roc_curve,auc
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import normalize
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import f_classif
from sklearn.metrics import confusion_matrix
from sklearn.metrics.classification import accuracy_score, log_loss
from sklearn.linear_model import SGDClassifier
```

```
from collections import Counter
from scipy.sparse import hstack
from sklearn.calibration import CalibratedClassifierCV
from sklearn.model_selection import train_test_split
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import RandomizedSearchCV
from sklearn.preprocessing import StandardScaler
from sklearn.ensemble import RandomForestClassifier
from bayes_opt import BayesianOptimization
from sklearn import model_selection
from sklearn.linear_model import LogisticRegression
from datetime import datetime
```

4. Application Train/Test Datasets

Function to Reduce the Memory Usage of a DataFrame

In [6]:
```
# Refer :- https://www.kaggle.com/rinnqd/reduce-memory-usage

def reduce_memory_usage(df):

    start_mem = df.memory_usage().sum() / 1024**2
    print('Memory usage of dataframe is {:.2f} MB'.format(start_mem))

    for col in df.columns:
        col_type = df[col].dtype

        if col_type != object:
            c_min = df[col].min()
            c_max = df[col].max()
            if str(col_type)[:3] == 'int':
                if c_min > np.iinfo(np.int8).min and c_max < np.iinfo(n
p.int8).max:
                    df[col] = df[col].astype(np.int8)
                elif c_min > np.iinfo(np.int16).min and c_max < np.iinf
o(np.int16).max:
```

```
                    df[col] = df[col].astype(np.int16)
                elif c_min > np.iinfo(np.int32).min and c_max < np.iinf
o(np.int32).max:
                    df[col] = df[col].astype(np.int32)
                elif c_min > np.iinfo(np.int64).min and c_max < np.iinf
o(np.int64).max:
                    df[col] = df[col].astype(np.int64)
            else:
                if c_min > np.finfo(np.float16).min and c_max < np.finf
o(np.float16).max:
                    df[col] = df[col].astype(np.float16)
                elif c_min > np.finfo(np.float32).min and c_max < np.fi
nfo(np.float32).max:
                    df[col] = df[col].astype(np.float32)
                else:
                    df[col] = df[col].astype(np.float64)

    end_mem = df.memory_usage().sum() / 1024**2
    print('Memory usage after optimization is: {:.2f} MB'.format(end_me
m))
    print('Decreased by {:.1f}%'.format(100 * (start_mem - end_mem) / s
tart_mem))

    return df
```

4.1. Basic Overview of the Train Data

In [3]:
```
train_data = reduce_memory_usage(pd.read_csv('home-credit-default-risk/
application_train.csv'))
print('Number of data points : ', train_data.shape[0])
print('Number of features : ', train_data.shape[1])
train_data.head()
```

```
Memory usage of dataframe is 286.23 MB
Memory usage after optimization is: 92.38 MB
Decreased by 67.7%
Number of data points :  307511
Number of features :  122
```

Out[3]:

	SK_ID_CURR	TARGET	NAME_CONTRACT_TYPE	CODE_GENDER	FLAG_OWN_CAR	FLAG_OV
0	100002	1	Cash loans	M	N	
1	100003	0	Cash loans	F	N	
2	100004	0	Revolving loans	M	Y	
3	100006	0	Cash loans	F	N	
4	100007	0	Cash loans	M	N	

5 rows × 122 columns

In [4]:
```
print('Features : ', train_data.columns.values)
```

```
Features :  ['SK_ID_CURR' 'TARGET' 'NAME_CONTRACT_TYPE' 'CODE_GENDER'
'FLAG_OWN_CAR'
 'FLAG_OWN_REALTY' 'CNT_CHILDREN' 'AMT_INCOME_TOTAL' 'AMT_CREDIT'
 'AMT_ANNUITY' 'AMT_GOODS_PRICE' 'NAME_TYPE_SUITE' 'NAME_INCOME_TYPE'
 'NAME_EDUCATION_TYPE' 'NAME_FAMILY_STATUS' 'NAME_HOUSING_TYPE'
 'REGION_POPULATION_RELATIVE' 'DAYS_BIRTH' 'DAYS_EMPLOYED'
 'DAYS_REGISTRATION' 'DAYS_ID_PUBLISH' 'OWN_CAR_AGE' 'FLAG_MOBIL'
 'FLAG_EMP_PHONE' 'FLAG_WORK_PHONE' 'FLAG_CONT_MOBILE' 'FLAG_PHONE'
 'FLAG_EMAIL' 'OCCUPATION_TYPE' 'CNT_FAM_MEMBERS' 'REGION_RATING_CLIEN
T'
 'REGION_RATING_CLIENT_W_CITY' 'WEEKDAY_APPR_PROCESS_START'
 'HOUR_APPR_PROCESS_START' 'REG_REGION_NOT_LIVE_REGION'
 'REG_REGION_NOT_WORK_REGION' 'LIVE_REGION_NOT_WORK_REGION'
 'REG_CITY_NOT_LIVE_CITY' 'REG_CITY_NOT_WORK_CITY'
 'LIVE_CITY_NOT_WORK_CITY' 'ORGANIZATION_TYPE' 'EXT_SOURCE_1'
 'EXT_SOURCE_2' 'EXT_SOURCE_3' 'APARTMENTS_AVG' 'BASEMENTAREA_AVG'
 'YEARS_BEGINEXPLUATATION_AVG' 'YEARS_BUILD_AVG' 'COMMONAREA_AVG'
 'ELEVATORS_AVG' 'ENTRANCES_AVG' 'FLOORSMAX_AVG' 'FLOORSMIN_AVG'
 'LANDAREA_AVG' 'LIVINGAPARTMENTS_AVG' 'LIVINGAREA_AVG'
 'NONLIVINGAPARTMENTS_AVG' 'NONLIVINGAREA_AVG' 'APARTMENTS_MODE'
 'BASEMENTAREA_MODE' 'YEARS_BEGINEXPLUATATION_MODE' 'YEARS_BUILD_MODE'
 'COMMONAREA_MODE' 'ELEVATORS_MODE' 'ENTRANCES_MODE' 'FLOORSMAX_MODE'
 'FLOORSMIN_MODE' 'LANDAREA_MODE' 'LIVINGAPARTMENTS_MODE'
```

```
 'LIVINGAREA_MODE' 'NONLIVINGAPARTMENTS_MODE' 'NONLIVINGAREA_MODE'
 'APARTMENTS_MEDI' 'BASEMENTAREA_MEDI' 'YEARS_BEGINEXPLUATATION_MEDI'
 'YEARS_BUILD_MEDI' 'COMMONAREA_MEDI' 'ELEVATORS_MEDI' 'ENTRANCES_MEDI'
 'FLOORSMAX_MEDI' 'FLOORSMIN_MEDI' 'LANDAREA_MEDI' 'LIVINGAPARTMENTS_ME
DI'
 'LIVINGAREA_MEDI' 'NONLIVINGAPARTMENTS_MEDI' 'NONLIVINGAREA_MEDI'
 'FONDKAPREMONT_MODE' 'HOUSETYPE_MODE' 'TOTALAREA_MODE'
 'WALLSMATERIAL_MODE' 'EMERGENCYSTATE_MODE' 'OBS_30_CNT_SOCIAL_CIRCLE'
 'DEF_30_CNT_SOCIAL_CIRCLE' 'OBS_60_CNT_SOCIAL_CIRCLE'
 'DEF_60_CNT_SOCIAL_CIRCLE' 'DAYS_LAST_PHONE_CHANGE' 'FLAG_DOCUMENT_2'
 'FLAG_DOCUMENT_3' 'FLAG_DOCUMENT_4' 'FLAG_DOCUMENT_5' 'FLAG_DOCUMENT_
6'
 'FLAG_DOCUMENT_7' 'FLAG_DOCUMENT_8' 'FLAG_DOCUMENT_9' 'FLAG_DOCUMENT_1
0'
 'FLAG_DOCUMENT_11' 'FLAG_DOCUMENT_12' 'FLAG_DOCUMENT_13'
 'FLAG_DOCUMENT_14' 'FLAG_DOCUMENT_15' 'FLAG_DOCUMENT_16'
 'FLAG_DOCUMENT_17' 'FLAG_DOCUMENT_18' 'FLAG_DOCUMENT_19'
 'FLAG_DOCUMENT_20' 'FLAG_DOCUMENT_21' 'AMT_REQ_CREDIT_BUREAU_HOUR'
 'AMT_REQ_CREDIT_BUREAU_DAY' 'AMT_REQ_CREDIT_BUREAU_WEEK'
 'AMT_REQ_CREDIT_BUREAU_MON' 'AMT_REQ_CREDIT_BUREAU_QRT'
 'AMT_REQ_CREDIT_BUREAU_YEAR']
```

4.2. Basic Overview of the Test Data

In [5]:
```
test_data = reduce_memory_usage(pd.read_csv('home-credit-default-risk/a
pplication_test.csv'))
print('Number of data points : ', test_data.shape[0])
print('Number of features : ', test_data.shape[1])
test_data.head()
```

```
Memory usage of dataframe is 45.00 MB
Memory usage after optimization is: 14.60 MB
Decreased by 67.6%
Number of data points :  48744
Number of features :  121
```

Out[5]:

SK_ID_CURR	NAME_CONTRACT_TYPE	CODE_GENDER	FLAG_OWN_CAR	FLAG_OWN_REALT

	SK_ID_CURR	NAME_CONTRACT_TYPE	CODE_GENDER	FLAG_OWN_CAR	FLAG_OWN_REALT
0	100001	Cash loans	F	N	
1	100005	Cash loans	M	N	
2	100013	Cash loans	M	Y	
3	100028	Cash loans	F	N	
4	100038	Cash loans	M	Y	

5 rows × 121 columns

Observations

- The Test Data has all the features same as the Train data except the 'Target' column.

4.3 Application Train Data Analysis

4.3.1 Univariate Analysis : Target

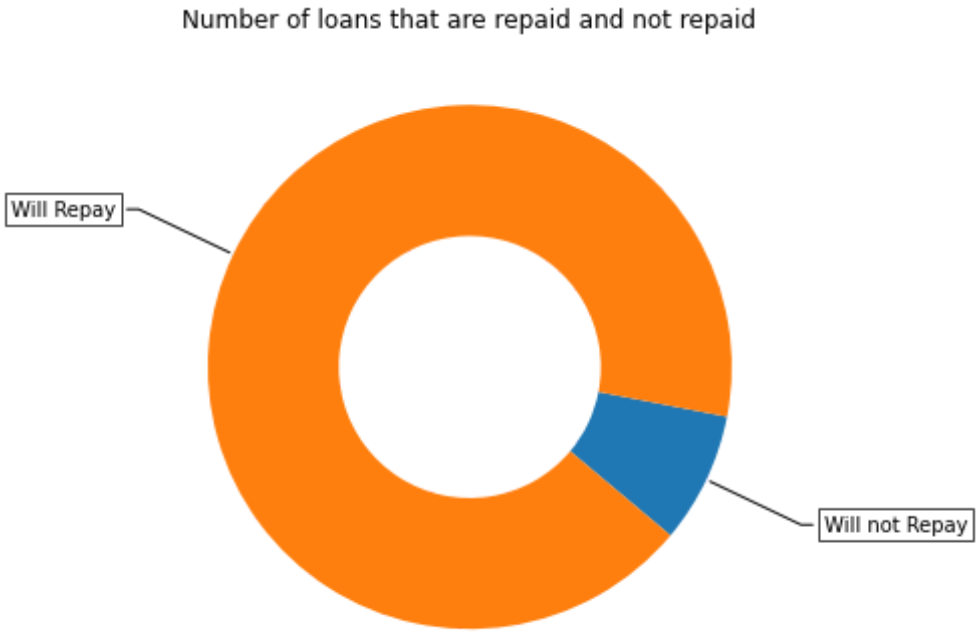
In [6]:
```
train_data['TARGET'].value_counts()
```

Out[6]:
```
0    282686
1     24825
Name: TARGET, dtype: int64
```

In [7]:
```
# Refer :- https://matplotlib.org/gallery/pie_and_polar_charts/pie_and_donut_labels.html#sphx-glr-gallery-pie-and-polar-charts-pie-and-donut-labels-py

y_value_counts = train_data['TARGET'].value_counts()
print("Number of customers who will not repay the loan on time: ", y_value_counts[1], ", (", (y_value_counts[1]/(y_value_counts[1]+y_value_cou
```

```
nts[0]))*100,"%)")
print("Number of customers who will repay the loan on time: ", y_value_
counts[0], ", (", (y_value_counts[0]/(y_value_counts[1]+y_value_counts[
0]))*100,"%)")

fig, ax = plt.subplots(figsize=(6, 6), subplot_kw=dict(aspect="equal"))
recipe = ["Will not Repay", "Will Repay"]

data = [y_value_counts[1], y_value_counts[0]]

wedges, texts = ax.pie(data, wedgeprops=dict(width=0.5),\
                       startangle=-40)

bbox_props = dict(boxstyle="square,pad=0.3", fc="w", ec="k", lw=0.72)
kw = dict(xycoords='data', textcoords='data', arrowprops=dict(arrowstyl
e="-"),
          bbox=bbox_props, zorder=0, va="center")

for i, p in enumerate(wedges):
    ang = (p.theta2 - p.theta1)/2. + p.theta1
    y = np.sin(np.deg2rad(ang))
    x = np.cos(np.deg2rad(ang))
    horizontalalignment = {-1: "right", 1: "left"}[int(np.sign(x))]
    connectionstyle = "angle,angleA=0,angleB={}".format(ang)
    kw["arrowprops"].update({"connectionstyle": connectionstyle})
    ax.annotate(recipe[i], xy=(x, y), xytext=(1.35*np.sign(x), 1.4*y),
                horizontalalignment=horizontalalignment, **kw)

ax.set_title("Number of loans that are repaid and not repaid")

plt.show()
```

```
Number of customers who will not repay the loan on time:  24825 , ( 8.0
72881945686495 %)
Number of customers who will repay the loan on time:  282686 , ( 91.927
11805431351 %)
```



Function to plot the Stacked Bar Plot

In [8]:
```
#stacked bar plots matplotlib: https://matplotlib.org/gallery/lines_bars_and_markers/bar_stacked.html
def stack_plot(data, xtick, col2='TARGET', col3='total'):
    ind = np.arange(data.shape[0])

    if len(data[xtick].unique())<5:
        plt.figure(figsize=(5,5))
```

```
    elif len(data[xtick].unique())>5 & len(data[xtick].unique())<10:
        plt.figure(figsize=(7,7))
    else:
        plt.figure(figsize=(15,15))
    p1 = plt.bar(ind, data[col3].values)
    p2 = plt.bar(ind, data[col2].values)

    plt.ylabel('Loans')
    plt.title('Number of loans aproved vs rejected')
    plt.xticks(ticks=ind,rotation=90,labels= list(data[xtick].values))
    plt.legend((p1[0], p2[0]), ('capable', 'not capable'))
    plt.show()
```

Function to plot the Univariate Bar Plot

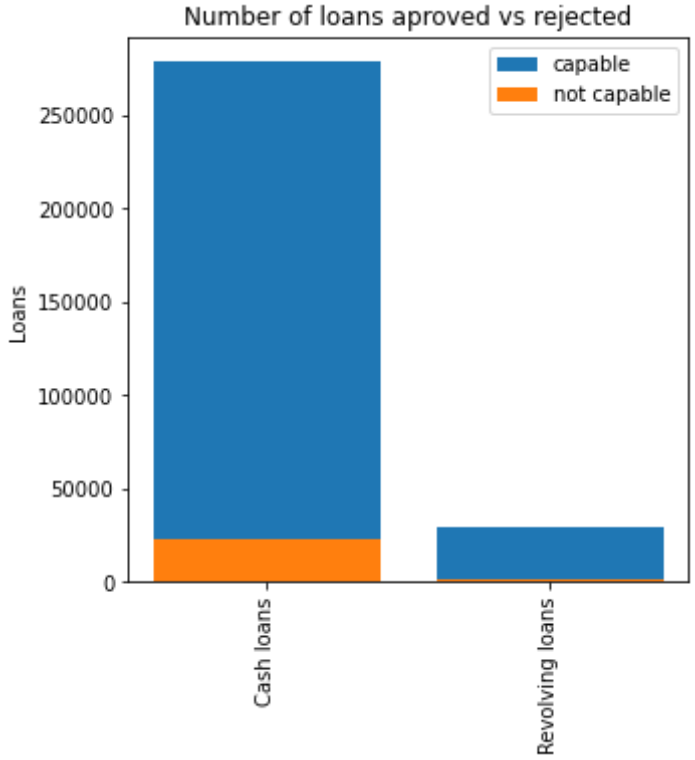
In [9]:
```
def univariate_barplots(data, col1, col2='TARGET', top=False):
    # Count number of zeros in dataframe python: https://stackoverflow.
com/a/51540521/4084039
    temp = pd.DataFrame(train_data.groupby(col1)[col2].agg(lambda x: x.
eq(1).sum())).reset_index()

    # Pandas dataframe grouby count: https://stackoverflow.com/a/193855
91/4084039
    temp['total'] = pd.DataFrame(train_data.groupby(col1)[col2].agg(tot
al='count')).reset_index()['total']
    temp['Avg'] = pd.DataFrame(train_data.groupby(col1)[col2].agg(Avg=
'mean')).reset_index()['Avg']

    temp.sort_values(by=['total'],inplace=True, ascending=False)

    if top:
        temp = temp[0:top]

    stack_plot(temp, xtick=col1, col2=col2, col3='total')
    print(temp.head(5))
    print("="*50)
    print(temp.tail(5))
```

4.3.2 Univariate Analysis : Name_Contract_Type

In [10]:
```
train_data['NAME_CONTRACT_TYPE'].value_counts()
```

Out[10]:
```
Cash loans         278232
Revolving loans     29279
Name: NAME_CONTRACT_TYPE, dtype: int64
```

In [11]:
```
univariate_barplots(train_data, 'NAME_CONTRACT_TYPE', 'TARGET', False)
```




```
  NAME_CONTRACT_TYPE  TARGET   total       Avg
0         Cash loans   23221  278232  0.083459
1    Revolving loans    1604   29279  0.054783
==================================================
```

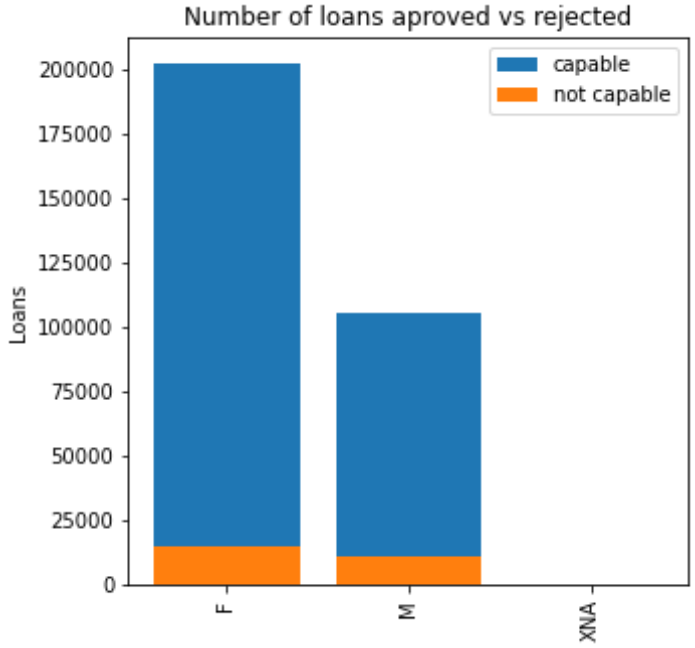
```
  NAME_CONTRACT_TYPE  TARGET   total       Avg
0         Cash loans   23221  278232  0.083459
1    Revolving loans    1604   29279  0.054783
```

Observations :

- Most of the people are taking loans in the form of cash loans instead of revolving loans such as credit cards.

4.3.3 Univariate Analysis : Code_Gender

In [12]:
```
univariate_barplots(train_data, 'CODE_GENDER', 'TARGET', False)
```




```
  CODE_GENDER  TARGET   total       Avg
0           F   14170  202448  0.069993
1           M   10655  105059  0.101419
```

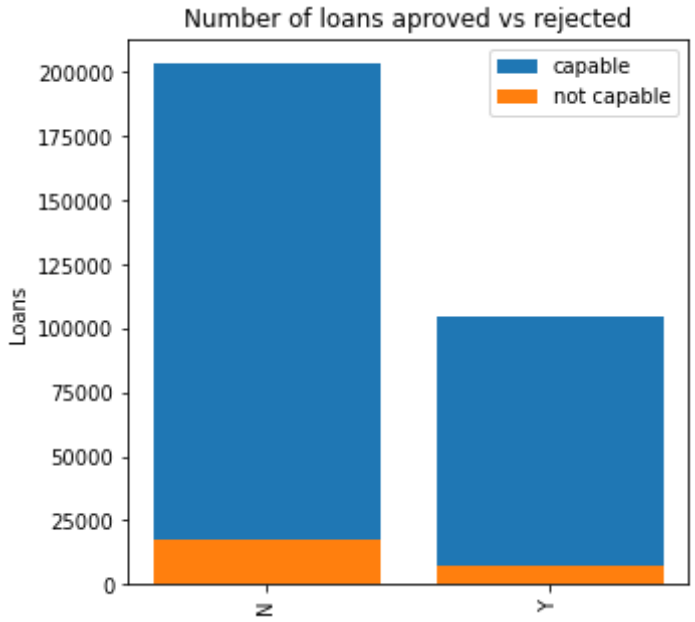
```
2          XNA       0       4  0.000000
================================================
  CODE_GENDER  TARGET   total       Avg
0           F   14170  202448  0.069993
1           M   10655  105059  0.101419
2         XNA       0       4  0.000000
```

Observations :

- The interesting part over here is the fact that Women took much more number of loans as compared to Men : Whereas Women took a total of 202K+ loans, Men only took 105K+ loans.
- However, at the same time, Men are slightly more capable of repaying the loan as compared to Women. Whereas Men are able to repay their loans in 10% of the cases, Women are only able to repay in 7% of the cases.
- There are 4 entries where Gender='XNA'. Since this is not providing us with much information, we can remove these entries later on.

4.3.4 Univariate Analysis : Flag_Own_Car

In [13]:
```
univariate_barplots(train_data, 'FLAG_OWN_CAR', 'TARGET', False)
```



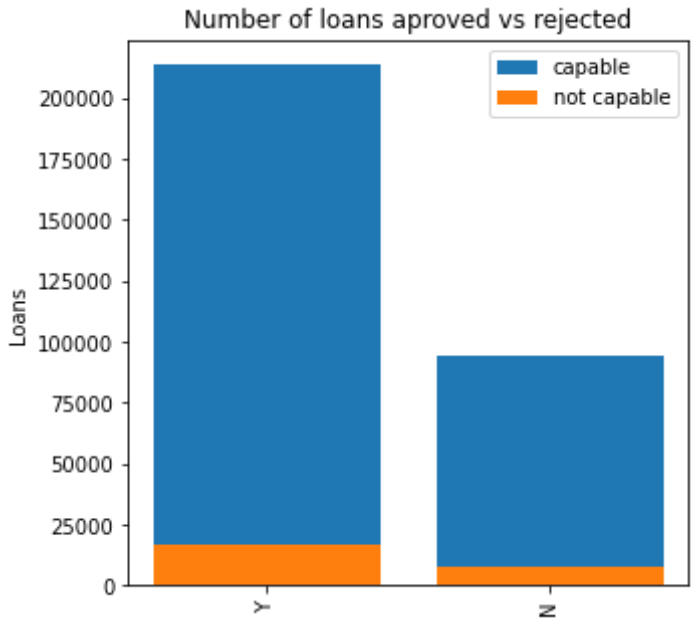

```
  FLAG_OWN_CAR  TARGET   total       Avg
0            N   17249  202924  0.085002
1            Y    7576  104587  0.072437
================================================
  FLAG_OWN_CAR  TARGET   total       Avg
0            N   17249  202924  0.085002
1            Y    7576  104587  0.072437
```

Observations :

- Most of the applicants for loans do not own a car.
- However, there is not much difference in the loan repayment status for the customer based on this information (8.5% and 7.2% respectively). We can conclude that this feature is not very useful.

4.3.5 Univariate Analysis : Flag_Own_Realty

In [14]:
```
univariate_barplots(train_data, 'FLAG_OWN_REALTY', 'TARGET', False)
```



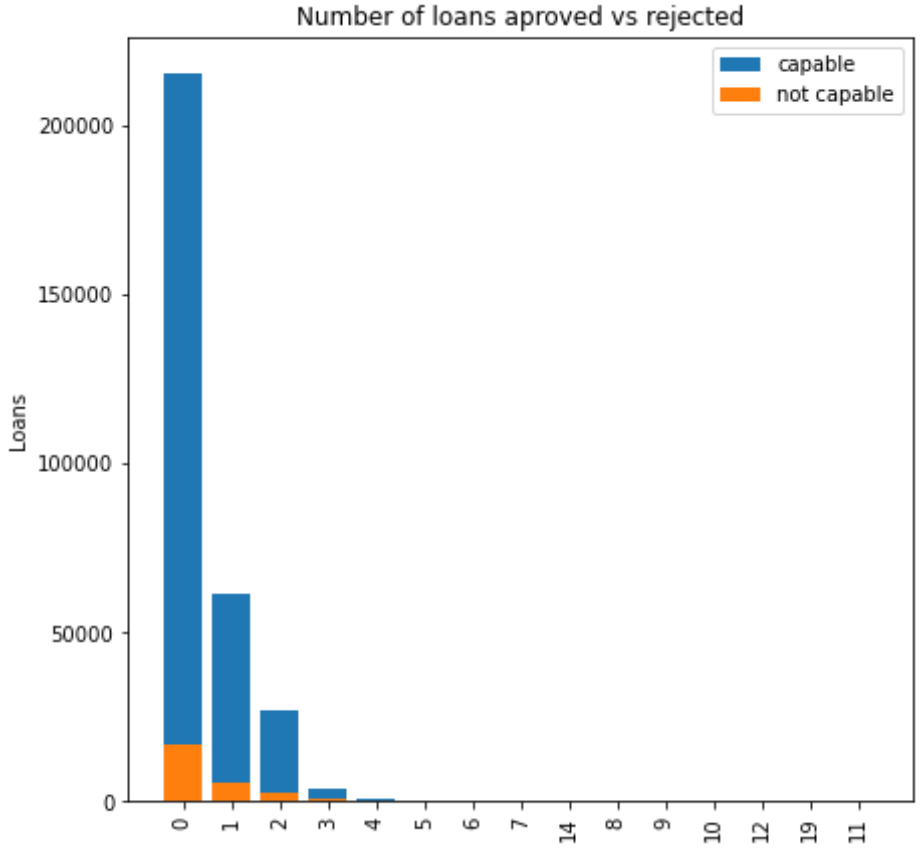

```
  FLAG_OWN_REALTY  TARGET   total       Avg
1               Y   16983  213312  0.079616
0               N    7842   94199  0.083249
==================================================
  FLAG_OWN_REALTY  TARGET   total       Avg
1               Y   16983  213312  0.079616
0               N    7842   94199  0.083249
```

Observations :

- Most of the applicants for loans own a flat/house, which is a little surprising.
- However, again, there is not much difference in the loan repayment status for the customer based on this information (7.9% and 8.3% respectively). We can conclude that this feature is not very useful.

4.3.6 Univariate Analysis : Cnt_Children

In [15]:
```
univariate_barplots(train_data, 'CNT_CHILDREN', 'TARGET', False)
```



	CNT_CHILDREN	TARGET	total	Avg
0	0	16609	215371	0.077118
1	1	5454	61119	0.089236
2	2	2333	26749	0.087218
3	3	358	3717	0.096314
4	4	55	429	0.128205

==========

	CNT_CHILDREN	TARGET	total	Avg
9	9	2	2	1.0

```
10              10       0      2  0.0
12              12       0      2  0.0
14              19       0      2  0.0
11              11       1      1  1.0
```

Observations :

- The applicants having no children take considerably higher number of loans.
- However, again, there is not much difference in the loan repayment status for the customer based on this information. We can conclude that this feature is not very useful.

4.3.7 Univariate Analysis : Amt_Income_Total

In [16]:
```
# https://stackoverflow.com/questions/22407798/how-to-reset-a-dataframes-indexes-for-all-groups-in-one-step
income_data = train_data.groupby('SK_ID_CURR').agg({'AMT_INCOME_TOTAL': 'mean'}).reset_index()
income_data.head(2)
```

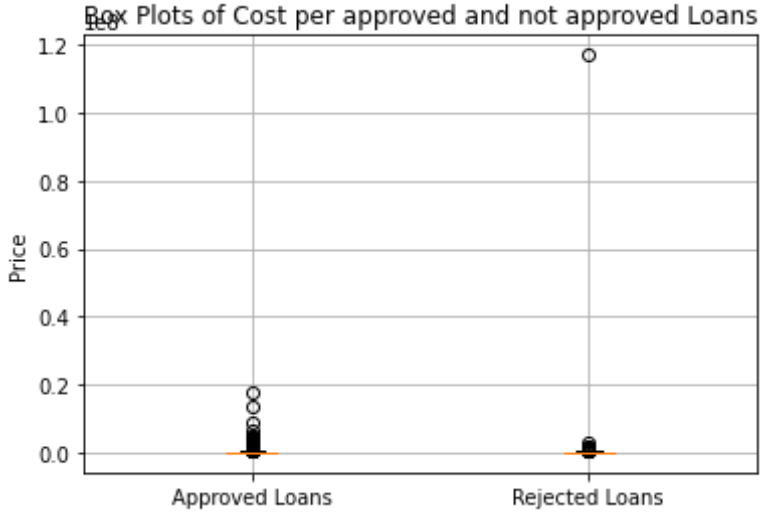
Out[16]:

	SK_ID_CURR	AMT_INCOME_TOTAL
0	100002	202500.0
1	100003	270000.0

In [17]:
```
income_data_final = pd.merge(train_data, income_data, on='SK_ID_CURR', how='left')

approved_income = income_data_final[income_data_final['TARGET']==0]['AMT_INCOME_TOTAL_x'].values
rejected_income = income_data_final[income_data_final['TARGET']==1]['AMT_INCOME_TOTAL_x'].values
```

In [18]:
```
# https://glowingpython.blogspot.com/2012/09/boxplot-with-matplotlib.html
```

```
plt.boxplot([approved_income, rejected_income])
plt.title('Box Plots of Cost per approved and not approved Loans')
plt.xticks([1,2],('Approved Loans','Rejected Loans'))
plt.ylabel('Price')
plt.grid()
plt.show()
```



In [19]:
```
# http://zetcode.com/python/prettytable/
from prettytable import PrettyTable

#If you get a ModuleNotFoundError error , install prettytable using: pi
p3 install prettytable

x = PrettyTable()
x.field_names = ["Percentile", "Approved Loans", "Not Approved Loans"]

for i in range(0,101,5):
    x.add_row([i,np.round(np.percentile(approved_income,i), 3),\
               np.round(np.percentile(rejected_income,i), 3)])
print(x)
```

```
+------------+----------------+--------------------+
| Percentile | Approved Loans | Not Approved Loans |
```

0	25650.0	25650.0
5	67500.0	67500.0
10	81000.0	81000.0
15	90000.0	90000.0
20	99000.0	99000.0
25	112500.0	112500.0
30	112500.0	112500.0
35	126000.0	120873.6
40	135000.0	135000.0
45	135000.0	135000.0
50	148500.0	135000.0
55	157500.0	157500.0
60	166500.0	157500.0
65	180000.0	175500.0
70	185400.0	180000.0
75	202500.0	202500.0
80	225000.0	202500.0
85	243000.0	225000.0
90	270000.0	256500.0
95	337500.0	315000.0
100	18000090.0	117000000.0

Observations :

- We can see over here that till the 45th percentile, both the Approved as well as Non-Approved loans have mostly the same value.
- However, as we go higher, as the customer's income increases, so do the chances of his loan getting approved.

4.3.8 Univariate Analysis : Amt_Credit

In [20]:
```
for i in train_data.groupby('SK_ID_CURR',as_index=False).size():
    if i>1:
```

```
        print(i)
#This means that each row in the train_data has a unique SK_ID_CURR
```

In [21]:
```
train_data[train_data['SK_ID_CURR']==100002]
```

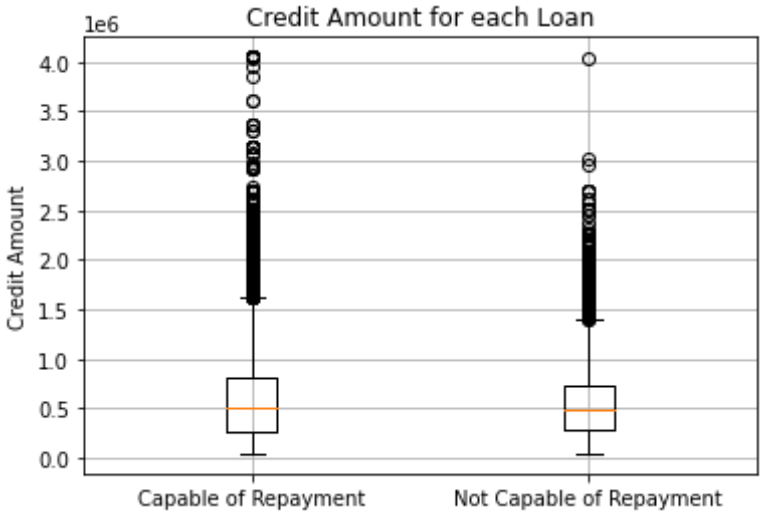
Out[21]:

	SK_ID_CURR	TARGET	NAME_CONTRACT_TYPE	CODE_GENDER	FLAG_OWN_CAR	FLAG_OV
0	100002	1	Cash loans	M	N	

1 rows × 122 columns

In [22]:
```
approved_loan_credit = train_data[train_data['TARGET']==0]['AMT_CREDIT'
].values
rejected_loan_credit = train_data[train_data['TARGET']==1]['AMT_CREDIT'
].values
```

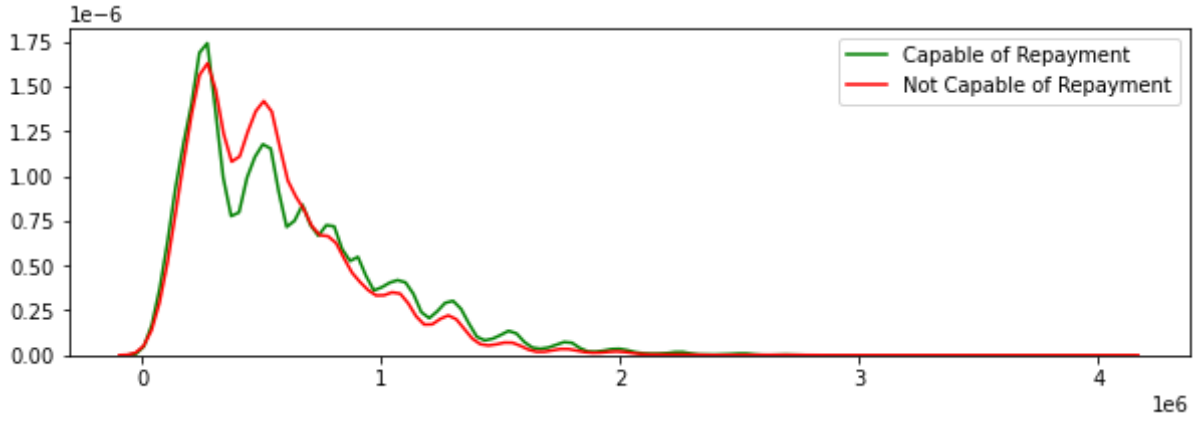
In [23]:
```
# https://glowingpython.blogspot.com/2012/09/boxplot-with-matplotlib.ht
ml
plt.boxplot([approved_loan_credit, rejected_loan_credit])
plt.title('Credit Amount for each Loan')
plt.xticks([1,2],('Capable of Repayment','Not Capable of Repayment'))
plt.ylabel('Credit Amount')
plt.grid()
plt.show()
```



Observations :

- We can see from the Boxplot above that the Median Value of the Credit Amount of the Customers who are capable of loan repayment is slightly larger than the Median Value of Customers who are not capable of repayment.
- This basically means that the customers with higher credit amount have a slightly higher chances of being capable of loan repayment than customers with lower credit amount.

In [24]:
```
plt.figure(figsize=(10,3))
sns.distplot(approved_loan_credit,hist=False,label="Capable of Repayment", color='green')
sns.distplot(rejected_loan_credit,hist=False,label="Not Capable of Repayment", color='red')
plt.legend()
plt.show()
```



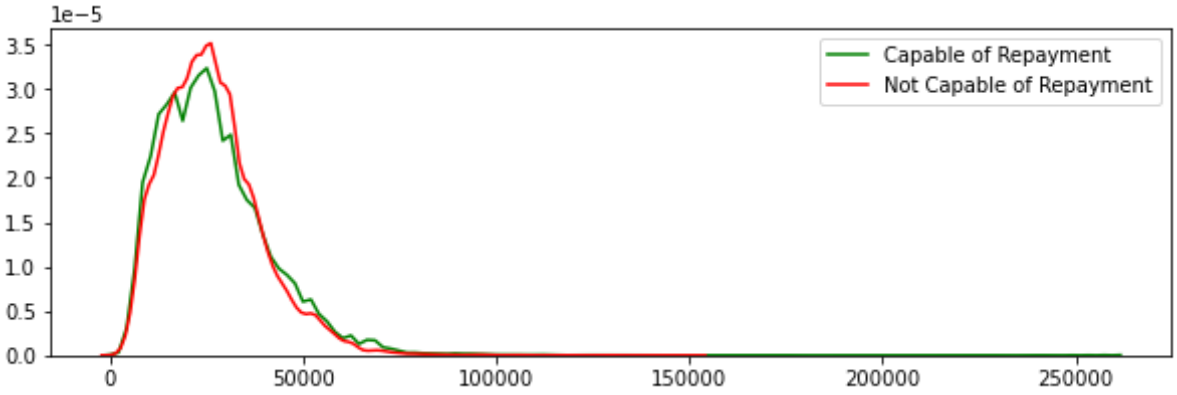
Observations :

- We can observe from above that the Credit Amount for most of the loans taken is less than 10 lakhs.

4.3.9 Univariate Analysis : Amt_Annuity

In [25]:
```
capable_loan_annuity = train_data[train_data['TARGET']==0]['AMT_ANNUIT
Y'].values
not_capable_loan_annuity = train_data[train_data['TARGET']==1]['AMT_ANN
UITY'].values
```

In [26]:
```
plt.figure(figsize=(10,3))
sns.distplot(capable_loan_annuity,hist=False,label="Capable of Repaymen
t", color='green')
sns.distplot(not_capable_loan_annuity,hist=False,label="Not Capable of
 Repayment", color='red')
plt.legend()
plt.show()
```



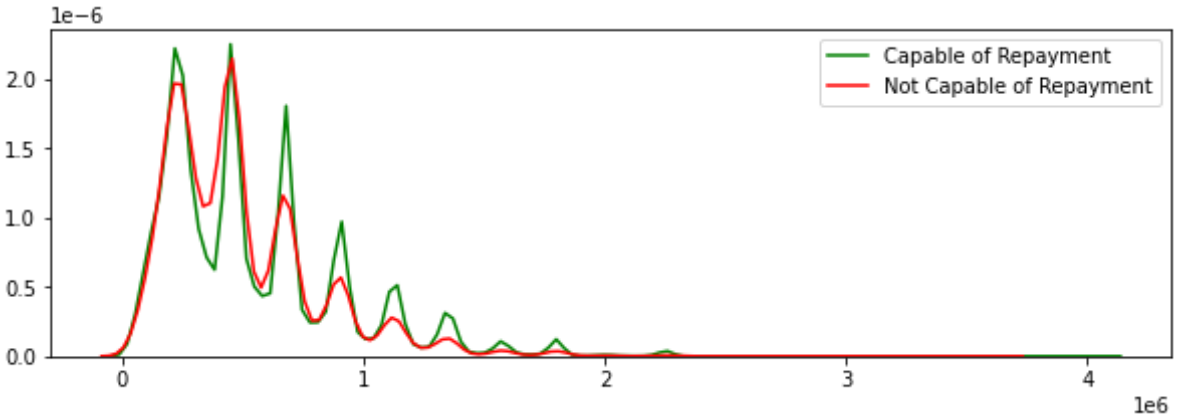
Observations :

- Most people pay annuity below Rs. 50K for the loans.

4.3.10 Univariate Analysis : Amt_Goods_Price

In [27]:
```
capable_loan_goods_price = train_data[train_data['TARGET']==0]['AMT_GOODS_PRICE'].values
not_capable_loan_goods_price = train_data[train_data['TARGET']==1]['AMT_GOODS_PRICE'].values
```

In [28]:
```
plt.figure(figsize=(10,3))
sns.distplot(capable_loan_goods_price,hist=False,label="Capable of Repayment", color='green')
sns.distplot(not_capable_loan_goods_price,hist=False,label="Not Capable of Repayment", color='red')
plt.legend()
plt.show()
```



Observations :

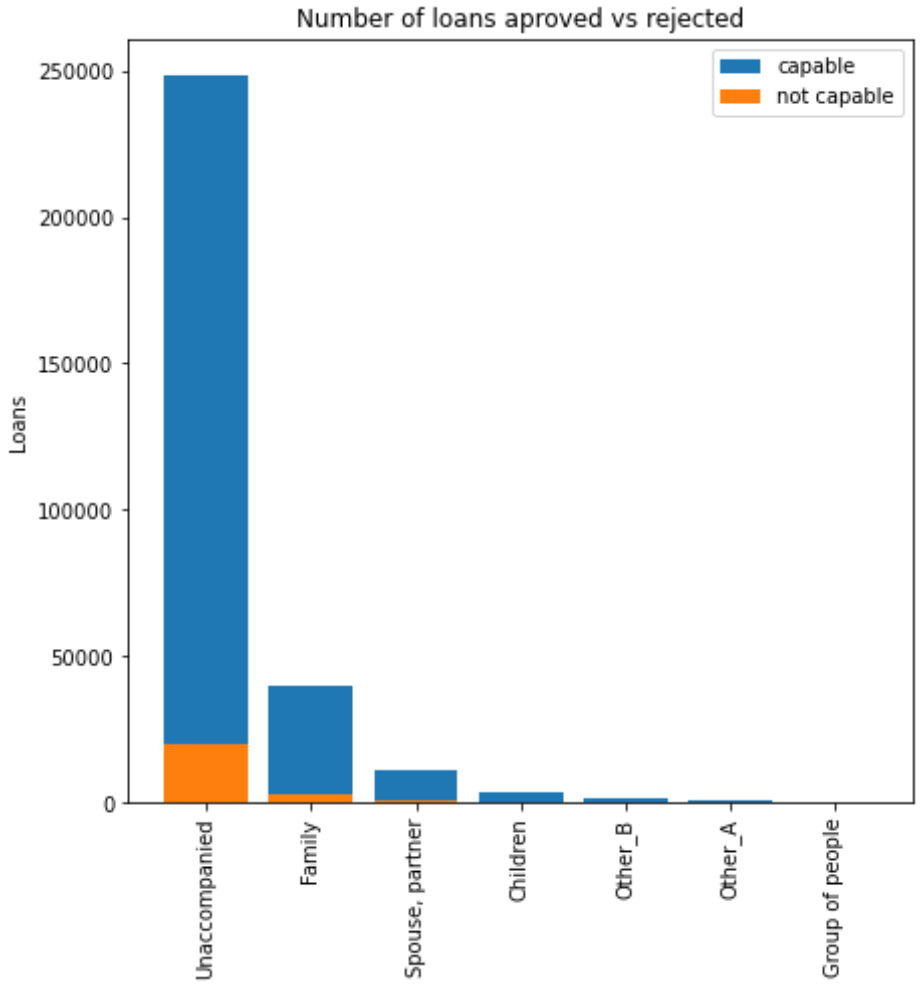
- Most number of loans are given for goods that are priced below Rs. 10 lakhs.

4.3.11 Univariate Analysis : Name_Type_Suite

In [29]:
```
train_data['NAME_TYPE_SUITE'].unique()
```

Out[29]:
```
array(['Unaccompanied', 'Family', 'Spouse, partner', 'Children',
       'Other_A', nan, 'Other_B', 'Group of people'], dtype=object)
```

In [30]:
```
univariate_barplots(train_data, 'NAME_TYPE_SUITE', 'TARGET', False)
```



	NAME_TYPE_SUITE	TARGET	total	Avg
6	Unaccompanied	20337	248526	0.081830
1	Family	3009	40149	0.074946
5	Spouse, partner	895	11370	0.078716
0	Children	241	3267	0.073768
4	Other_B	174	1770	0.098305

==========

	NAME_TYPE_SUITE	TARGET	total	Avg
5	Spouse, partner	895	11370	0.078716

```
0         Children     241   3267  0.073768
4          Other_B     174   1770  0.098305
3          Other_A      76    866  0.087760
2  Group of people      23    271  0.084871
```

Observations :

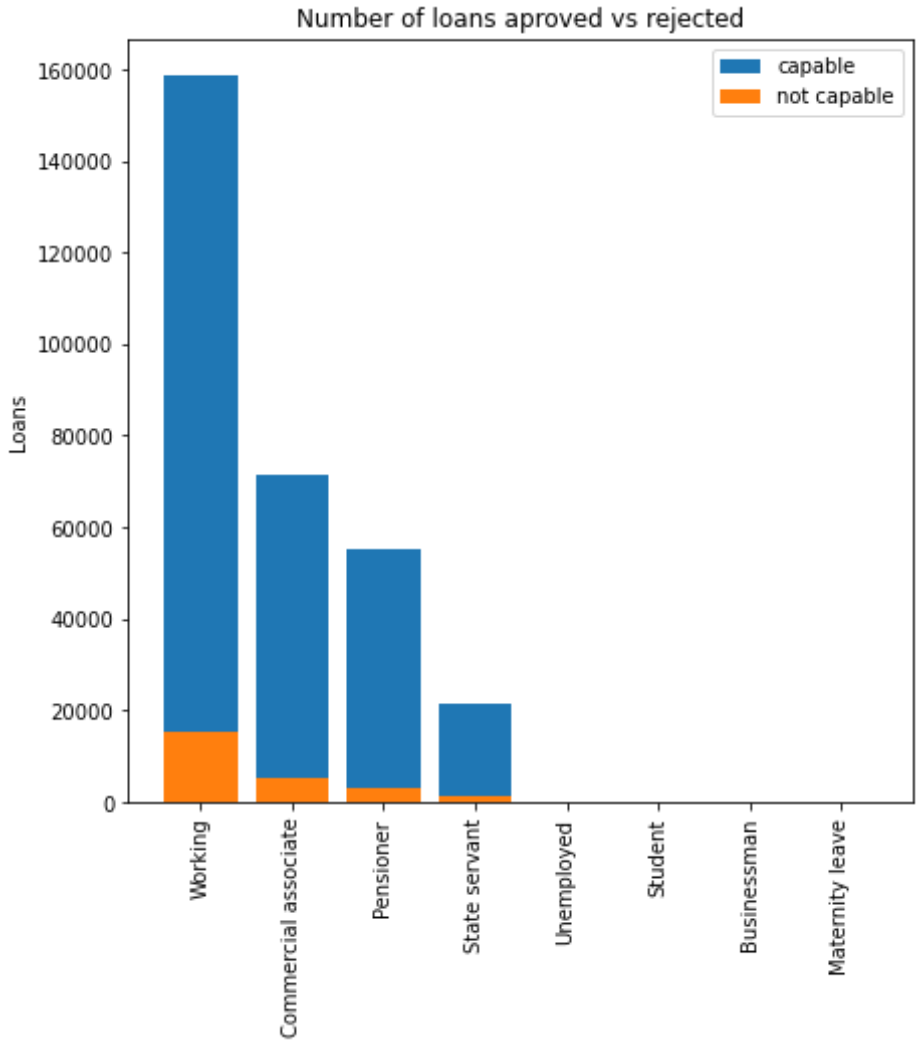
- For the various types of people accompanying the client for loan, the client comes unaccompanied to the bank in the most number of cases, out of which approx. 92% of the time, the bank finds the client to be capable of loan repayment whereas the remaining 8% of the time, the client is not capable of the same.
- Both in capability and non capability, 'Unaccompanied' as a class is the majority class in this case.
- The curve over here falls very sharply, which means that there is a lot of variability.

4.3.12 Univariate Analysis : Name_Income_Type

In [31]:
```
train_data['NAME_INCOME_TYPE'].unique()
```

Out[31]:
```
array(['Working', 'State servant', 'Commercial associate', 'Pensioner',
       'Unemployed', 'Student', 'Businessman', 'Maternity leave'],
      dtype=object)
```

In [32]:
```
train_data['NAME_INCOME_TYPE'].fillna('Data_Not_Available', inplace=True)
univariate_barplots(train_data, 'NAME_INCOME_TYPE', 'TARGET', False)
```



	NAME_INCOME_TYPE	TARGET	total	Avg
7	Working	15224	158774	0.095885
1	Commercial associate	5360	71617	0.074843
3	Pensioner	2982	55362	0.053864
4	State servant	1249	21703	0.057550
6	Unemployed	8	22	0.363636

==========

	NAME_INCOME_TYPE	TARGET	total	Avg
4	State servant	1249	21703	0.057550
6	Unemployed	8	22	0.363636
5	Student	0	18	0.000000
0	Businessman	0	10	0.000000
2	Maternity leave	2	5	0.400000

Observations :

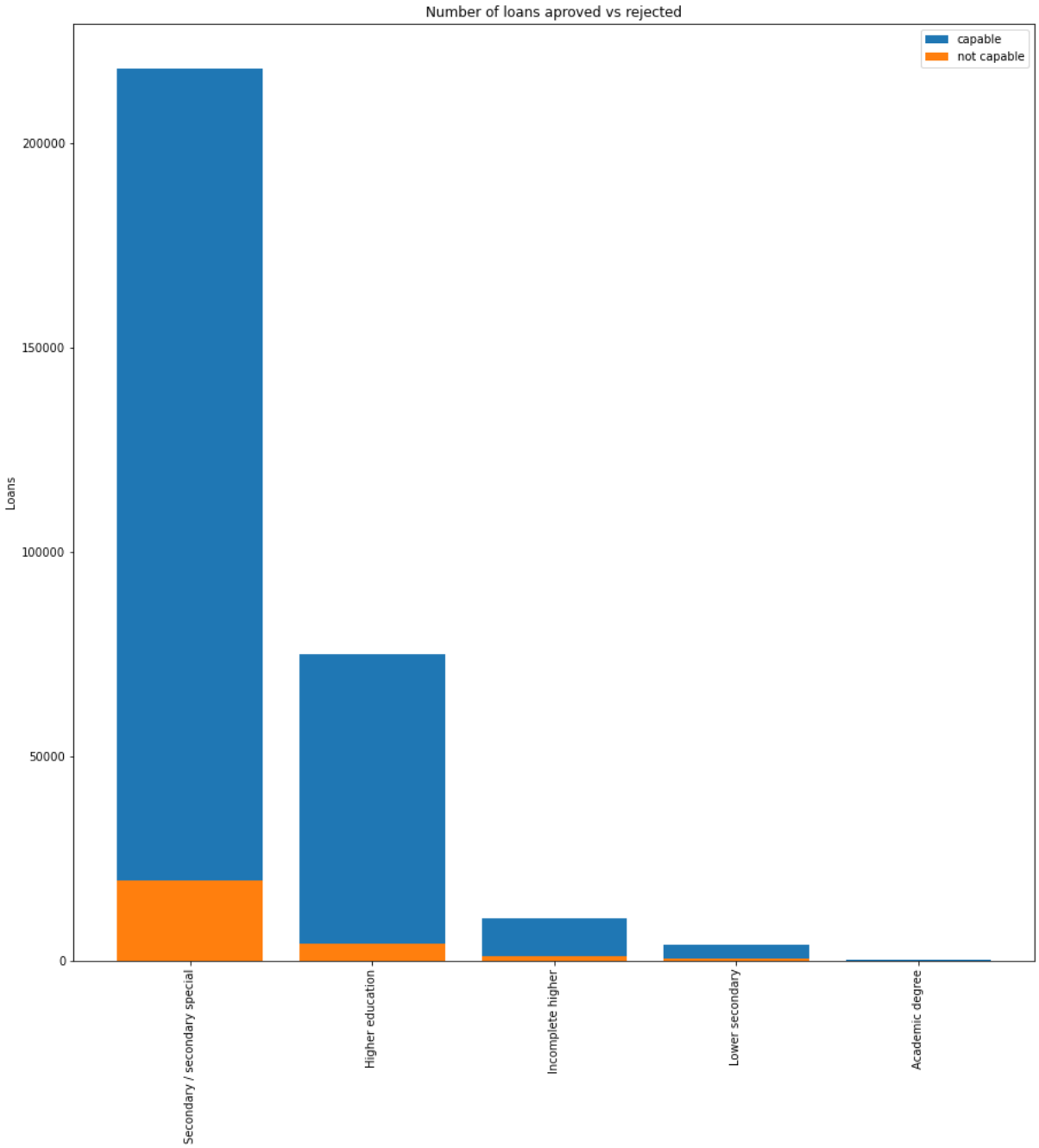
- The people who are working take the most number of loans whereas Commercial Associates, Pensioners and State Servants take considerably lesser number of loans.
- We have very little datapoints related to Unemployed people, Students, Businessmen and women on Maternity leave. Again, there's a lot of variability in this scenario.
- One interesting observation over here is the fact that whatever loans the students and businessmen have applied to, they have been deemed capable of repayment of the same.

4.3.13 Univariate Analysis : Name_Education_Type

In [33]:
```
train_data['NAME_EDUCATION_TYPE'].unique()
```

Out[33]:
```
array(['Secondary / secondary special', 'Higher education',
       'Incomplete higher', 'Lower secondary', 'Academic degree'],
      dtype=object)
```

In [34]:
```
train_data['NAME_EDUCATION_TYPE'].fillna('Data_Not_Available', inplace=
True)
univariate_barplots(train_data, 'NAME_EDUCATION_TYPE', 'TARGET', False)
```



```
                NAME_EDUCATION_TYPE  TARGET   total       Avg
4  Secondary / secondary special   19524  218391  0.089399
1                Higher education    4009   74863  0.053551
2              Incomplete higher     872   10277  0.084850
3                Lower secondary     417    3816  0.109277
0                Academic degree       3     164  0.018293
==================================================
                NAME_EDUCATION_TYPE  TARGET   total       Avg
4  Secondary / secondary special   19524  218391  0.089399
1                Higher education    4009   74863  0.053551
2              Incomplete higher     872   10277  0.084850
3                Lower secondary     417    3816  0.109277
0                Academic degree       3     164  0.018293
```

Observations :

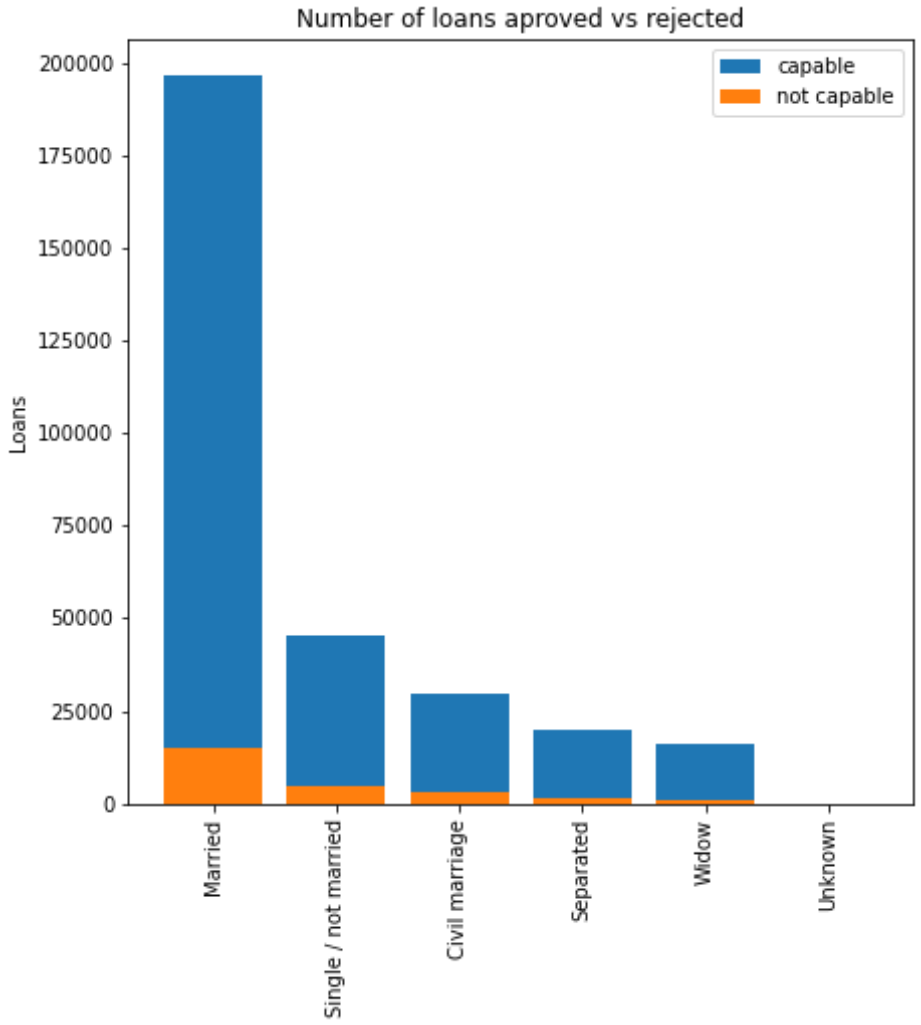
- Again, there's a lot of variability in this scenario among education types of the applicants.
- People with Secondary/Secondary Special as the highest level of education apply for most number of loans and they are also the highest defaulters. However, the default percentage is not very different across various education levels.

4.3.14 Univariate Analysis : Name_Family_Status

In [35]:
```
train_data['NAME_FAMILY_STATUS'].unique()
```

Out[35]:
```
array(['Single / not married', 'Married', 'Civil marriage', 'Widow',
       'Separated', 'Unknown'], dtype=object)
```

In [36]:
```
train_data['NAME_FAMILY_STATUS'].fillna('Data_Not_Available', inplace=True)
univariate_barplots(train_data, 'NAME_FAMILY_STATUS', 'TARGET', False)
```



	NAME_FAMILY_STATUS	TARGET	total	Avg
1	Married	14850	196432	0.075599
3	Single / not married	4457	45444	0.098077
0	Civil marriage	2961	29775	0.099446
2	Separated	1620	19770	0.081942
5	Widow	937	16088	0.058242

==========

```
     NAME_FAMILY_STATUS  TARGET  total       Avg
3  Single / not married    4457  45444  0.098077
0        Civil marriage    2961  29775  0.099446
2             Separated    1620  19770  0.081942
5                 Widow     937  16088  0.058242
4               Unknown       0      2  0.000000
```

Observations :

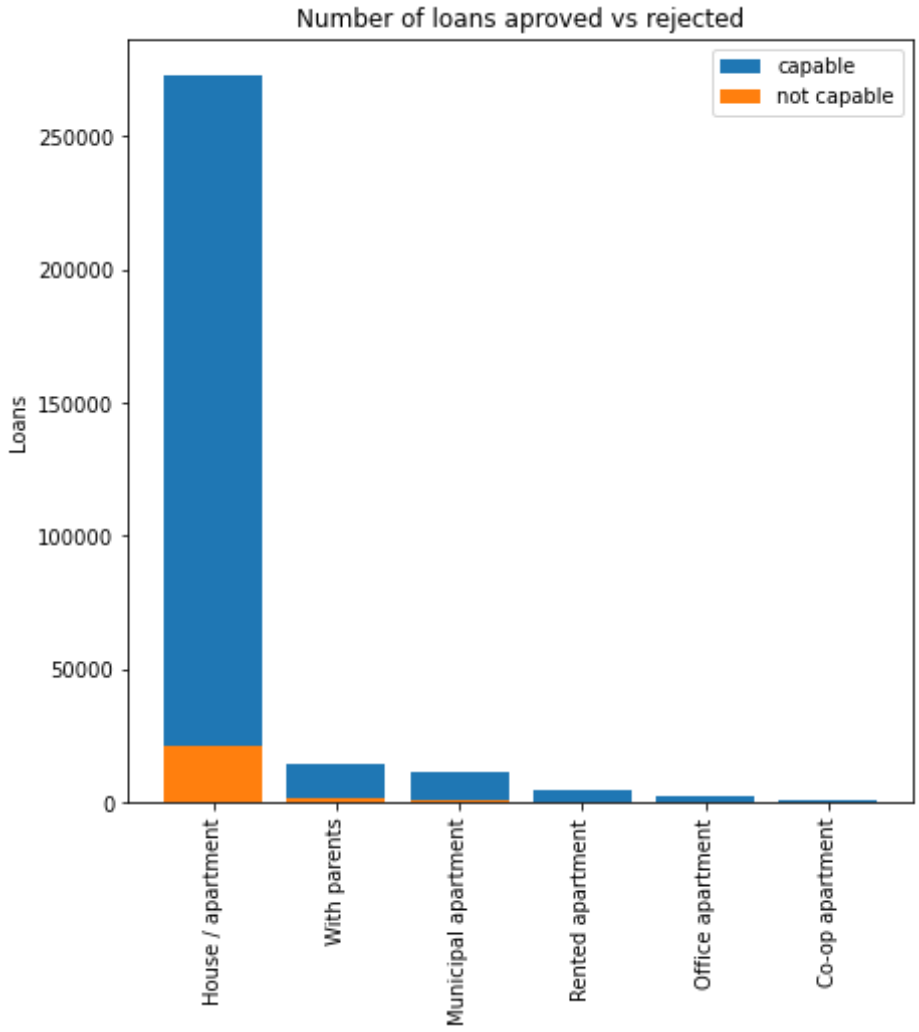
- There is variability among the Family Status of the applicants but there is not much variability if the majority class (Married) is ignored.
- Married people apply for the most number of loans and the number of people deemed incapable of repayment is also the highest.

4.3.15 Univariate Analysis : Name_Housing_Type

In [37]:
```
train_data['NAME_HOUSING_TYPE'].unique()
```

Out[37]:
```
array(['House / apartment', 'Rented apartment', 'With parents',
       'Municipal apartment', 'Office apartment', 'Co-op apartment'],
      dtype=object)
```

In [38]:
```
train_data['NAME_HOUSING_TYPE'].fillna('Data_Not_Available', inplace=True)
univariate_barplots(train_data, 'NAME_HOUSING_TYPE', 'TARGET', False)
```



	NAME_HOUSING_TYPE	TARGET	total	Avg
1	House / apartment	21272	272868	0.077957
5	With parents	1736	14840	0.116981
2	Municipal apartment	955	11183	0.085397
4	Rented apartment	601	4881	0.123131
3	Office apartment	172	2617	0.065724

==========

	NAME_HOUSING_TYPE	TARGET	total	Avg

5	With parents	1736	14840	0.116981
2	Municipal apartment	955	11183	0.085397
4	Rented apartment	601	4881	0.123131
3	Office apartment	172	2617	0.065724
0	Co-op apartment	89	1122	0.079323

Observations :

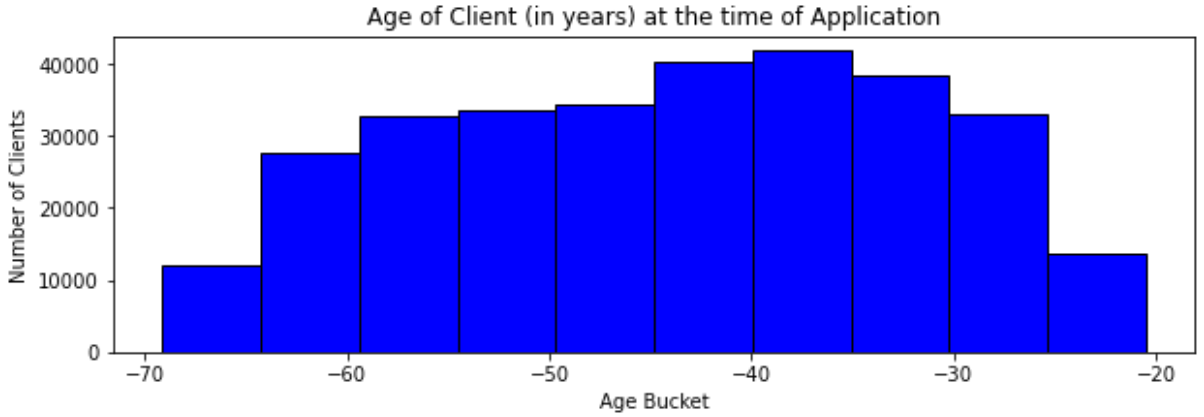
- People living in a House/Apartment apply for the most number of loans and the number of people deemed incapable of repayment in this case is also the highest, whereas if percentages are looked at, people living in rented apartment have the highest chance of default.

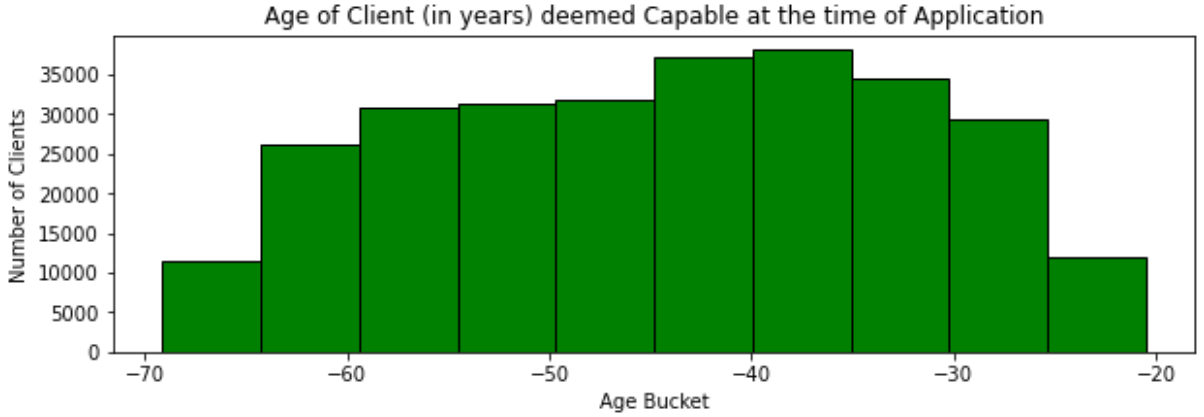
4.3.16 Univariate Analysis : Days_Birth

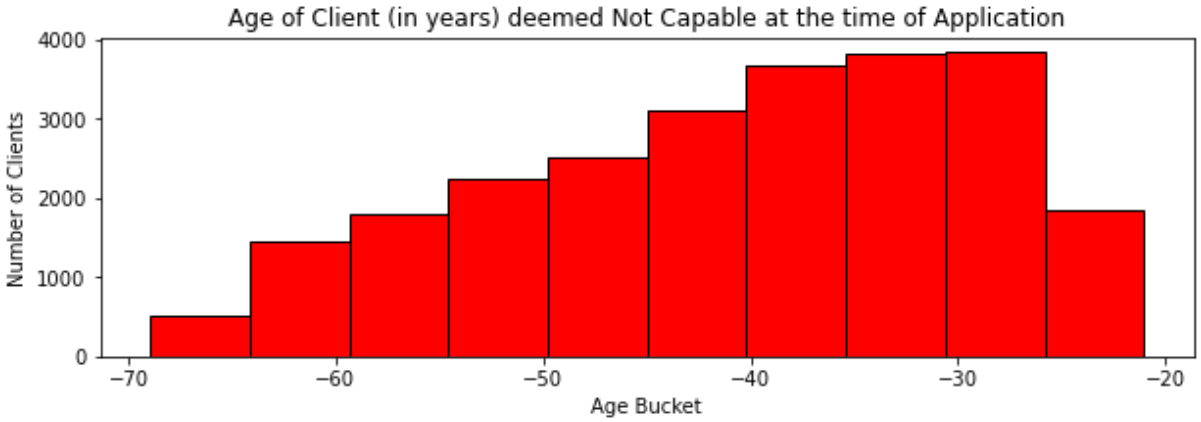
In [39]:
```
capable_days_birth = train_data[train_data['TARGET']==0]['DAYS_BIRTH'].
values/365
not_capable_days_birth = train_data[train_data['TARGET']==1]['DAYS_BIRT
H'].values/365
```

In [40]:
```
plt.figure(figsize=(10,3))
plt.hist(train_data['DAYS_BIRTH'].values/365, bins=10, edgecolor='blac
k', color='blue')
plt.title('Age of Client (in years) at the time of Application')
plt.xlabel('Age Bucket')
plt.ylabel('Number of Clients')
plt.show()

plt.figure(figsize=(10,3))
plt.hist(capable_days_birth, bins=10, edgecolor='black', color='green')
plt.title('Age of Client (in years) deemed Capable at the time of Appli
cation')
plt.xlabel('Age Bucket')
plt.ylabel('Number of Clients')
```

```
plt.show()

plt.figure(figsize=(10,3))
plt.hist(not_capable_days_birth, bins=10, edgecolor='black', color='red')
plt.title('Age of Client (in years) deemed Not Capable at the time of Application')
plt.xlabel('Age Bucket')
plt.ylabel('Number of Clients')
plt.show()
```







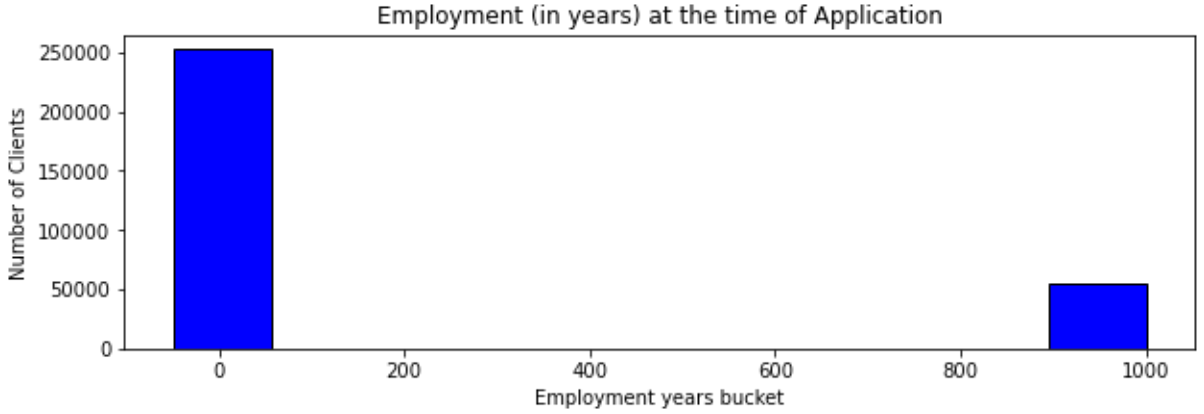
Observations :

- Most number of people applying for loans are in the range of (35-40) years whereas this is followed by people in the range of (40-45) years whereas the number of applicants in people aged <25 or aged>65 is very low.
- Again, for the people who are deemed capable of loan repayment, people in the same age buckets of (35-40) years and (40-45) years are deemed to be most capable.
- People aged in the buckets (25-30) years and (30-35) years have a large chance of being deemed not capable for loan repayment.

4.3.17 Univariate Analysis : Days_Employed

In [41]:
```
capable_days_employed = train_data[train_data['TARGET']==0]['DAYS_EMPLOYED'].values/365
not_capable_days_employed = train_data[train_data['TARGET']==1]['DAYS_EMPLOYED'].values/365
```

In [42]:
```
plt.figure(figsize=(10,3))
plt.hist(train_data['DAYS_EMPLOYED'].values/365, bins=10, edgecolor='black', color='blue')
plt.title('Employment (in years) at the time of Application')
plt.xlabel('Employment years bucket')
plt.ylabel('Number of Clients')
plt.show()
```



Observations :

- Here also, we have already converted the days into years for easy analysis, and we can see from the histogram that there are some clients that have worked for 1000 years.
- This is clearly impossible and is an outlier, which we will deal with later.

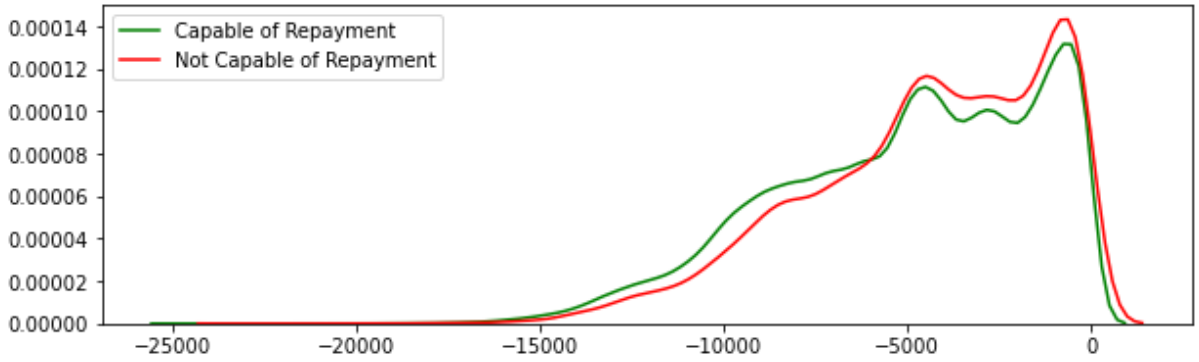
4.3.18 Univariate Analysis: Days_Registration

Days_Registration basically defines the number of days before the loan application that the client has changed the registration.

In [43]:
```
capable_days_registration = train_data[train_data['TARGET']==0]['DAYS_REGISTRATION'].values
```

```
not_capable_days_registration = train_data[train_data['TARGET']==1]['DA
YS_REGISTRATION'].values
```

In [44]:
```
plt.figure(figsize=(10,3))
sns.distplot(capable_days_registration,hist=False,label="Capable of Rep
ayment", color='green')
sns.distplot(not_capable_days_registration,hist=False,label="Not Capabl
e of Repayment", color='red')
plt.legend()
plt.show()
```



Observations :

- Most of the clients have changed their registration less than 15000 days (41 years) before the loan application, whereas in most cases it is less than 5000 days (13 years).

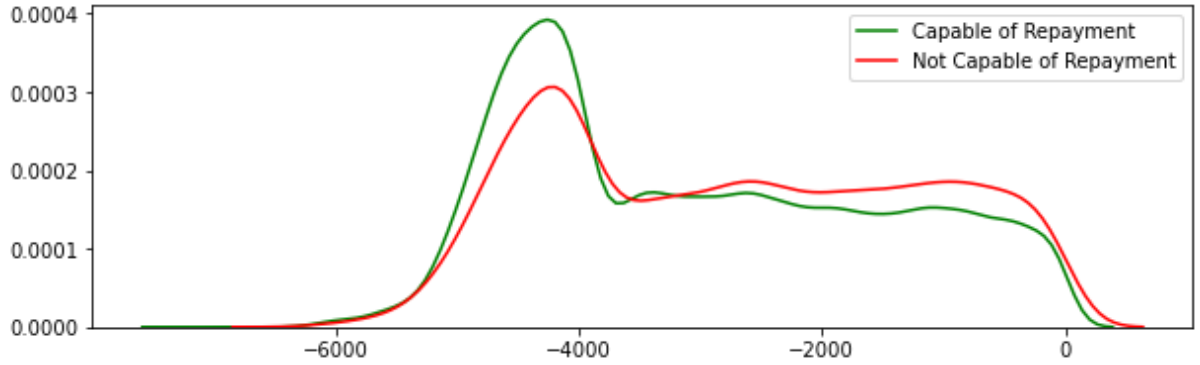
4.3.19 Univariate Analysis: Days_ID_Publish

This basically refers to the number of days before the application date that the client changed the identity document with which he applied for the loan.

In [45]:
```
capable_days_id_publish = train_data[train_data['TARGET']==0]['DAYS_ID_
PUBLISH'].values
```

```
not_capable_days_id_publish = train_data[train_data['TARGET']==1]['DAYS
_ID_PUBLISH'].values
```

In [46]:
```
plt.figure(figsize=(10,3))
sns.distplot(capable_days_id_publish,hist=False,label="Capable of Repay
ment", color='green')
sns.distplot(not_capable_days_id_publish,hist=False,label="Not Capable
 of Repayment", color='red')
plt.legend()
plt.show()
```



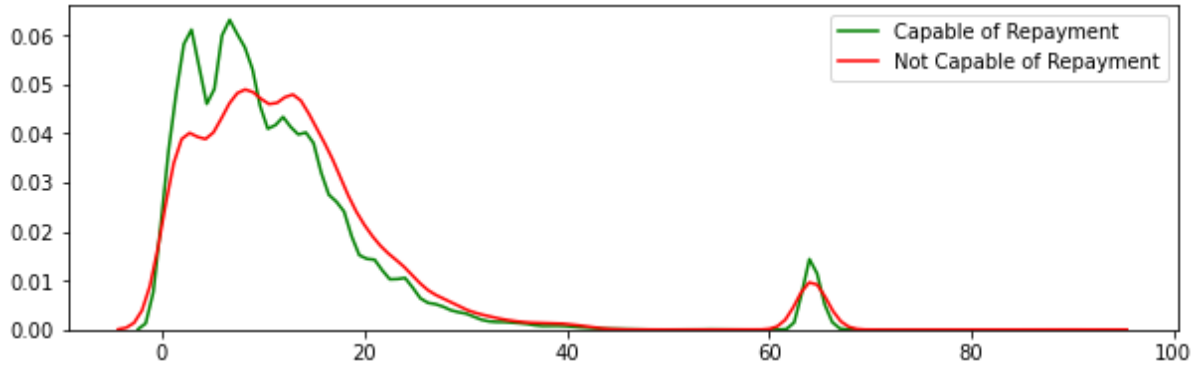
Observations :

- Most of the clients have changed their identity document around 4000 days (10.95 years) before the application date.

4.3.20 Univariate Analysis: Own_Car_Age

In [47]:
```
capable_car_age = train_data[train_data['TARGET']==0]['OWN_CAR_AGE'].va
lues
not_capable_car_age = train_data[train_data['TARGET']==1]['OWN_CAR_AGE'
].values
```

In [48]:
```
plt.figure(figsize=(10,3))
sns.distplot(capable_car_age,hist=False,label="Capable of Repayment", c
olor='green')
sns.distplot(not_capable_car_age,hist=False,label="Not Capable of Repay
ment", color='red')
plt.legend()
plt.show()
```



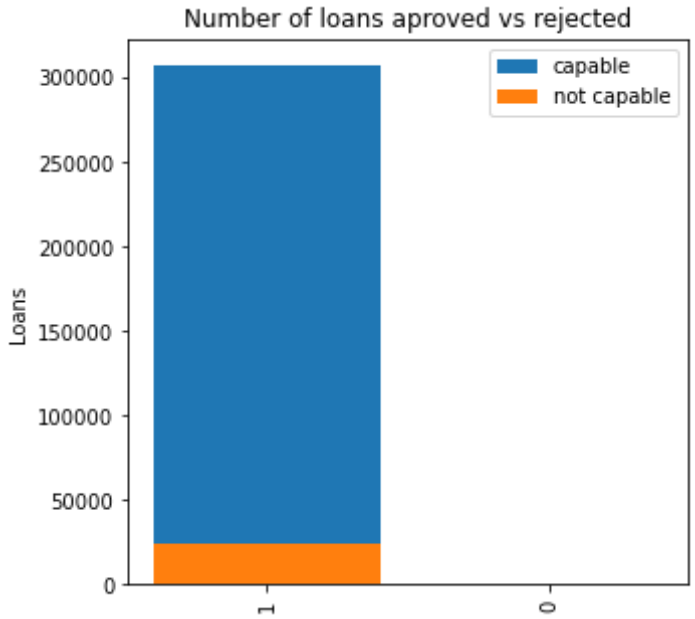
Observations :

- Most of the clients have their cars less than 20 years old whereas there are very few cars that are older than 20 years.

4.3.21 Univariate Analysis: Flag_Mobil

This is the Flag that indicates whether the loan applicant possesses a Mobile Phone or not. 1 means that yes, he/she owns a Mobile Phone whereas 0 means that he/she doesn't own a mobile phone.

In [49]:
```
train_data['FLAG_MOBIL'].fillna('Data_Not_Available', inplace=True)
univariate_barplots(train_data, 'FLAG_MOBIL', 'TARGET', False)
```



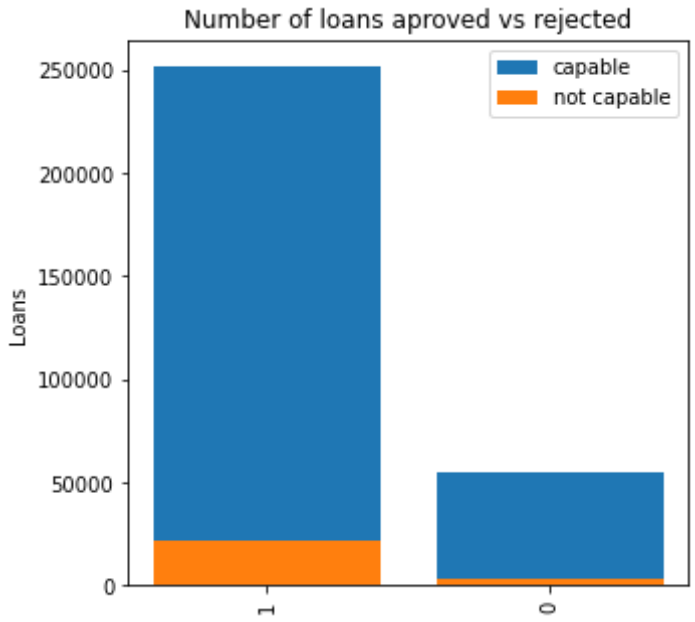

```
   FLAG_MOBIL  TARGET   total       Avg
1           1   24825  307510  0.080729
0           0       0       1  0.000000
================================================
   FLAG_MOBIL  TARGET   total       Avg
1           1   24825  307510  0.080729
0           0       0       1  0.000000
```

Observations :

- Out of the total 307511 loan applications in the Train Data, nearly everyone in the application owns a Mobile Phone out of which 92% are deemed capable and 8% are deemed incapable.
- There is only 1 applicant in the training data that does not own a Mobile Phone.

4.3.22 Univariate Analysis: Flag_Emp_Phone

In [50]:
```
train_data['FLAG_EMP_PHONE'].fillna('Data_Not_Available', inplace=True)
univariate_barplots(train_data, 'FLAG_EMP_PHONE', 'TARGET', False)
```



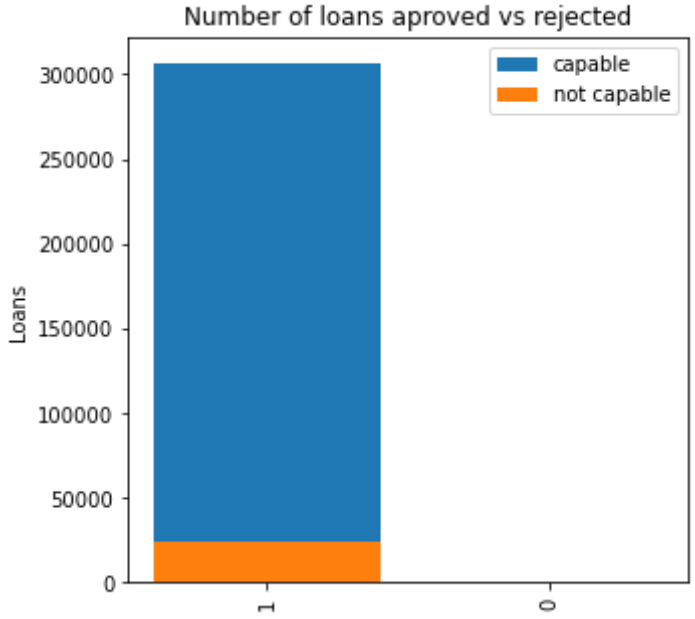

```
   FLAG_EMP_PHONE  TARGET   total       Avg
1               1   21834  252125  0.086600
0               0    2991   55386  0.054003
================================================
   FLAG_EMP_PHONE  TARGET   total       Avg
1               1   21834  252125  0.086600
0               0    2991   55386  0.054003
```

Observations :

- A lot of the applicants provided their Work Phone (83%) as compared to the applicants who did not provide their work phone (17%).

4.3.23 Univariate Analysis: Flag_Cont_Mobile

In [51]:
```
train_data['FLAG_CONT_MOBILE'].fillna('Data_Not_Available', inplace=True)
univariate_barplots(train_data, 'FLAG_CONT_MOBILE', 'TARGET', False)
```



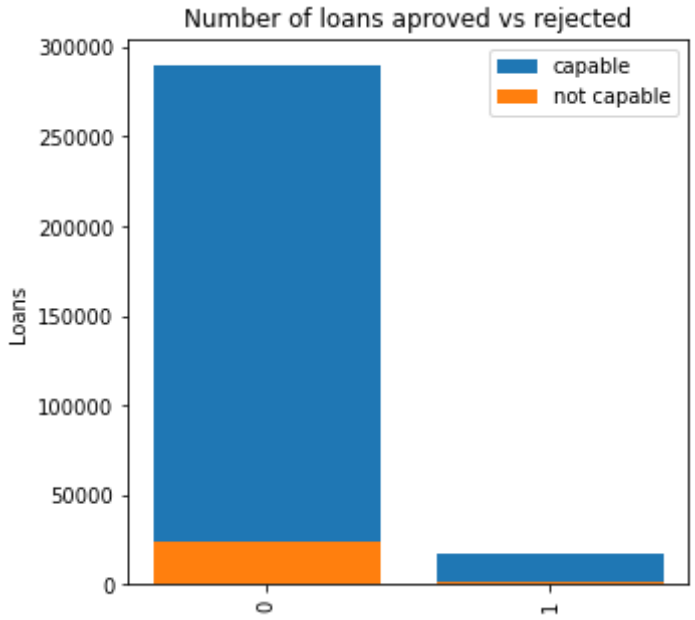

```
   FLAG_CONT_MOBILE  TARGET   total       Avg
1                 1   24780  306937  0.080733
0                 0      45     574  0.078397
================================================
   FLAG_CONT_MOBILE  TARGET   total       Avg
1                 1   24780  306937  0.080733
0                 0      45     574  0.078397
```

Observations :

- 99% of the applicants had their phone reachable when the Bank tried to contact them.

4.3.24 Univariate Analysis: Flag_Email

In [52]:
```
train_data['FLAG_EMAIL'].fillna('Data_Not_Available', inplace=True)
univariate_barplots(train_data, 'FLAG_EMAIL', 'TARGET', False)
```




```
   FLAG_EMAIL  TARGET   total       Avg
0           0   23451  290069  0.080846
1           1    1374   17442  0.078775
==================================================
   FLAG_EMAIL  TARGET   total       Avg
0           0   23451  290069  0.080846
1           1    1374   17442  0.078775
```

Observations :

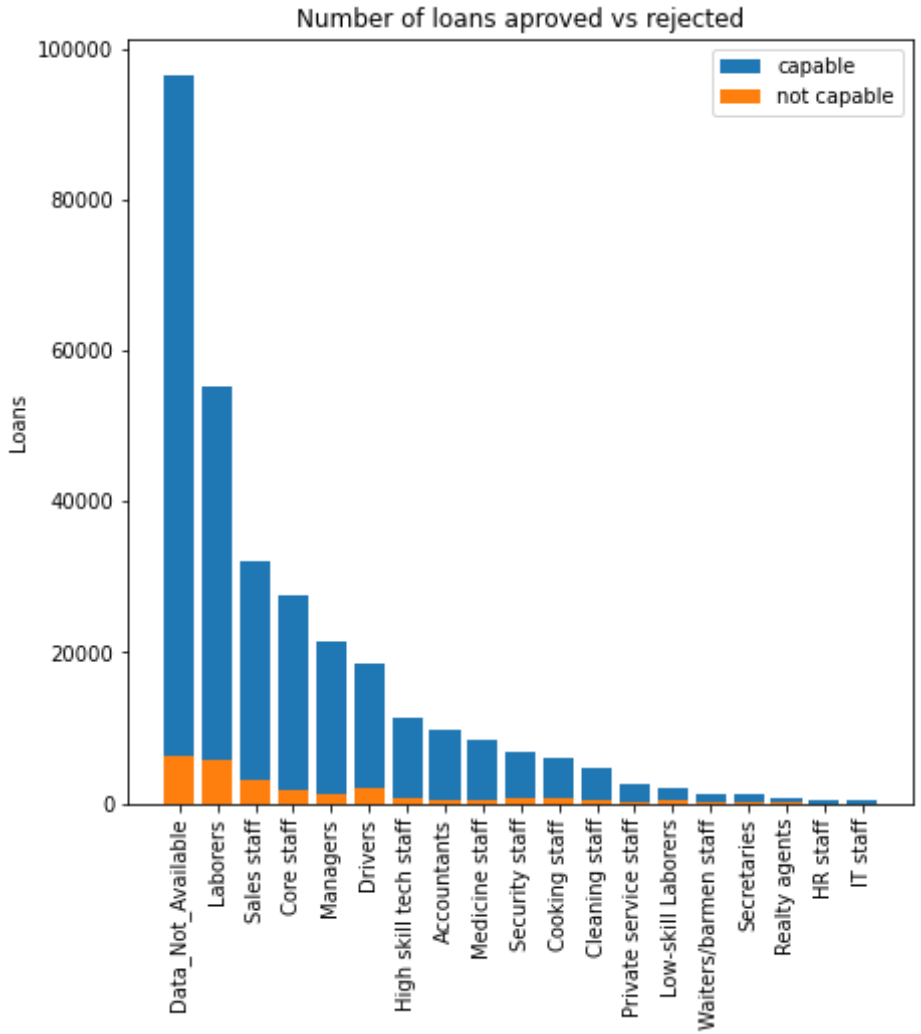
- This goes on to show that approximately 94% of the clients did not provide their Email Address in the application, and only 6% of the clients provided Email details.

4.3.25 Univariate Analysis: Occupation_Type

In [53]:
```
train_data['OCCUPATION_TYPE'].unique()
```

Out[53]:
```
array(['Laborers', 'Core staff', 'Accountants', 'Managers', nan,
       'Drivers', 'Sales staff', 'Cleaning staff', 'Cooking staff',
       'Private service staff', 'Medicine staff', 'Security staff',
       'High skill tech staff', 'Waiters/barmen staff',
       'Low-skill Laborers', 'Realty agents', 'Secretaries', 'IT staf
f',
       'HR staff'], dtype=object)
```

In [54]:
```
train_data['OCCUPATION_TYPE'].fillna('Data_Not_Available', inplace=True
)
univariate_barplots(train_data, 'OCCUPATION_TYPE', 'TARGET', False)
```



	OCCUPATION_TYPE	TARGET	total	Avg
4	Data_Not_Available	6278	96391	0.065131
9	Laborers	5838	55186	0.105788
15	Sales staff	3092	32102	0.096318
3	Core staff	1738	27570	0.063040
11	Managers	1328	21371	0.062140

==========

```
          OCCUPATION_TYPE  TARGET  total       Avg
18  Waiters/barmen staff     152   1348  0.112760
16           Secretaries      92   1305  0.070498
14         Realty agents      59    751  0.078562
6               HR staff      36    563  0.063943
8               IT staff      34    526  0.064639
```

Observations :

- Out of all the possible Occupation Types, the majority of applicants have not provided their Occupation Type in the application (approx. 31.39%) which is followed by Laborers (approx. 18%).
- Out of all the occupations, Waiters/barmen staff are considered to be the least capable of repayment followed by Laborers -> though laborers have considerably higher applications as compared to Waiters/barmen staff.

4.3.26 Univariate Analysis: Cnt_Fam_Members

In [55]:
```
train_data['CNT_FAM_MEMBERS'].isnull().sum()
```

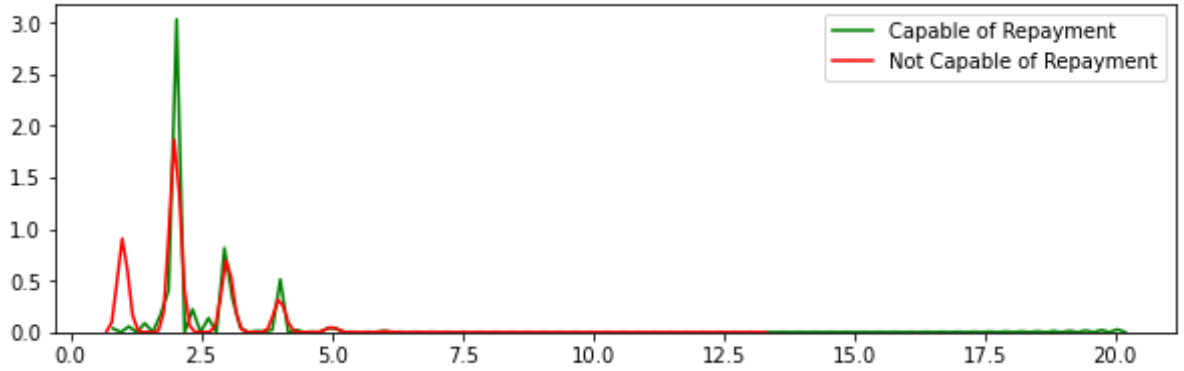
Out[55]: 2

In [56]:
```
#Replace NA with the most frequently occuring class for Count of Client Family Members
train_data['CNT_FAM_MEMBERS'].fillna(train_data['CNT_FAM_MEMBERS'].value_counts().idxmax(), \
                                     inplace=True)
```

In [57]:
```
capable_family_members = train_data[train_data['TARGET']==0]['CNT_FAM_MEMBERS'].values
not_capable_family_members = train_data[train_data['TARGET']==1]['CNT_FAM_MEMBERS'].values
```

In [58]:
```
plt.figure(figsize=(10,3))
```

```
sns.distplot(capable_family_members,hist=False,label="Capable of Repaym
ent", color='green')
sns.distplot(not_capable_family_members,hist=False,label="Not Capable o
f Repayment", color='red')
plt.legend()
plt.show()
```

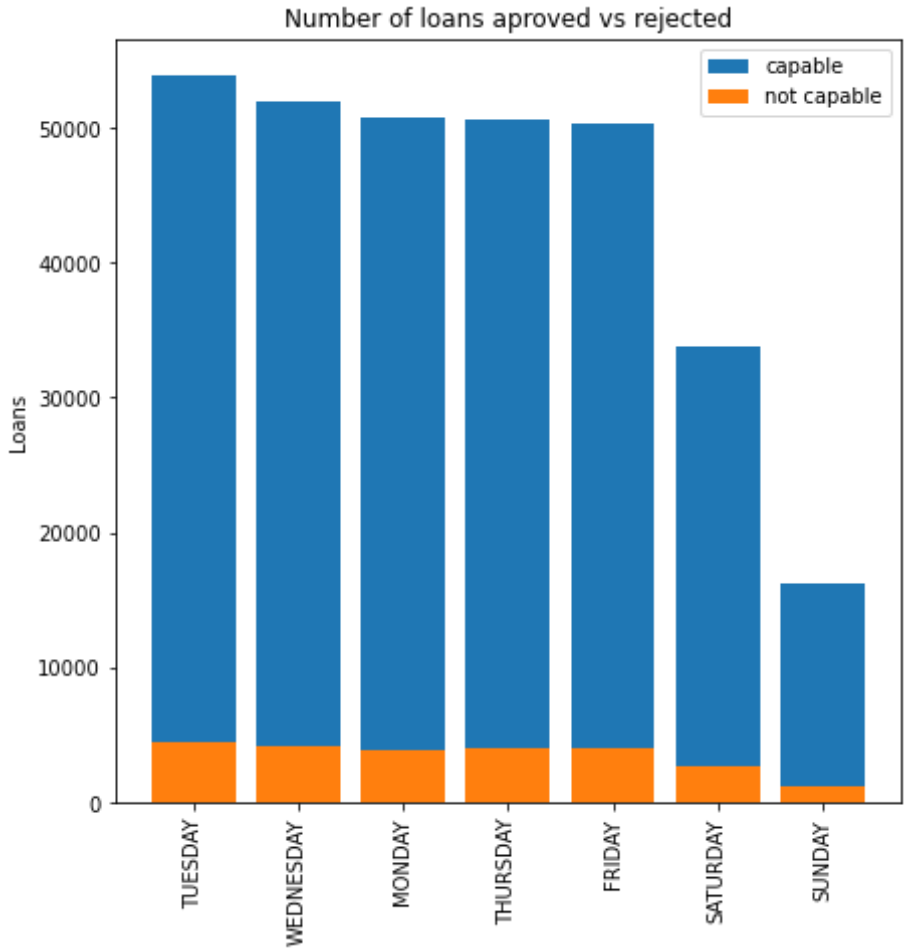


Observations :

- Most of the applicants have 2 Family Members and there are very few applicants with >5 family members.

4.3.27 Univariate Analysis: Weekday_Appr_Process_Start

In [59]:
```
univariate_barplots(train_data, 'WEEKDAY_APPR_PROCESS_START', 'TARGET',
False)
```



	WEEKDAY_APPR_PROCESS_START	TARGET	total	Avg
5	TUESDAY	4501	53901	0.083505
6	WEDNESDAY	4238	51934	0.081604
1	MONDAY	3934	50714	0.077572
4	THURSDAY	4098	50591	0.081003
0	FRIDAY	4101	50338	0.081469

==========

	WEEKDAY_APPR_PROCESS_START	TARGET	total	Avg
1	MONDAY	3934	50714	0.077572

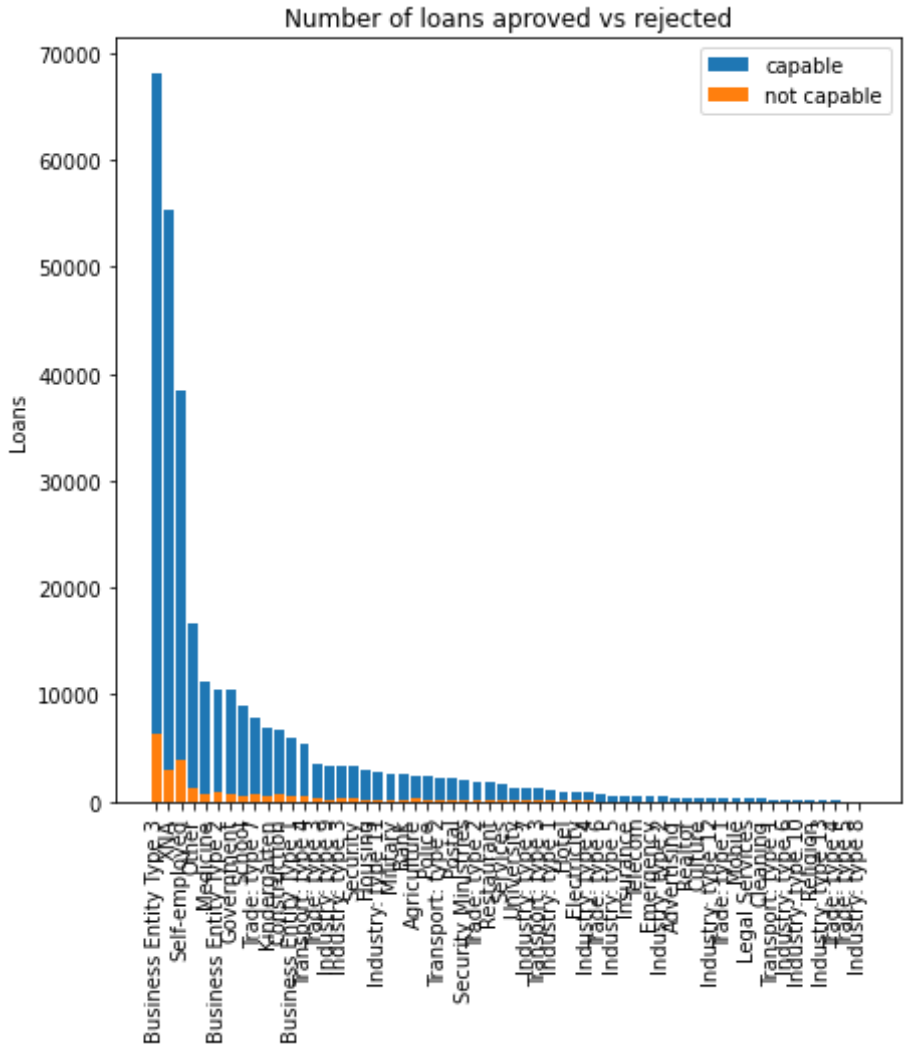
```
4                  THURSDAY    4098  50591  0.081003
0                    FRIDAY    4101  50338  0.081469
2                  SATURDAY    2670  33852  0.078873
3                    SUNDAY    1283  16181  0.079291
```

Observations :

- This is very interesting because the number of applications are spread almost uniformly throughout the weekdays (Monday-Friday) (approx. 50K applications per day) (16-17%), whereas the number of applications is very low on a Sunday.

4.3.28 Univariate Analysis: Organization_Type

In [60]:
```
univariate_barplots(train_data, 'ORGANIZATION_TYPE', 'TARGET', False)
```



	ORGANIZATION_TYPE	TARGET	total	Avg
5	Business Entity Type 3	6323	67992	0.092996
57	XNA	2990	55374	0.053996
42	Self-employed	3908	38412	0.101739
33	Other	1275	16683	0.076425
30	Medicine	737	11193	0.065845

==========

	ORGANIZATION_TYPE	TARGET	total	Avg

```
37           Religion       5     85  0.058824
18  Industry: type 13       9     67  0.134328
48      Trade: type 4       2     64  0.031250
49      Trade: type 5       3     49  0.061224
25   Industry: type 8       3     24  0.125000
```

Observations :

- Business people and XNA (No information provided) are the highest number of applicants but Business People and Self-Employed applicants have some of the highest default rate.

4.3.29 Univariate Analysis: EXT_SOURCE_1

EXT_SOURCE_1 is a Normalized Score from an External Data Source - 1.

In [61]:
```
train_data['EXT_SOURCE_1'].isnull().sum()
```

Out[61]: 173378

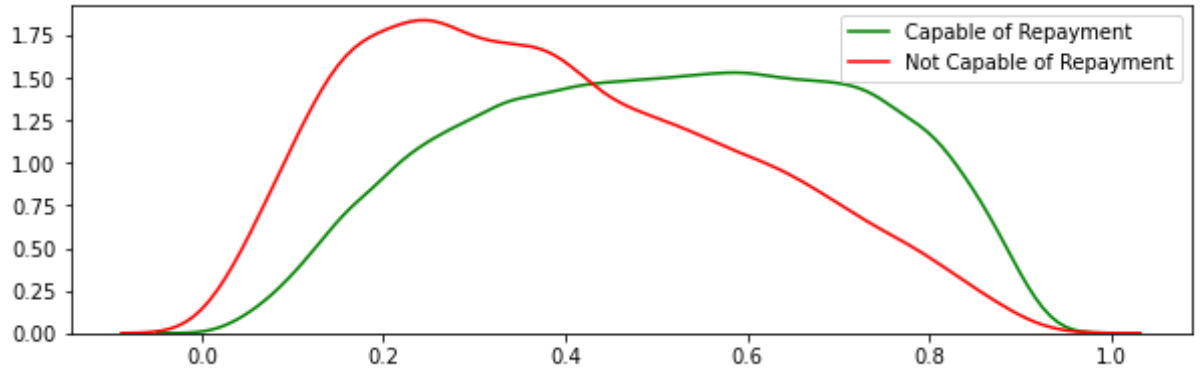
Note :

This basically shows that approx. 173378/307511 ie. nearly 56% of the entries in the EXT_SOURCE_1 column are nulls ie. empty values, and we will look at the remaining values only. We can carry out replacement of these values with the Mean/Median/Mode for the entire numerical column but since this is a large number of nulls, we will not follow that approach.

In [62]:
```
capable_ext_source_1 = train_data[train_data['TARGET']==0]['EXT_SOURCE_
1'].values
not_capable_ext_source_1 = train_data[train_data['TARGET']==1]['EXT_SOU
RCE_1'].values
```

In [63]:
```
plt.figure(figsize=(10,3))
sns.distplot(capable_ext_source_1,hist=False,label="Capable of Repaymen
```

```
t", color='green')
sns.distplot(not_capable_ext_source_1,hist=False,label="Not Capable of
 Repayment", color='red')
plt.legend()
plt.show()
```



Observations :

- This is the first feature that we are seeing where there is some considerable difference among the 2 classes, as we can see from the PDF plot.
- Therefore, EXT_SOURCE_1 is going to be an important feature.

4.3.30 Univariate Analysis: EXT_SOURCE_2

EXT_SOURCE_2 is a Normalized Score from an External Data Source - 2.

In [64]:
```
train_data['EXT_SOURCE_2'].isnull().sum()
```

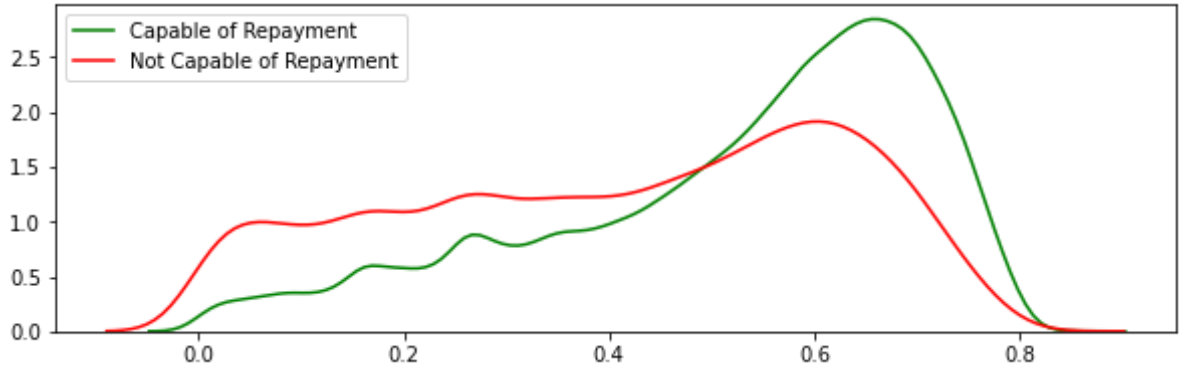
Out[64]:
```
660
```

Note :

This basically shows that approx. 660/307511 ie. only 0.2% of the entries in the column are nulls. We will analyse on the non-empty values in the column.

In [65]:
```
capable_ext_source_2 = train_data[train_data['TARGET']==0]['EXT_SOURCE_
2'].values
not_capable_ext_source_2 = train_data[train_data['TARGET']==1]['EXT_SOU
RCE_2'].values
```

In [66]:
```
plt.figure(figsize=(10,3))
sns.distplot(capable_ext_source_2,hist=False,label="Capable of Repaymen
t", color='green')
sns.distplot(not_capable_ext_source_2,hist=False,label="Not Capable of
 Repayment", color='red')
plt.legend()
plt.show()
```



Observations :

- Again, in this case also we can see that the data is reasonably well separated and hence this will also be an important feature.

4.3.31 Univariate Analysis: EXT_SOURCE_3

EXT_SOURCE_3 is a Normalized Score from an External Data Source - 3.

In [67]:
```
train_data['EXT_SOURCE_3'].isnull().sum()
```

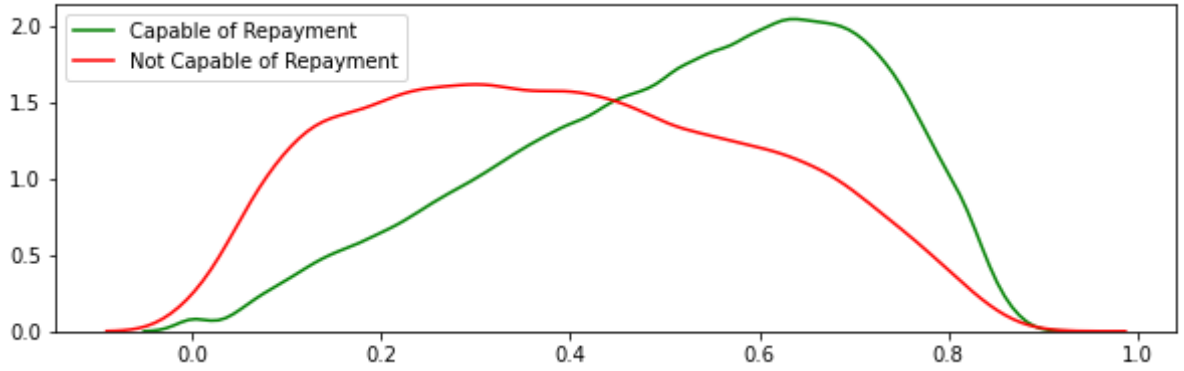
Out[67]:
```
60965
```

Note :

This basically shows that 60965/307511 ie. 19% of the entries in the column are nulls. We will analyse on the non-empty values in the column.

In [68]:
```
capable_ext_source_3 = train_data[train_data['TARGET']==0]['EXT_SOURCE_
3'].values
not_capable_ext_source_3 = train_data[train_data['TARGET']==1]['EXT_SOU
RCE_3'].values
```

In [69]:
```
plt.figure(figsize=(10,3))
sns.distplot(capable_ext_source_3,hist=False,label="Capable of Repaymen
t", color='green')
sns.distplot(not_capable_ext_source_3,hist=False,label="Not Capable of
 Repayment", color='red')
plt.legend()
plt.show()
```



Observations :

- Again, the data is reasonably well separated and hence this will also be an important feature.

4.3.32 Univariate Analysis: Flag_Document_x

There are multiple Flag Document Columns in the Train and Test Dataframes that we are to analyse if they are going to add much value to the models that we will build or not. In order to analyse, we will first take a subset of all the columns from the Dataframe that has the term 'FLAG' present in the column name.

In [70]:
```
# Refer :- https://stackoverflow.com/questions/43643506/select-columns-based-on-columns-names-containing-a-specific-string-in-pandas
flag_document_df = train_data.loc[:, train_data.columns.str.contains('DOCUMENT')]
flag_document_df.head(5)
```

Out[70]:

	FLAG_DOCUMENT_2	FLAG_DOCUMENT_3	FLAG_DOCUMENT_4	FLAG_DOCUMENT_5	FLAG_D
0	0	1	0	0	
1	0	1	0	0	
2	0	0	0	0	
3	0	1	0	0	
4	0	0	0	0	

Note

As we can see over here, we have a total of 20 Flag Document Columns in the dataframe pertaining to different documents where 1 denotes that the client has provided the corresponding

document in the loan application and 0 denotes that the client has not provided the corresponding document.

In [71]:
```
for column in flag_document_df:

    count_0 = flag_document_df[column].value_counts()[0]
    count_1 = flag_document_df[column].value_counts()[1]
    total_rows = flag_document_df.shape[0]

    percent_0 = np.round((count_0*100/total_rows),2)
    percent_1 = np.round(100 - percent_0,2)

    print(column, "contains percentage of 1's = ",percent_1,\
    "and percentage of 0's =", percent_0)
```

```
FLAG_DOCUMENT_2 contains percentage of 1's =  0.0 and percentage of 0's
= 100.0
FLAG_DOCUMENT_3 contains percentage of 1's =  71.0 and percentage of
0's = 29.0
FLAG_DOCUMENT_4 contains percentage of 1's =  0.01 and percentage of
0's = 99.99
FLAG_DOCUMENT_5 contains percentage of 1's =  1.51 and percentage of
0's = 98.49
FLAG_DOCUMENT_6 contains percentage of 1's =  8.81 and percentage of
0's = 91.19
FLAG_DOCUMENT_7 contains percentage of 1's =  0.02 and percentage of
0's = 99.98
FLAG_DOCUMENT_8 contains percentage of 1's =  8.14 and percentage of
0's = 91.86
FLAG_DOCUMENT_9 contains percentage of 1's =  0.39 and percentage of
0's = 99.61
FLAG_DOCUMENT_10 contains percentage of 1's =  0.0 and percentage of
0's = 100.0
FLAG_DOCUMENT_11 contains percentage of 1's =  0.39 and percentage of
0's = 99.61
FLAG_DOCUMENT_12 contains percentage of 1's =  0.0 and percentage of
0's = 100.0
FLAG_DOCUMENT_13 contains percentage of 1's =  0.35 and percentage of
0's = 99.65
```

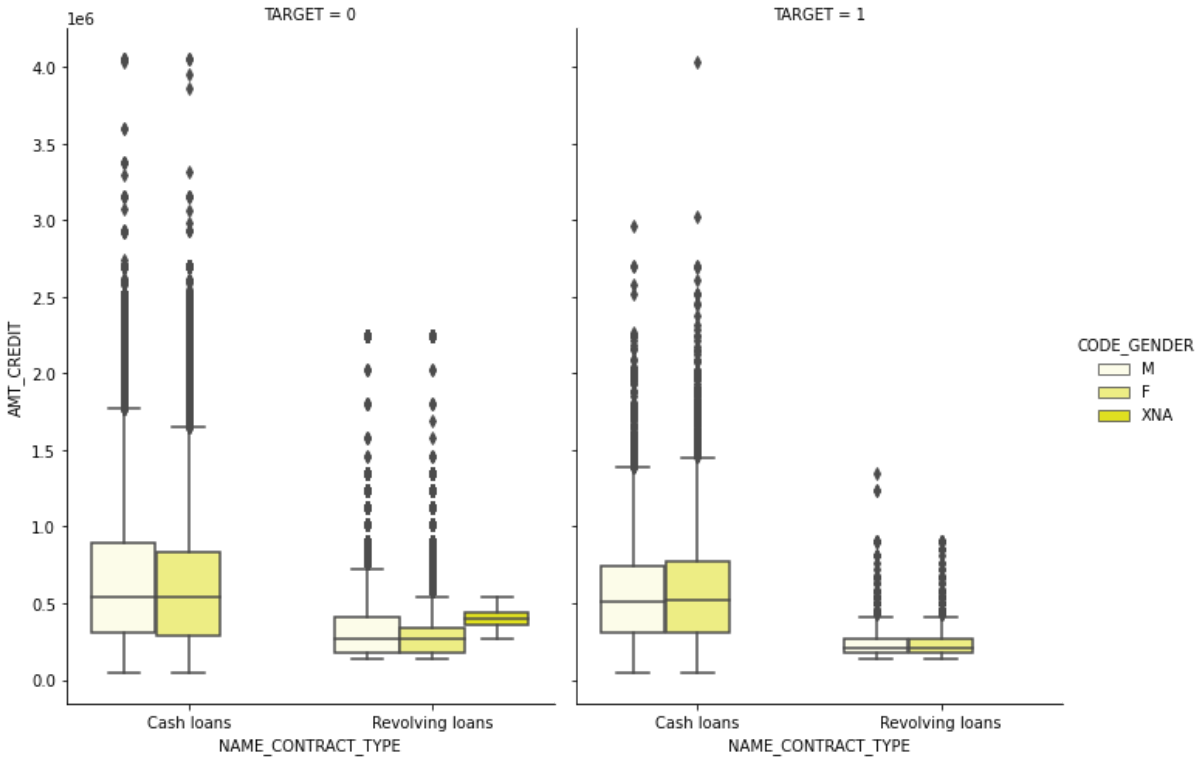
```
FLAG_DOCUMENT_14 contains percentage of 1's =  0.29 and percentage of
0's = 99.71
FLAG_DOCUMENT_15 contains percentage of 1's =  0.12 and percentage of
0's = 99.88
FLAG_DOCUMENT_16 contains percentage of 1's =  0.99 and percentage of
0's = 99.01
FLAG_DOCUMENT_17 contains percentage of 1's =  0.03 and percentage of
0's = 99.97
FLAG_DOCUMENT_18 contains percentage of 1's =  0.81 and percentage of
0's = 99.19
FLAG_DOCUMENT_19 contains percentage of 1's =  0.06 and percentage of
0's = 99.94
FLAG_DOCUMENT_20 contains percentage of 1's =  0.05 and percentage of
0's = 99.95
FLAG_DOCUMENT_21 contains percentage of 1's =  0.03 and percentage of
0's = 99.97
```

Observations :

- We can see from here that the percentage of 1s (ie. the client has submitted the document) is very small in most of the cases, which means that the data is highly imbalanced and its presence in the dataset is not going to help us very much.
- However, Flag_Document_3 has a good presence of 1s and we can remove all the Flag_Document columns except this one.

4.3.33 Bivariate Analysis: NAME_CONTRACT_TYPE vs AMT_CREDIT

In [72]:
```
sns.catplot(x="NAME_CONTRACT_TYPE", y="AMT_CREDIT", hue="CODE_GENDER" ,
col="TARGET",\
            data=train_data,color = "yellow",kind="box", height=7, aspe
ct=.7);
```



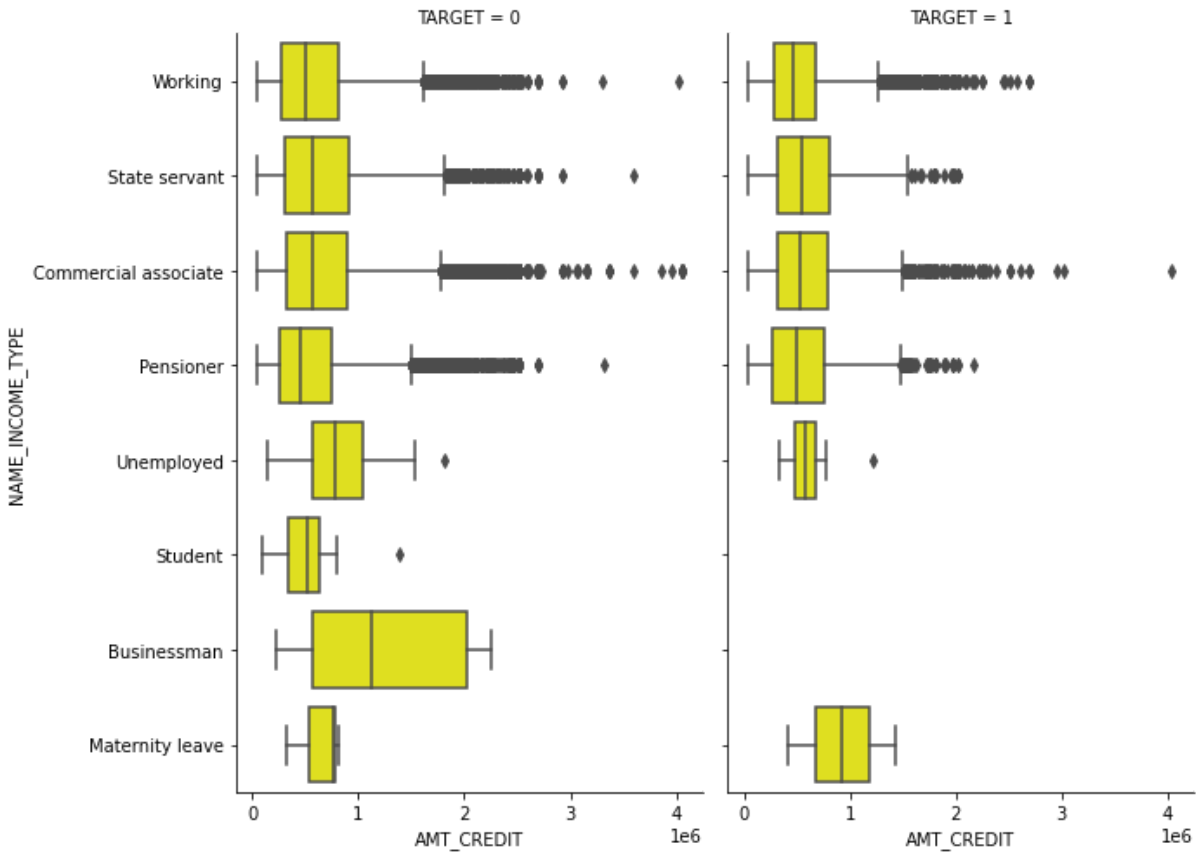
Observations :

- This shows that Men & Women with Cash Loans have higher chances of being deemed capable of loan repayment based on their Credit Amount.

4.3.34 Bivariate Analysis: AMT_CREDIT vs NAME_INCOME_TYPE

In [73]:
```
sns.catplot(x="AMT_CREDIT", y="NAME_INCOME_TYPE", col="TARGET",\
            data=train_data,color = "yellow",kind="box", height=7, aspe
```

```
ct=.7);
```



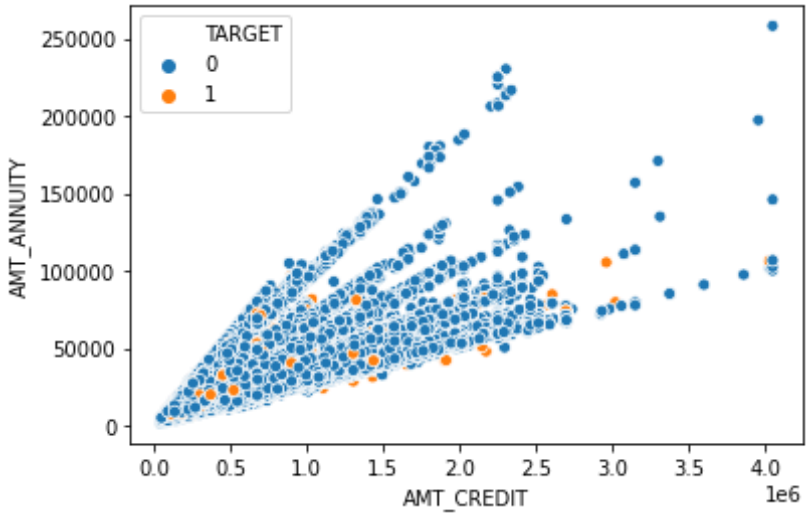
Observations :

- This shows that Applicants with a Higher Value of Credit Amount across various income types have a Higher Likelihood of deemed capable of Loan Repayment, especially in the case of 'Unemployed','Student' and 'Businessmen'.

4.3.35 Bivariate Analysis: AMT_CREDIT vs AMT_ANNUITY

In [74]:
```
sns.scatterplot(
    data=train_data, x="AMT_CREDIT", y="AMT_ANNUITY", hue="TARGET",\
    sizes=(20, 200)
)
```

Out[74]: <matplotlib.axes._subplots.AxesSubplot at 0x7f7fc25692e0>



Observations :

- This shows that both the Amount Credit and Amount Annuity are directly proportional to each other. If the Credit Amount is high, the Annuity Amount for the same will also be high.
- We are able to see an almost Linear graph. However, based on these 2 numeric features, we are not able to carry out Binary Classification with the help of Simple Logistic Regression. This means we need to carry out Feature Engineering on the same.

4.4 Fixing Null Values and Outliers

Since whatever implementations we are going to carry out on Train Data, we also need to carry out the same on Test Data, we will first combine both the datasets and then filter out NAs and

irrelevant classes from Categorical features and outliers from the Numerical columns, before we proceed with One Hot Encoding for categorical features and manual feature engineering.

4.4.1 Days_Birth

In [75]:
```
print("The shape of the train dataset = ",train_data.shape)
train_data['DAYS_BIRTH'].describe()
```

```
The shape of the train dataset =  (307511, 122)
```

Out[75]:
```
count    307511.000000
mean     -16036.995067
std        4363.988632
min      -25229.000000
25%      -19682.000000
50%      -15750.000000
75%      -12413.000000
max       -7489.000000
Name: DAYS_BIRTH, dtype: float64
```

In [76]:
```
print("The maximum age across all applicants (in years) = ",\
      - min(train_data['DAYS_BIRTH'].values)/365)

print("The minimum age across all applicants (in years) = ",\
      - max(train_data['DAYS_BIRTH'].values)/365)
```

```
The maximum age across all applicants (in years) =  69.12054794520547
The minimum age across all applicants (in years) =  20.517808219178082
```

Observations :

- This basically means that both the minimum as well as the maxium ages are admissible and there are no outliers present in the 'Days_Birth' column.

4.4.2 Days_Employed

The column 'Days_Employed' basically refers to the number ofdays before the loan application that the client started his/her first job, and since this is relative to the application, all values must be negative and there cannot be any positive value.

In [77]:
```
train_data['DAYS_EMPLOYED'].describe()
```

Out[77]:
```
count    307511.000000
mean      63815.045904
std      141275.766519
min      -17912.000000
25%       -2760.000000
50%       -1213.000000
75%        -289.000000
max      365243.000000
Name: DAYS_EMPLOYED, dtype: float64
```

In [78]:
```
print("The minimum employment across all applicants (in years) = ",\
       max(train_data['DAYS_EMPLOYED'].values)/365)

print("The maximum employment across all applicants (in years) = ",\
       min(train_data['DAYS_EMPLOYED'].values)/365)
```

```
The minimum employment across all applicants (in years) =  1000.6657534
246575
The maximum employment across all applicants (in years) =  -49.07397260
273972
```

Observations :

- As shown over here, the maximum time (of employment) in years is 1000 years, which is clearly an error, and we will remove all the positive values in the same column.

In [79]:
```
train_data.replace(max(train_data['DAYS_EMPLOYED'].values), np.nan, inp
```

```
lace=True)
```

4.4.3 Days_Registration

Again, Days_Registration also should be non-positive values and we will check the outliers in this case.

In [80]:
```
train_data['DAYS_REGISTRATION'].describe()
```

Out[80]:
```
count    307511.0
mean          NaN
std           NaN
min      -24672.0
25%       -7480.0
50%       -4504.0
75%       -2010.0
max           0.0
Name: DAYS_REGISTRATION, dtype: float64
```

In [81]:
```
print("The minimum days of registration across all applicants (in year
s) = ",\
        max(train_data['DAYS_EMPLOYED'].values)/365)

print("The maximum days of registration across all applicants (in year
s) = ",\
        min(train_data['DAYS_EMPLOYED'].values)/365)
```

```
The minimum days of registration across all applicants (in years) =  0.
0
The maximum days of registration across all applicants (in years) =  -4
9.07397260273972
```

Observations :

- This shows that both the minimum as well as the maximum days of registration are admissible and there are no outliers present in the 'Days_Registration' column.

4.4.4 Removing Outliers and Null Values

In [82]:
```
def fix_nulls_outliers(data):

    data['NAME_FAMILY_STATUS'].fillna('Data_Not_Available', inplace=Tru
e)
    data['NAME_HOUSING_TYPE'].fillna('Data_Not_Available', inplace=True
)

    data['FLAG_MOBIL'].fillna('Data_Not_Available', inplace=True)
    data['FLAG_EMP_PHONE'].fillna('Data_Not_Available', inplace=True)
    data['FLAG_CONT_MOBILE'].fillna('Data_Not_Available', inplace=True)
    data['FLAG_EMAIL'].fillna('Data_Not_Available', inplace=True)

    data['OCCUPATION_TYPE'].fillna('Data_Not_Available', inplace=True)

    #Replace NA with the most frequently occuring class for Count of Cl
ient Family Members
    data['CNT_FAM_MEMBERS'].fillna(data['CNT_FAM_MEMBERS'].value_counts
().idxmax(), \
                                           inplace=True)

    data.replace(max(data['DAYS_EMPLOYED'].values), np.nan, inplace=Tru
e)

    data['CODE_GENDER'].replace('XNA','M',inplace=True)
    #There are a total of 4 applicants with Gender provided as 'XNA'

    data['AMT_ANNUITY'].fillna(0, inplace=True)
    #A total of 36 datapoints are there where Annuity Amount is null.

    data['AMT_GOODS_PRICE'].fillna(0, inplace=True)
    #A total of 278 datapoints are there where Annuity Amount is null.
```

```
    data['NAME_TYPE_SUITE'].fillna('Unaccompanied', inplace=True)
    #Removing datapoints where 'Name_Type_Suite' is null.

    data['NAME_FAMILY_STATUS'].replace('Unknown','Married', inplace=Tru
e)
    #Removing datapoints where 'Name_Family_Status' is Unknown.

    data['OCCUPATION_TYPE'].fillna('Data_Not_Available', inplace=True)

    data['EXT_SOURCE_1'].fillna(0, inplace=True)
    data['EXT_SOURCE_2'].fillna(0, inplace=True)
    data['EXT_SOURCE_3'].fillna(0, inplace=True)

    return data
```

In [83]:
```
train_data = fix_nulls_outliers(train_data)
train_data.shape
```

Out[83]: (307511, 122)

4.5 Feature Engineering on Application Data

In [84]:
```
# Refer :- https://www.kaggle.com/willkoehrsen/start-here-a-gentle-intr
oduction#Feature-Engineering

def FE_application_data(data):

    data['CREDIT_INCOME_PERCENT'] = data['AMT_CREDIT'] / data['AMT_INCO
ME_TOTAL']
    data['ANNUITY_INCOME_PERCENT'] = data['AMT_ANNUITY'] / data['AMT_IN
COME_TOTAL']
    data['CREDIT_ANNUITY_PERCENT'] = data['AMT_CREDIT'] / data['AMT_ANN
UITY']

    data['FAMILY_CNT_INCOME_PERCENT'] = data['AMT_INCOME_TOTAL'] / data
['CNT_FAM_MEMBERS']
    data['CREDIT_TERM'] = data['AMT_ANNUITY'] / data['AMT_CREDIT']
```

```
    data['BIRTH_EMPLOYED_PERCENT'] = data['DAYS_EMPLOYED'] / data['DAYS
_BIRTH']
    data['CHILDREN_CNT_INCOME_PERCENT'] = data['AMT_INCOME_TOTAL']/data
['CNT_CHILDREN']

    data['CREDIT_GOODS_DIFF'] = data['AMT_CREDIT'] - data['AMT_GOODS_PR
ICE']
    data['EMPLOYED_REGISTRATION_PERCENT'] = data['DAYS_EMPLOYED'] / dat
a['DAYS_REGISTRATION']
    data['BIRTH_REGISTRATION_PERCENT'] = data['DAYS_BIRTH'] / data['DAY
S_REGISTRATION']
    data['ID_REGISTRATION_DIFF'] = data['DAYS_ID_PUBLISH'] - data['DAYS
_REGISTRATION']

    data['ANNUITY_LENGTH_EMPLOYED_PERCENT'] = data['CREDIT_TERM']/ data
['DAYS_EMPLOYED']

    data['AGE_LOAN_FINISH'] = data['DAYS_BIRTH']*(-1.0/365) + \
                          (data['AMT_CREDIT']/data['AMT_ANNUITY']) *(1.0
/12)
    # (This basically refers to the client's age when he/she finishes l
oan repayment)

    data['CAR_AGE_EMP_PERCENT'] = data['OWN_CAR_AGE']/data['DAYS_EMPLOY
ED']
    data['CAR_AGE_BIRTH_PERCENT'] = data['OWN_CAR_AGE']/data['DAYS_BIRT
H']
    data['PHONE_CHANGE_EMP_PERCENT'] = data['DAYS_LAST_PHONE_CHANGE']/d
ata['DAYS_EMPLOYED']
    data['PHONE_CHANGE_BIRTH_PERCENT'] = data['DAYS_LAST_PHONE_CHANGE']
/data['DAYS_BIRTH']

    income_by_contract = data[['AMT_INCOME_TOTAL', 'NAME_CONTRACT_TYPE'
]].groupby('NAME_CONTRACT_TYPE').median()['AMT_INCOME_TOTAL']
    data['MEDIAN_INCOME_CONTRACT_TYPE'] = data['NAME_CONTRACT_TYPE'].ma
p(income_by_contract)

    income_by_suite = data[['AMT_INCOME_TOTAL', 'NAME_TYPE_SUITE']].gro
upby('NAME_TYPE_SUITE').median()['AMT_INCOME_TOTAL']
    data['MEDIAN_INCOME_SUITE_TYPE'] = data['NAME_TYPE_SUITE'].map(inco
```

```
me_by_suite)

    income_by_housing = data[['AMT_INCOME_TOTAL', 'NAME_HOUSING_TYPE']]
.groupby('NAME_HOUSING_TYPE').median()['AMT_INCOME_TOTAL']
    data['MEDIAN_INCOME_HOUSING_TYPE'] = data['NAME_HOUSING_TYPE'].map(
income_by_housing)

    income_by_org = data[['AMT_INCOME_TOTAL', 'ORGANIZATION_TYPE']].gro
upby('ORGANIZATION_TYPE').median()['AMT_INCOME_TOTAL']
    data['MEDIAN_INCOME_ORG_TYPE'] = data['ORGANIZATION_TYPE'].map(inco
me_by_org)

    income_by_occu = data[['AMT_INCOME_TOTAL', 'OCCUPATION_TYPE']].grou
pby('OCCUPATION_TYPE').median()['AMT_INCOME_TOTAL']
    data['MEDIAN_INCOME_OCCU_TYPE'] = data['OCCUPATION_TYPE'].map(incom
e_by_occu)

    income_by_education = data[['AMT_INCOME_TOTAL', 'NAME_EDUCATION_TYP
E']].groupby('NAME_EDUCATION_TYPE').median()['AMT_INCOME_TOTAL']
    data['MEDIAN_INCOME_EDU_TYPE'] = data['NAME_EDUCATION_TYPE'].map(in
come_by_education)

    data['ORG_TYPE_INCOME_PERCENT'] = data['MEDIAN_INCOME_ORG_TYPE']/da
ta['AMT_INCOME_TOTAL']
    data['OCCU_TYPE_INCOME_PERCENT'] = data['MEDIAN_INCOME_OCCU_TYPE']/
data['AMT_INCOME_TOTAL']
    data['EDU_TYPE_INCOME_PERCENT'] = data['MEDIAN_INCOME_EDU_TYPE']/da
ta['AMT_INCOME_TOTAL']

    data= data.drop(['FLAG_DOCUMENT_2','FLAG_DOCUMENT_4','FLAG_DOCUMENT
_5','FLAG_DOCUMENT_6','FLAG_DOCUMENT_7',
    'FLAG_DOCUMENT_8','FLAG_DOCUMENT_9','FLAG_DOCUMENT_10', 'FLAG_DOCUM
ENT_11','FLAG_DOCUMENT_12','FLAG_DOCUMENT_13',
    'FLAG_DOCUMENT_14','FLAG_DOCUMENT_15','FLAG_DOCUMENT_16','FLAG_DOCU
MENT_17','FLAG_DOCUMENT_18','FLAG_DOCUMENT_19',
    'FLAG_DOCUMENT_20','FLAG_DOCUMENT_21'],axis=1)

    cat_col = [category for category in data.columns if data[category].
dtype == 'object']
    data = pd.get_dummies(data, columns= cat_col)
```

```
    return data
```

In [85]:
```
train_data_temp_1 = FE_application_data(train_data)
print(train_data_temp_1.shape)
```

```
(307511, 252)
```

Function to Carry out One Hot Encoding for Categorical Features :-

In [86]:
```
#Refer :- https://www.kaggle.com/jsaguiar/lightgbm-7th-place-solution
#one hot encode the categorical data

def one_hot_encode(df):
    original_columns = list(df.columns)
    categories = [cat for cat in df.columns if df[cat].dtype == 'objec
t']
    df = pd.get_dummies(df, columns= categories, dummy_na= True) #one_h
ot_encode the categorical features
    categorical_columns = [cat for cat in df.columns if cat not in orig
inal_columns]
    return df, categorical_columns
```

5. Bureau Dataset and Bureau Balance Datasets

5.1. Basic Overview of the Bureau Data

In [87]:
```
bureau_data = reduce_memory_usage(pd.read_csv('home-credit-default-ris
k/bureau.csv'))
print('Number of data points : ', bureau_data.shape[0])
print('Number of features : ', bureau_data.shape[1])
bureau_data.head()
```

```
Memory usage of dataframe is 222.62 MB
```

```
Memory usage after optimization is: 112.95 MB
Decreased by 49.3%
Number of data points :  1716428
Number of features :  17
```

Out[87]:

	SK_ID_CURR	SK_ID_BUREAU	CREDIT_ACTIVE	CREDIT_CURRENCY	DAYS_CREDIT	CREDIT_
0	215354	5714462	Closed	currency 1	-497	
1	215354	5714463	Active	currency 1	-208	
2	215354	5714464	Active	currency 1	-203	
3	215354	5714465	Active	currency 1	-203	
4	215354	5714466	Active	currency 1	-629	

In [88]:
```
bureau_data.columns
```

Out[88]:
```
Index(['SK_ID_CURR', 'SK_ID_BUREAU', 'CREDIT_ACTIVE', 'CREDIT_CURRENC
Y',
       'DAYS_CREDIT', 'CREDIT_DAY_OVERDUE', 'DAYS_CREDIT_ENDDATE',
       'DAYS_ENDDATE_FACT', 'AMT_CREDIT_MAX_OVERDUE', 'CNT_CREDIT_PROLO
NG',
       'AMT_CREDIT_SUM', 'AMT_CREDIT_SUM_DEBT', 'AMT_CREDIT_SUM_LIMIT',
       'AMT_CREDIT_SUM_OVERDUE', 'CREDIT_TYPE', 'DAYS_CREDIT_UPDATE',
       'AMT_ANNUITY'],
      dtype='object')
```

5.2 Analysis of Bureau Data

5.2.1 Univariate Analysis: Credit_Active

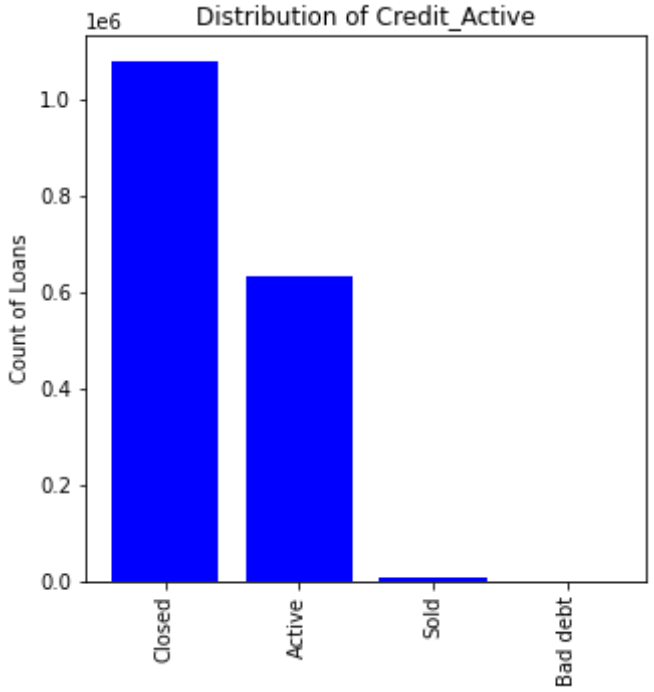
In [89]:
```
from collections import Counter
```

```
credit_active_counter = Counter()
for word in bureau_data['CREDIT_ACTIVE'].values:
    credit_active_counter.update(word.split(','))

print(credit_active_counter)
```

```
Counter({'Closed': 1079273, 'Active': 630607, 'Sold': 6527, 'Bad debt':
21})
```

In [90]:

```
# dict sort by value python: https://stackoverflow.com/a/613218/4084039
status_dict = dict(credit_active_counter)
sorted_status_dict = dict(sorted(status_dict.items(), key=lambda kv: kv
[1], reverse=True))


ind = np.arange(len(sorted_status_dict))
plt.figure(figsize=(5,5))
p1 = plt.bar(ind, list(sorted_status_dict.values()),color='blue')

plt.ylabel('Count of Loans')
plt.title('Distribution of Credit_Active')
plt.xticks(ind, list(sorted_status_dict.keys()),rotation=90)
plt.show()
```



Observations :

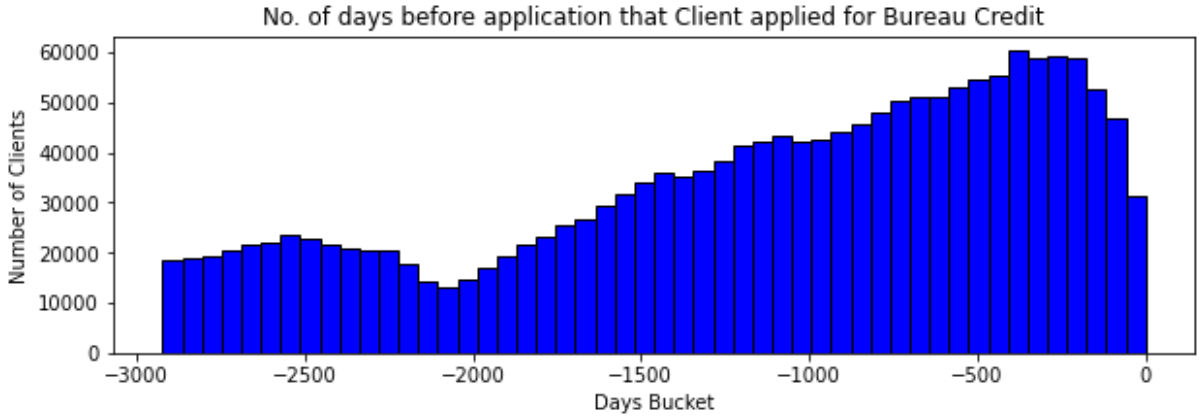
- Most of the applications in the Bureau Data are closed, which is followed by the status being Active.
- There are very few loans that are 'Sold' or considered to be 'Bad Debt'.

5.2.2 Univariate Analysis: Days_Credit

The Days_Credit Column refers to the length of time (in days) before the application that the client applied for Bureau Credit.

In [91]:
```
plt.figure(figsize=(10,3))
```

```
plt.hist(bureau_data['DAYS_CREDIT'].values, bins=50, edgecolor='black',
color='blue')
plt.title('No. of days before application that Client applied for Burea
u Credit')
plt.xlabel('Days Bucket')
plt.ylabel('Number of Clients')
plt.show()
```



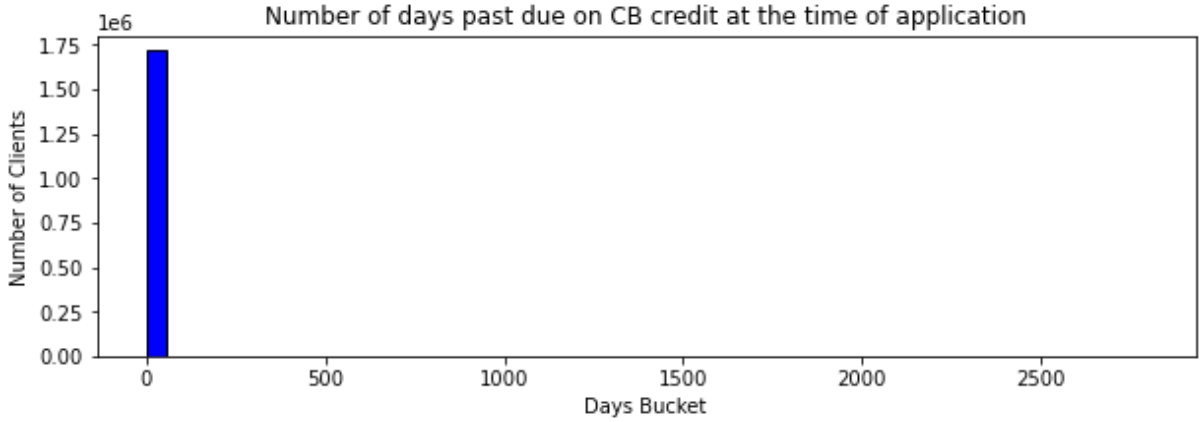
Observations :

- Most of the clients applied for Bureau Credit less than 500 days before the date of loan application.

5.2.3 Univariate Analysis: Credit_Day_Overdue

In [92]:
```
plt.figure(figsize=(10,3))
plt.hist(bureau_data['CREDIT_DAY_OVERDUE'].values, bins=50, edgecolor=
'black', color='blue')
plt.title('Number of days past due on CB credit at the time of applicat
ion')
plt.xlabel('Days Bucket')
```

```
plt.ylabel('Number of Clients')
plt.show()
```



Note :

This shows that most of the clients have a low DPD (close to 0) on CB Credit at the Time of their application since the histogram is very peaked near 0. To analyse this further, we will have a look at the respective percentile values as shown below.

In [93]:
```
y = PrettyTable()
y.field_names = ["Percentile", "No. of days past due on CB Credit"]

for i in range(0,101,10):
    y.add_row([i,np.percentile(bureau_data['CREDIT_DAY_OVERDUE'].values
,i)])
print(y)
```

```
+------------+-----------------------------------+
| Percentile | No. of days past due on CB Credit |
+------------+-----------------------------------+
|     0      |                0.0                |
|     10     |                0.0                |
|     20     |                0.0                |
|     30     |                0.0                |
|     40     |                0.0                |
```

```
|     40     |                0.0                 |
|     50     |                0.0                 |
|     60     |                0.0                 |
|     70     |                0.0                 |
|     80     |                0.0                 |
|     90     |                0.0                 |
|    100     |               2792.0               |
+------------+------------------------------------+
```

Observations :

- There is 0 DPD till the 90th percentile as well, whereas the 100th percentile is a value = 2792.
- We will further zoom into the values between the 99th and 100th percentiles.

In [94]:
```
z = PrettyTable()
z.field_names = ["Percentile", "No. of days past due on CB Credit"]

for i in np.arange(99,100,0.1):
    z.add_row(np.round([i,np.percentile(bureau_data['CREDIT_DAY_OVERDUE'].values,i)],2))
print(z)
```

```
+------------+------------------------------------+
| Percentile | No. of days past due on CB Credit  |
+------------+------------------------------------+
|    99.0    |                0.0                 |
|    99.1    |                0.0                 |
|    99.2    |                0.0                 |
|    99.3    |                0.0                 |
|    99.4    |                0.0                 |
|    99.5    |                0.0                 |
|    99.6    |                0.0                 |
|    99.7    |                0.0                 |
|    99.8    |                13.0                |
|    99.9    |               52.57                |
+------------+------------------------------------+
```

Observations :

- We can see from here that only the Top 0.3 percentile of values over here are non-zeroes.

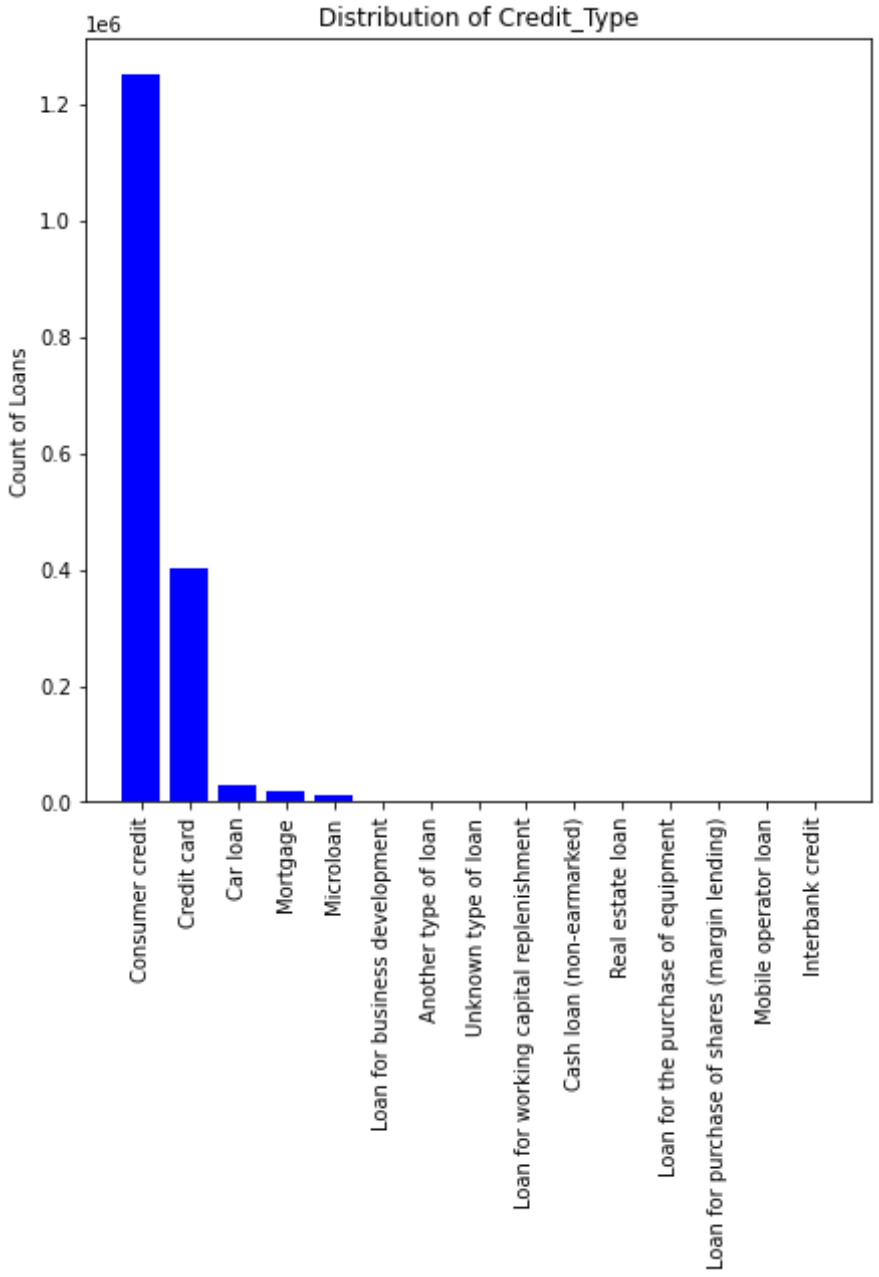
In [95]:
```
bureau_data[bureau_data['CREDIT_DAY_OVERDUE']>0].shape[0]
#This basically shows that there are a total of 4217 datapoints where t
he 'Credit_Day_Overdue' value is greater
#than 0.
```

Out[95]: 4217

5.2.4 Univariate Analysis: Credit_Type

In [96]:
```
credit_type_counter = Counter()
for type in bureau_data['CREDIT_TYPE'].values:
    credit_type_counter.update(type.split(','))
```

In [97]:
```
# dict sort by value python: https://stackoverflow.com/a/613218/4084039
type_dict = dict(credit_type_counter)
sorted_type_dict = dict(sorted(type_dict.items(), key=lambda kv: kv[1],
reverse=True))

ind_2 = np.arange(len(sorted_type_dict))
plt.figure(figsize=(7,7))
p1 = plt.bar(ind_2, list(sorted_type_dict.values()),color='blue')

plt.ylabel('Count of Loans')
plt.title('Distribution of Credit_Type')
plt.xticks(ind_2, list(sorted_type_dict.keys()),rotation=90)
plt.show()
```

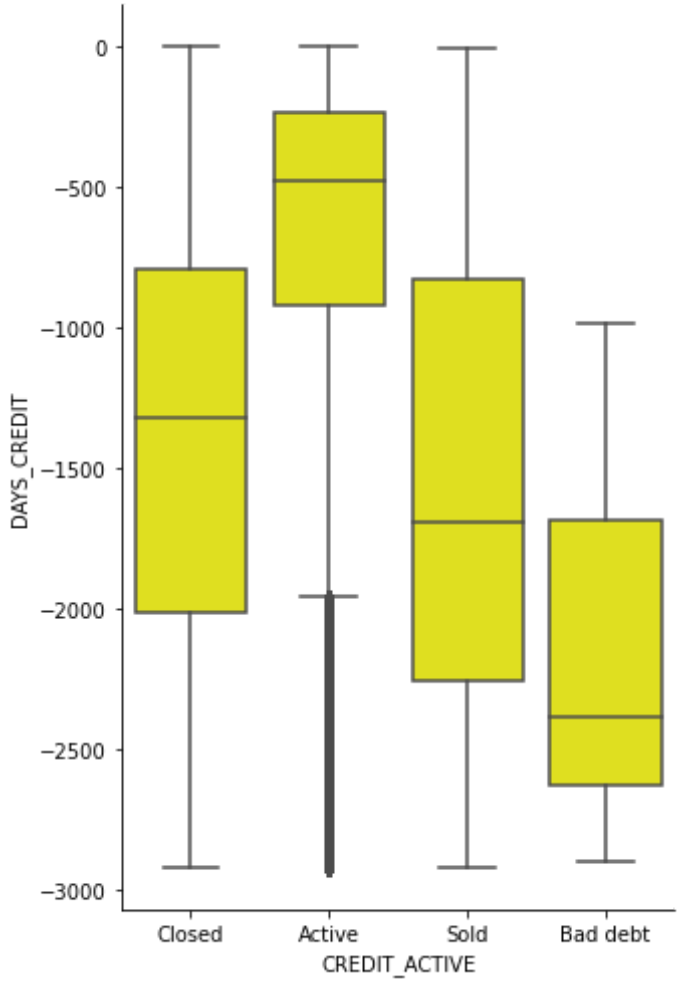


Observations :

- Consumer Credit and Credit Cards are the mostly registered credit types in the Credit Bureau.

5.2.5 Bivariate Analysis: Credit_Active vs Days_Credit

In [98]:
```
sns.catplot(x="CREDIT_ACTIVE", y="DAYS_CREDIT",\
            data=bureau_data,color = "yellow",kind="box", height=7, asp
ect=.7);
```



Observations :

- When the Credit Status is Active, it means that the corresponding 'Days_Credit' ie. number of days before Application, the median value is approx. 500 days.

5.3. Basic Overview of the Bureau Balance Data

In [99]:
```
bureau_balance = reduce_memory_usage(pd.read_csv('home-credit-default-risk/bureau_balance.csv'))
print('Number of data points : ', bureau_balance.shape[0])
print('Number of features : ', bureau_balance.shape[1])
bureau_balance.head()
```

```
Memory usage of dataframe is 624.85 MB
Memory usage after optimization is: 338.46 MB
Decreased by 45.8%
Number of data points :  27299925
Number of features :  3
```

Out[99]:

	SK_ID_BUREAU	MONTHS_BALANCE	STATUS
0	5715448	0	C
1	5715448	-1	C
2	5715448	-2	C
3	5715448	-3	C
4	5715448	-4	C

In [100]:
```
bureau_balance.columns
```

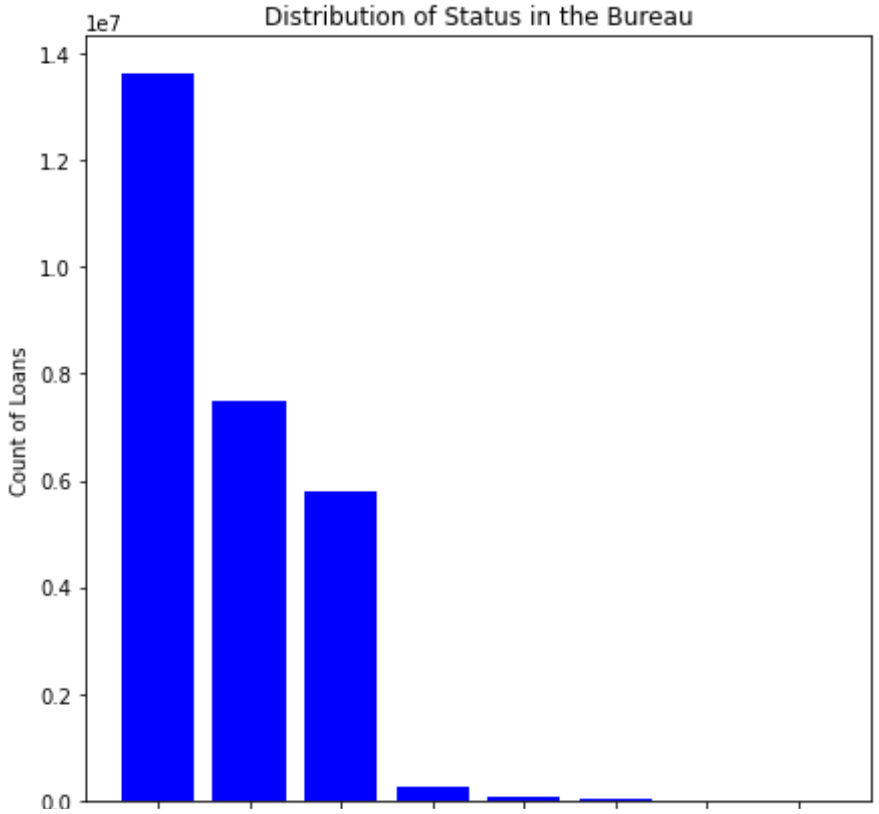
Out[100]:
```
Index(['SK_ID_BUREAU', 'MONTHS_BALANCE', 'STATUS'], dtype='object')
```

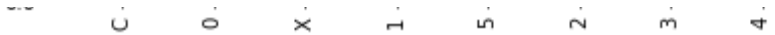
5.4. Bureau Balance Data Analysis

5.4.1 Univariate Analysis: Status

In [101]:
```
bureau_status_counter = Counter()
for status in bureau_balance['STATUS'].values:
    bureau_status_counter.update(status.split(','))
```

In [102]:
```
# dict sort by value python: https://stackoverflow.com/a/613218/4084039
status_dict = dict(bureau_status_counter)
sorted_status_dict = dict(sorted(status_dict.items(), key=lambda kv: kv
[1], reverse=True))


ind_3 = np.arange(len(sorted_status_dict))
plt.figure(figsize=(7,7))
p1 = plt.bar(ind_3, list(sorted_status_dict.values()),color='blue')

plt.ylabel('Count of Loans')
plt.title('Distribution of Status in the Bureau')
plt.xticks(ind_3, list(sorted_status_dict.keys()),rotation=90)
plt.show()
```





Observations :

- Most of the loans are Closed in the Credit Bureau, which is followed by clients with 0 DPD and then by applicants whose status is unknown.
- We can conclude that there are very few annuity defaulters in the data.

5.5 Feature Engineering on Bureau Data and Bureau Balance Data

In [103]:
```
def generate_credit_type_code(x):
    if x == 'Closed':
        y = 0
    elif x=='Active':
        y = 1
    else:
        y = 2
    return y


def FE_bureau_data_1(bureau_data):

    bureau_data['CREDIT_DURATION'] = -bureau_data['DAYS_CREDIT'] + bure
au_data['DAYS_CREDIT_ENDDATE']
    bureau_data['ENDDATE_DIFF'] = bureau_data['DAYS_CREDIT_ENDDATE'] -
bureau_data['DAYS_ENDDATE_FACT']
    bureau_data['UPDATE_DIFF'] = bureau_data['DAYS_CREDIT_ENDDATE'] - b
ureau_data['DAYS_CREDIT_UPDATE']
    bureau_data['DEBT_PERCENTAGE'] = bureau_data['AMT_CREDIT_SUM'] / bu
reau_data['AMT_CREDIT_SUM_DEBT']
    bureau_data['DEBT_CREDIT_DIFF'] = bureau_data['AMT_CREDIT_SUM'] - b
ureau_data['AMT_CREDIT_SUM_DEBT']
    bureau_data['CREDIT_TO_ANNUITY_RATIO'] = bureau_data['AMT_CREDIT_SU
M'] / bureau_data['AMT_ANNUITY']
```

```
    bureau_data['DEBT_TO_ANNUITY_RATIO'] = bureau_data['AMT_CREDIT_SUM_
DEBT'] / bureau_data['AMT_ANNUITY']
    bureau_data['CREDIT_OVERDUE_DIFF'] = bureau_data['AMT_CREDIT_SUM']
- bureau_data['AMT_CREDIT_SUM_OVERDUE']

    #Refer :- https://www.kaggle.com/c/home-credit-default-risk/discuss
ion/57750
    #Calculating the Number of Past Loans for each Customer
    no_loans_per_customer = bureau_data[['SK_ID_CURR', 'SK_ID_BUREAU']]
.groupby(by = \
                                                                ['S
K_ID_CURR'])['SK_ID_BUREAU'].count()
    no_loans_per_customer = no_loans_per_customer.reset_index().rename(
columns={'SK_ID_BUREAU': 'CUSTOMER_LOAN_COUNT'})
    bureau_data = bureau_data.merge(no_loans_per_customer, on='SK_ID_CU
RR', how='left')

    #Calculating the Past Credit Types per Customer
    credit_types_per_customer = bureau_data[['SK_ID_CURR','CREDIT_TYPE'
]].groupby(by=['SK_ID_CURR'])['CREDIT_TYPE'].nunique()
    credit_types_per_customer = credit_types_per_customer.reset_index()
.rename(columns={'CREDIT_TYPE':'CUSTOMER_CREDIT_TYPES'})
    bureau_data = bureau_data.merge(credit_types_per_customer, on='SK_I
D_CURR',how='left')

    #Average Loan Type per Customer
    bureau_data['AVG_LOAN_TYPE'] = bureau_data['CUSTOMER_LOAN_COUNT']/b
ureau_data['CUSTOMER_CREDIT_TYPES']

    bureau_data['CREDIT_TYPE_CODE'] = bureau_data.apply(lambda x:\
                                        generate_credit_type_code(x.CRE
DIT_ACTIVE), axis=1)

    customer_credit_code_mean = bureau_data[['SK_ID_CURR','CREDIT_TYPE_
CODE']].groupby(by=['SK_ID_CURR'])['CREDIT_TYPE_CODE'].mean()
    customer_credit_code_mean.reset_index().rename(columns={'CREDIT_TYP
E_CODE':'CUSTOMER_CREDIT_CODE_MEAN'})
    bureau_data = bureau_data.merge(customer_credit_code_mean, on='SK_I
D_CURR', how='left')
```

```
    #Computing the Ratio of Total Customer Credit and the Total Custome
r Debt
    bureau_data['AMT_CREDIT_SUM'] = bureau_data['AMT_CREDIT_SUM'].filln
a(0)
    bureau_data['AMT_CREDIT_SUM_DEBT'] = bureau_data['AMT_CREDIT_SUM_DE
BT'].fillna(0)
    bureau_data['AMT_ANNUITY'] = bureau_data['AMT_ANNUITY'].fillna(0)

    credit_sum_customer = bureau_data[['SK_ID_CURR','AMT_CREDIT_SUM']].
groupby(by=\

['SK_ID_CURR'])['AMT_CREDIT_SUM'].sum()
    credit_sum_customer = credit_sum_customer.reset_index().rename(colu
mns={'AMT_CREDIT_SUM':'TOTAL_CREDIT_SUM'})
    bureau_data = bureau_data.merge(credit_sum_customer, on='SK_ID_CUR
R', how='left')

    credit_debt_sum_customer = bureau_data[['SK_ID_CURR','AMT_CREDIT_SU
M_DEBT']].groupby(by=\

['SK_ID_CURR'])['AMT_CREDIT_SUM_DEBT'].sum()
    credit_debt_sum_customer = credit_debt_sum_customer.reset_index().r
ename(columns={'AMT_CREDIT_SUM_DEBT':'TOTAL_DEBT_SUM'})
    bureau_data = bureau_data.merge(credit_debt_sum_customer, on='SK_ID
_CURR', how='left')
    bureau_data['CREDIT_DEBT_RATIO'] = bureau_data['TOTAL_CREDIT_SUM']/
bureau_data['TOTAL_DEBT_SUM']

    return bureau_data
```

In [104]:
```
bureau_data_fe = FE_bureau_data_1(bureau_data)

#One Hot Encoding the Bureau Datasets
bureau_data, bureau_data_columns = one_hot_encode(bureau_data_fe)
bureau_balance, bureau_balance_columns = one_hot_encode(bureau_balance)
```

In [105]:
```
def FE_bureau_data_2(bureau_data,bureau_balance,bureau_data_columns,bur
eau_balance_columns):
```

```
    bureau_balance_agg = {'MONTHS_BALANCE': ['min','max','mean','size'
]}

    for column in bureau_balance_columns:
        bureau_balance_agg[column] = ['min','max','mean','size']
        bureau_balance_final_agg = bureau_balance.groupby('SK_ID_BUREA
U').agg(bureau_balance_agg)

    col_list_1 =[]

    for col in bureau_balance_final_agg.columns.tolist():
        col_list_1.append(col[0] + "_" + col[1].upper())

    bureau_balance_final_agg.columns = pd.Index(col_list_1)
    bureau_data_balance = bureau_data.join(bureau_balance_final_agg, ho
w='left', on='SK_ID_BUREAU')
    bureau_data_balance.drop(['SK_ID_BUREAU'], axis=1, inplace= True)

    del bureau_balance_final_agg
    gc.collect()


    numerical_agg = {'AMT_CREDIT_SUM_DEBT': ['mean', 'sum'],'AMT_CREDIT
_SUM_OVERDUE': ['mean','sum'],
        'DAYS_CREDIT': ['mean', 'var'],'DAYS_CREDIT_UPDATE': ['mean','m
in'],'CREDIT_DAY_OVERDUE': ['mean','min'],
        'DAYS_CREDIT_ENDDATE': ['mean'],'CNT_CREDIT_PROLONG': ['sum'],
'MONTHS_BALANCE_SIZE': ['mean', 'sum'],
        'AMT_CREDIT_SUM_LIMIT': ['mean', 'sum'],'AMT_CREDIT_MAX_OVERDU
E': ['mean','max'],
        'AMT_ANNUITY': ['max', 'mean','sum'],'AMT_CREDIT_SUM': ['mean',
'sum','max']
      }
    categorical_agg = {}

    for col in bureau_data_columns:
        categorical_agg[col] = ['mean']
        categorical_agg[col] = ['max']
```

```
    for col in bureau_balance_columns:
        categorical_agg[col + "_MEAN"] = ['mean']
        categorical_agg[col + "_MIN"] = ['min']
        categorical_agg[col + "_MAX"] = ['max']

    bureau_data_balance_2 = bureau_data_balance.groupby('SK_ID_CURR').a
gg({**numerical_agg,\

**categorical_agg})
    col_list_2=[]

    for col in bureau_data_balance_2.columns.tolist():
        col_list_2.append('BUREAU_'+col[0]+'_'+col[1])
    bureau_data_balance_2.columns = pd.Index(col_list_2)


    bureau_data_balance_3 = bureau_data_balance[bureau_data_balance['CR
EDIT_ACTIVE_Active'] == 1]
    bureau_data_balance_3_agg = bureau_data_balance_3.groupby('SK_ID_CU
RR').agg(numerical_agg)

    col_list_3=[]
    for col in bureau_data_balance_3_agg.columns.tolist():
        col_list_3.append('A_'+col[0]+'_'+col[1].upper())

    bureau_data_balance_3_agg.columns = pd.Index(col_list_3)
    b3_final = bureau_data_balance_2.join(bureau_data_balance_3_agg, ho
w='left', \
                                          on='SK_ID_CURR')

    bureau_data_balance_4 = bureau_data_balance[bureau_data_balance['CR
EDIT_ACTIVE_Closed'] == 1]
    bureau_data_balance_4_agg = bureau_data_balance_4.groupby('SK_ID_CU
RR').agg(numerical_agg)
    col_list_4 =[]

    for col in bureau_data_balance_4_agg.columns.tolist():
        col_list_4.append('C_'+col[0]+'_'+col[1].upper())

    bureau_data_balance_4_agg.columns = pd.Index(col_list_4)
```

```
    bureau_data_balance_final = bureau_data_balance_2.join(bureau_data_
balance_4_agg, \
                                                           how='left', on='SK_
ID_CURR')

    del bureau_data_balance_3, bureau_data_balance_4_agg
    gc.collect()

    return bureau_data_balance_final
```

In [106]:
```
bureau_data_balance_final = FE_bureau_data_2(bureau_data, bureau_balanc
e,bureau_data_columns,\
                                             bureau_balance_columns)
train_data_temp_2 = train_data_temp_1.join(bureau_data_balance_final, h
ow='left', \
                                           on='SK_ID_CURR')

del bureau_data_balance_final
gc.collect()

train_data_temp_2.shape
```

Out[106]:
```
(307511, 353)
```

6. Previous Application Dataset

6.1. Basic Overview of the Train Data

In [107]:
```
previous_application = reduce_memory_usage(pd.read_csv('home-credit-def
ault-risk/previous_application.csv'))
print('Number of data points : ', previous_application.shape[0])
print('Number of features : ', previous_application.shape[1])
previous_application.head()
```

```
Memory usage of dataframe is 471.48 MB
Memory usage after optimization is: 309.01 MB
```

```
Decreased by 34.5%
Number of data points :  1670214
Number of features :  37
```

Out[107]:

	SK_ID_PREV	SK_ID_CURR	NAME_CONTRACT_TYPE	AMT_ANNUITY	AMT_APPLICATION	AMT
0	2030495	271877	Consumer loans	1730.430054	17145.0	
1	2802425	108129	Cash loans	25188.615234	607500.0	
2	2523466	122040	Cash loans	15060.735352	112500.0	
3	2819243	176158	Cash loans	47041.335938	450000.0	
4	1784265	202054	Cash loans	31924.394531	337500.0	

5 rows × 37 columns

In [108]:
```
previous_application.columns
```

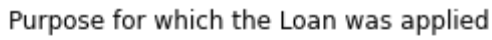
Out[108]:
```
Index(['SK_ID_PREV', 'SK_ID_CURR', 'NAME_CONTRACT_TYPE', 'AMT_ANNUITY',
       'AMT_APPLICATION', 'AMT_CREDIT', 'AMT_DOWN_PAYMENT', 'AMT_GOODS_
PRICE',
       'WEEKDAY_APPR_PROCESS_START', 'HOUR_APPR_PROCESS_START',
       'FLAG_LAST_APPL_PER_CONTRACT', 'NFLAG_LAST_APPL_IN_DAY',
       'RATE_DOWN_PAYMENT', 'RATE_INTEREST_PRIMARY',
       'RATE_INTEREST_PRIVILEGED', 'NAME_CASH_LOAN_PURPOSE',
       'NAME_CONTRACT_STATUS', 'DAYS_DECISION', 'NAME_PAYMENT_TYPE',
       'CODE_REJECT_REASON', 'NAME_TYPE_SUITE', 'NAME_CLIENT_TYPE',
       'NAME_GOODS_CATEGORY', 'NAME_PORTFOLIO', 'NAME_PRODUCT_TYPE',
       'CHANNEL_TYPE', 'SELLERPLACE_AREA', 'NAME_SELLER_INDUSTRY',
       'CNT_PAYMENT', 'NAME_YIELD_GROUP', 'PRODUCT_COMBINATION',
       'DAYS_FIRST_DRAWING', 'DAYS_FIRST_DUE', 'DAYS_LAST_DUE_1ST_VERSI
ON',
       'DAYS_LAST_DUE', 'DAYS_TERMINATION', 'NFLAG_INSURED_ON_APPROVA
L'],
      dtype='object')
```

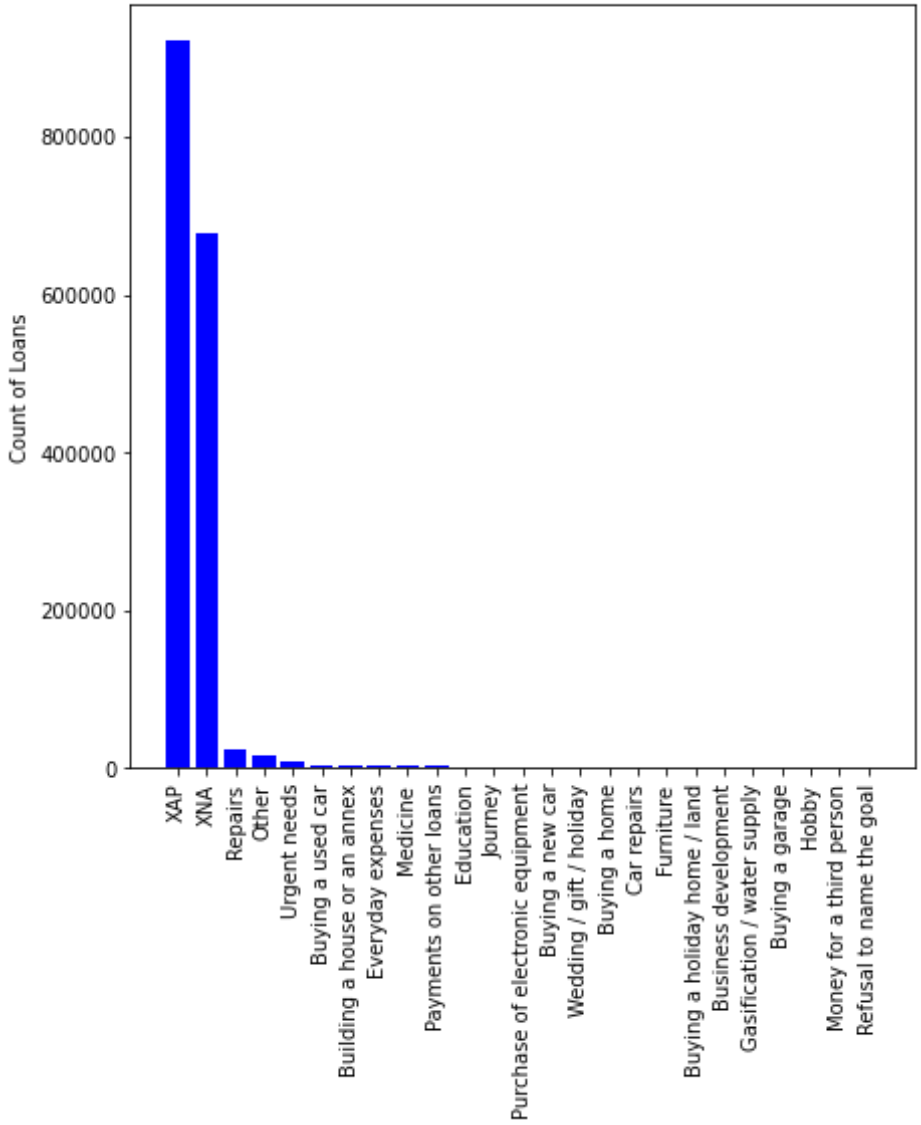
6.2 Previous Application Data Analysis

6.2.1 Univariate Analysis: Name_Cash_Loan_Purpose

In [109]:
```
application_loan_purpose = Counter()
for purpose in previous_application['NAME_CASH_LOAN_PURPOSE'].values:
    application_loan_purpose.update(purpose.split(','))
```

In [110]:
```
# dict sort by value python: https://stackoverflow.com/a/613218/4084039
purpose_dict = dict(application_loan_purpose)
sorted_purpose_dict = dict(sorted(purpose_dict.items(), key=lambda kv:
kv[1], reverse=True))

ind_4 = np.arange(len(sorted_purpose_dict))
plt.figure(figsize=(7,7))
p1 = plt.bar(ind_4, list(sorted_purpose_dict.values()),color='blue')

plt.ylabel('Count of Loans')
plt.title('Purpose for which the Loan was applied')
plt.xticks(ind_4, list(sorted_purpose_dict.keys()),rotation=90)
plt.show()
```





Observations :

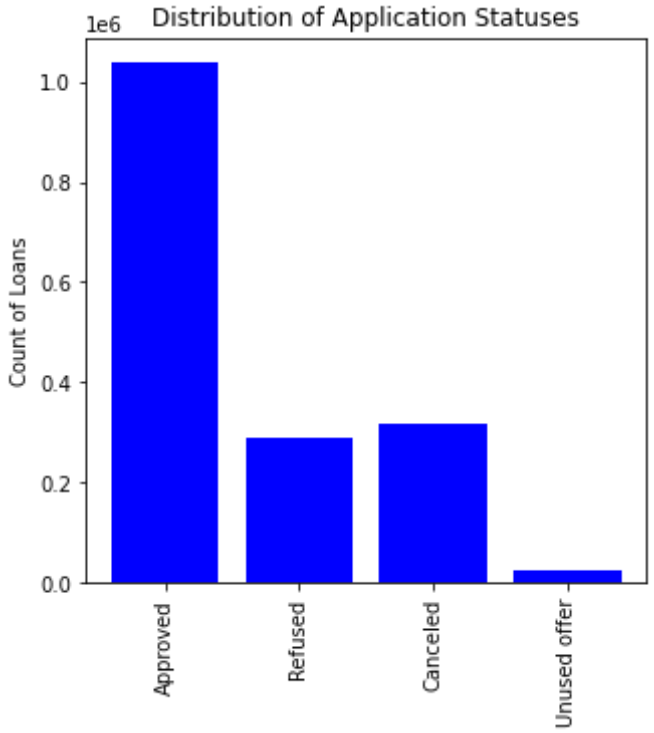
- The purpose for most of the Loan Applications is XAP, which is followed by XNA. However, the definition of these terms is not provided in the columns_description.csv.

- This may mean that the loan application purpose was not shared by the applicant, though we cannot be sure.

6.2.2 Univariate Analysis: Name_Contract_Status

In [111]:
```
application_contract_status = Counter()
for status in previous_application['NAME_CONTRACT_STATUS'].values:
    application_contract_status.update(status.split(','))
```

In [112]:
```
# dict sort by value python: https://stackoverflow.com/a/613218/4084039
contract_status_dict = dict(application_contract_status)
sorted_contract_status_dict = dict(sorted(contract_status_dict.items(),
\
                                          key=lambda kv: kv[1], reverse=T
rue))

ind_5 = np.arange(len(contract_status_dict))
plt.figure(figsize=(5,5))
p1 = plt.bar(ind_5, list(contract_status_dict.values()),color='blue')

plt.ylabel('Count of Loans')
plt.title('Distribution of Application Statuses')
plt.xticks(ind_5, list(contract_status_dict.keys()),rotation=90)
plt.show()
```



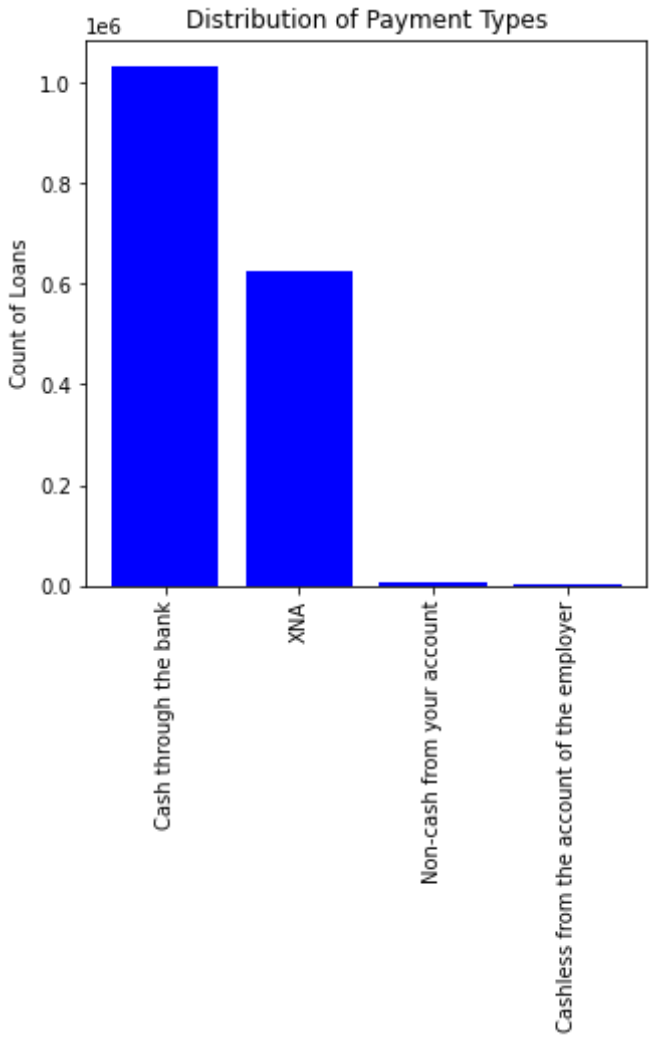
Observations :

- Most of the previous applications for the clients were approved.
- This is followed by applications that were cancelled and refused.
- There were very few applications that were approved but the loans were unused by the applicant.

6.2.3 Univariate Analysis: Name_Payment_Type

In [113]:
```
application_payment_type = Counter()
```

```
for type in previous_application['NAME_PAYMENT_TYPE'].values:
    application_payment_type.update(type.split(','))

# dict sort by value python: https://stackoverflow.com/a/613218/4084039
payment_type_dict = dict(application_payment_type)
sorted_payment_type_dict = dict(sorted(payment_type_dict.items(), \
                                       key=lambda kv: kv[1], reverse=T
rue))

ind_6 = np.arange(len(payment_type_dict))
plt.figure(figsize=(5,5))
p1 = plt.bar(ind_6, list(payment_type_dict.values()),color='blue')

plt.ylabel('Count of Loans')
plt.title('Distribution of Payment Types')
plt.xticks(ind_6, list(payment_type_dict.keys()),rotation=90)
plt.show()
```



Observations :

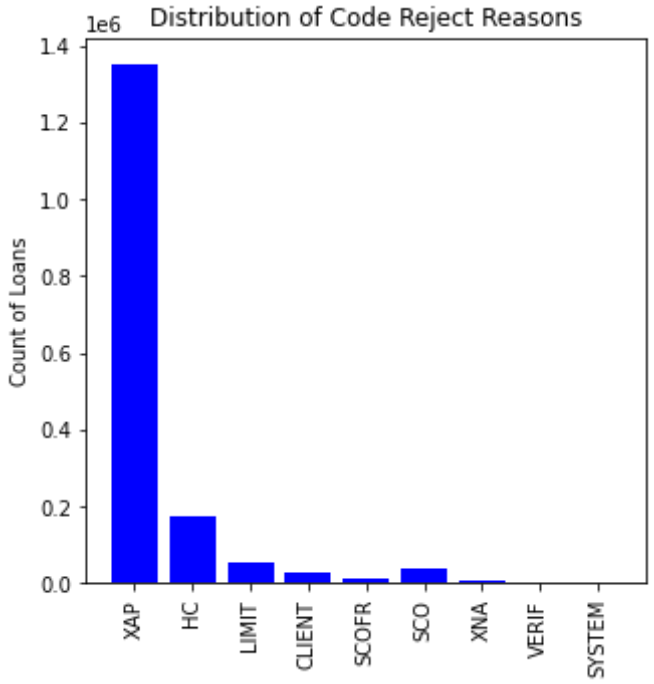
- The Payment Type basically refers to the Payment Method that the client chose to pay for the previous application, and as we can see here, most of the clients chose to pay via Cash

through the Bank for the same.

- This is followed by people whose payment type is XNA.

6.2.4 Univariate Analysis: Code_Reject_Reason

In [114]:
```
application_code_reject_reason = Counter()
for reason in previous_application['CODE_REJECT_REASON'].values:
    application_code_reject_reason.update(reason.split(','))

# dict sort by value python: https://stackoverflow.com/a/613218/4084039
code_reject_reason_dict = dict(application_code_reject_reason)
sorted_code_reject_reason_dict = dict(sorted(code_reject_reason_dict.it
ems(), \
                                             key=lambda kv: kv[1], reverse=T
rue))

ind_7 = np.arange(len(code_reject_reason_dict))
plt.figure(figsize=(5,5))
p1 = plt.bar(ind_7, list(code_reject_reason_dict.values()),color='blue'
)

plt.ylabel('Count of Loans')
plt.title('Distribution of Code Reject Reasons')
plt.xticks(ind_7, list(code_reject_reason_dict.keys()),rotation=90)
plt.show()
```



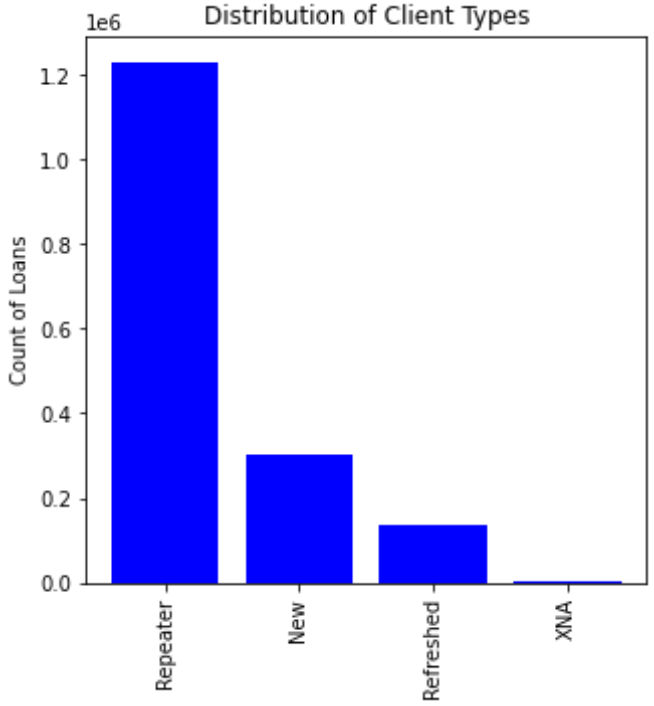
Observations :

- Code_Reject_Reason basically refers to the reason why the previous loan application of the client was rejected by the bank. As can be seen from here, in most of the cases XAP, was the reason provided. (Not Applicable)
- This is followed by HC as the second most prominent reason.

6.2.5 Univariate Analysis: Name_Client_Type

In [115]:
```
application_client_type = Counter()
for type in previous_application['NAME_CLIENT_TYPE'].values:
    application_client_type.update(type.split(','))
```

```
# dict sort by value python: https://stackoverflow.com/a/613218/4084039
client_type_dict = dict(application_client_type)
sorted_client_type_dict = dict(sorted(client_type_dict.items(), \
                                      key=lambda kv: kv[1], reverse=T
rue))

ind_8 = np.arange(len(client_type_dict))
plt.figure(figsize=(5,5))
p1 = plt.bar(ind_8, list(client_type_dict.values()),color='blue')

plt.ylabel('Count of Loans')
plt.title('Distribution of Client Types')
plt.xticks(ind_8, list(client_type_dict.keys()),rotation=90)
plt.show()
```

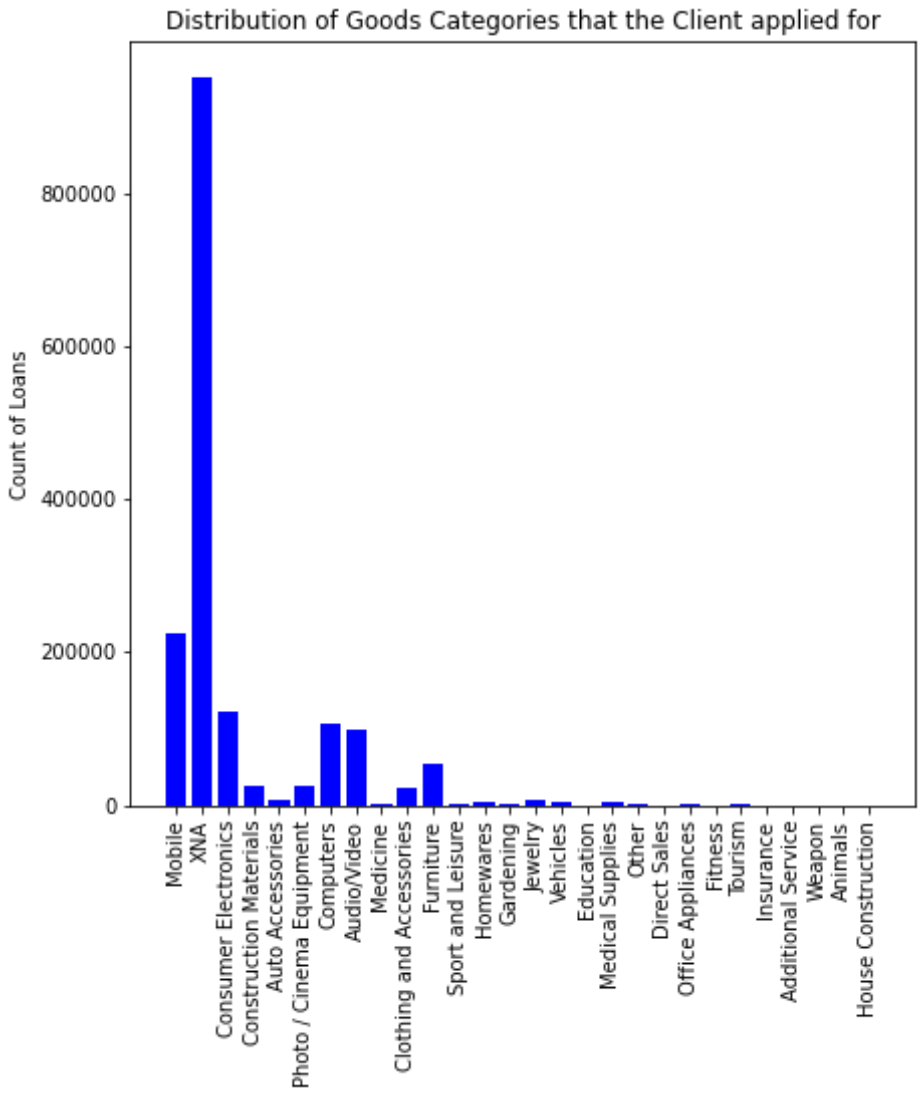


Observations :

- This particular column defines whether the client was old or new when he/she was applying for the previous application. We can see from here that most of the applicants for the previous application were repeaters and there were very few first time applicants.

6.2.6 Univariate Analysis: Name_Goods_Category

In [116]:
```
application_goods_category = Counter()
for category in previous_application['NAME_GOODS_CATEGORY'].values:
    application_goods_category.update(category.split(','))

# dict sort by value python: https://stackoverflow.com/a/613218/4084039
goods_category_dict = dict(application_goods_category)
sorted_goods_category = dict(sorted(goods_category_dict.items(), \
                                    key=lambda kv: kv[1], reverse=T
rue))

ind_9 = np.arange(len(goods_category_dict))
plt.figure(figsize=(7,7))
p1 = plt.bar(ind_9, list(goods_category_dict.values()),color='blue')

plt.ylabel('Count of Loans')
plt.title('Distribution of Goods Categories that the Client applied fo
r')
plt.xticks(ind_9, list(goods_category_dict.keys()),rotation=90)
plt.show()
```

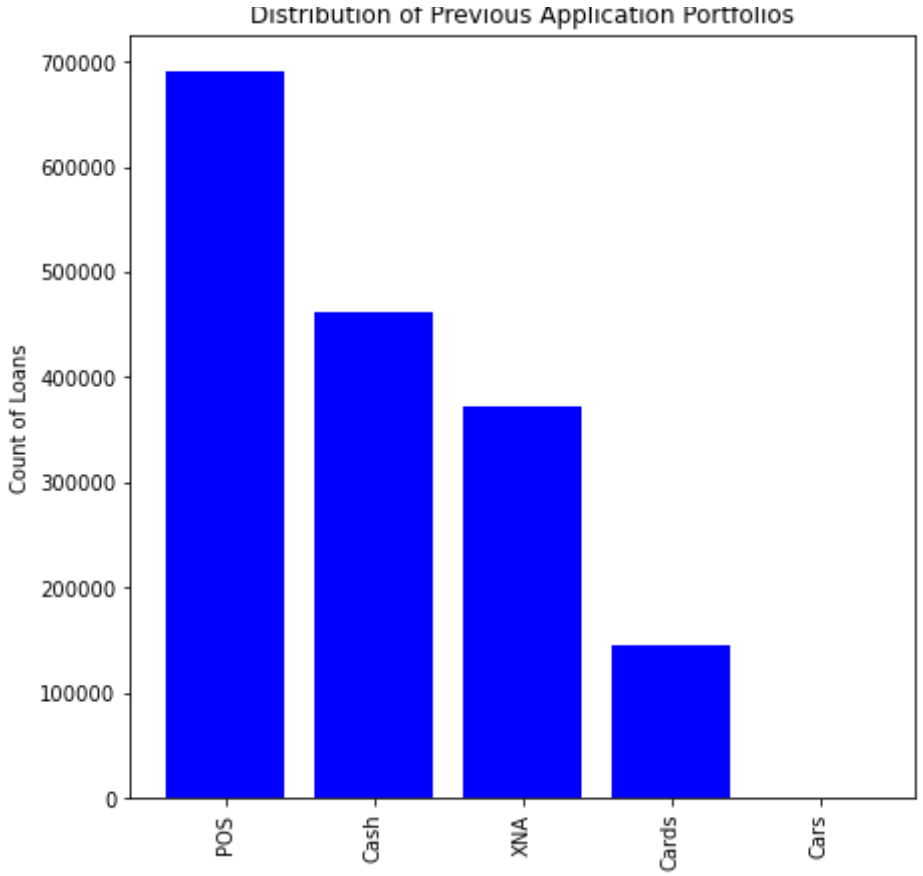


Observations :

- This defines the kind of goods that the client applied for in the previous application, and as can be seen, XNA is the most popular goods category followed by Mobiles.

6.2.7 Univariate Analysis: Name_Portfolio

In [117]:
```
application_name_portfolio = Counter()
for name in previous_application['NAME_PORTFOLIO'].values:
    application_name_portfolio.update(name.split(','))

# dict sort by value python: https://stackoverflow.com/a/613218/4084039
application_name_dict = dict(application_name_portfolio)
sorted_name_portfolio = dict(sorted(application_name_dict.items(), \
                                    key=lambda kv: kv[1], reverse=T
rue))

ind_10 = np.arange(len(application_name_dict))
plt.figure(figsize=(7,7))
p1 = plt.bar(ind_10, list(application_name_dict.values()),color='blue')

plt.ylabel('Count of Loans')
plt.title('Distribution of Previous Application Portfolios')
plt.xticks(ind_10, list(application_name_dict.keys()),rotation=90)
plt.show()
```



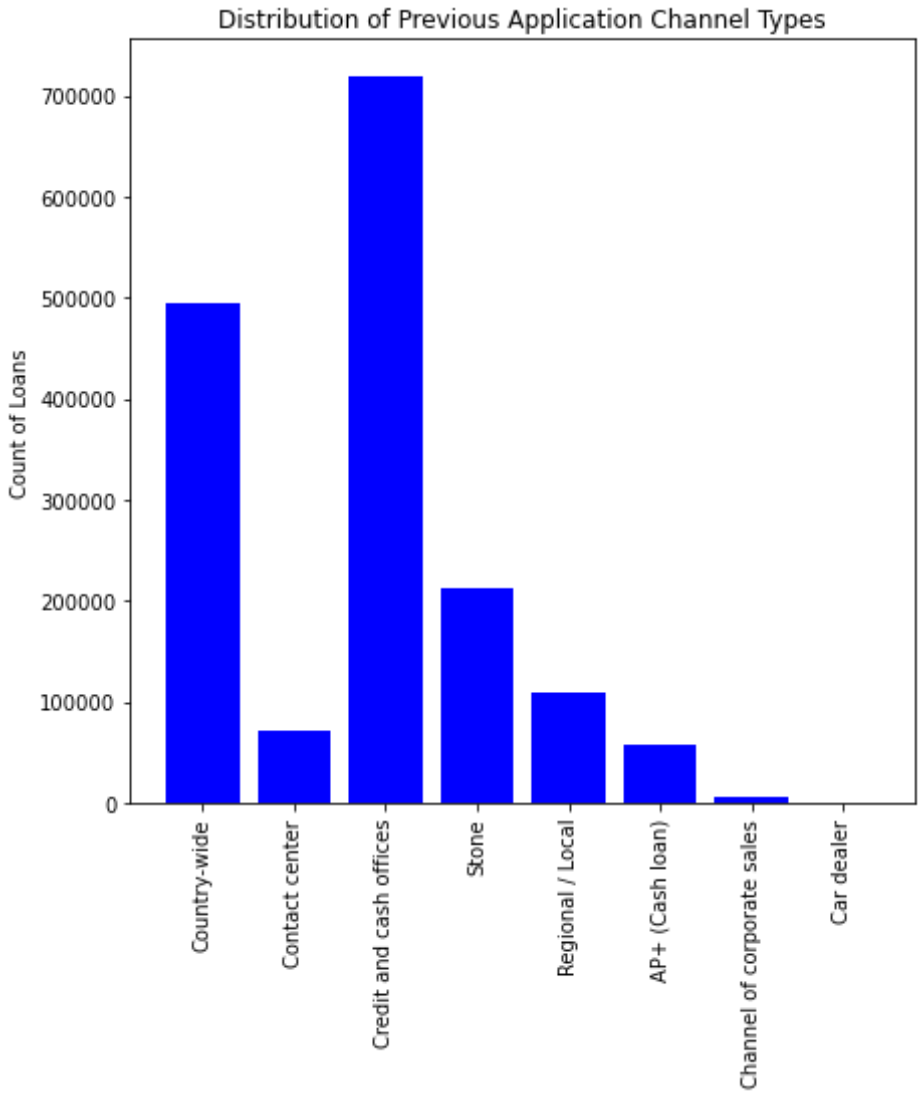
Observations :

- This shows that most of the previous applications were for POS, which is followed by Cash and XNA.

6.2.8 Univariate Analysis: Channel_Type

In [118]:
```
application_channel_type = Counter()
for type in previous_application['CHANNEL_TYPE'].values:
```

```
    application_channel_type.update(type.split(','))

# dict sort by value python: https://stackoverflow.com/a/613218/4084039
channel_type_dict = dict(application_channel_type)
sorted_channel_type = dict(sorted(application_channel_type.items(), \
                                          key=lambda kv: kv[1], reverse=T
rue))

ind_11 = np.arange(len(channel_type_dict))
plt.figure(figsize=(7,7))
p1 = plt.bar(ind_11, list(channel_type_dict.values()),color='blue')

plt.ylabel('Count of Loans')
plt.title('Distribution of Previous Application Channel Types')
plt.xticks(ind_11, list(channel_type_dict.keys()),rotation=90)
plt.show()
```

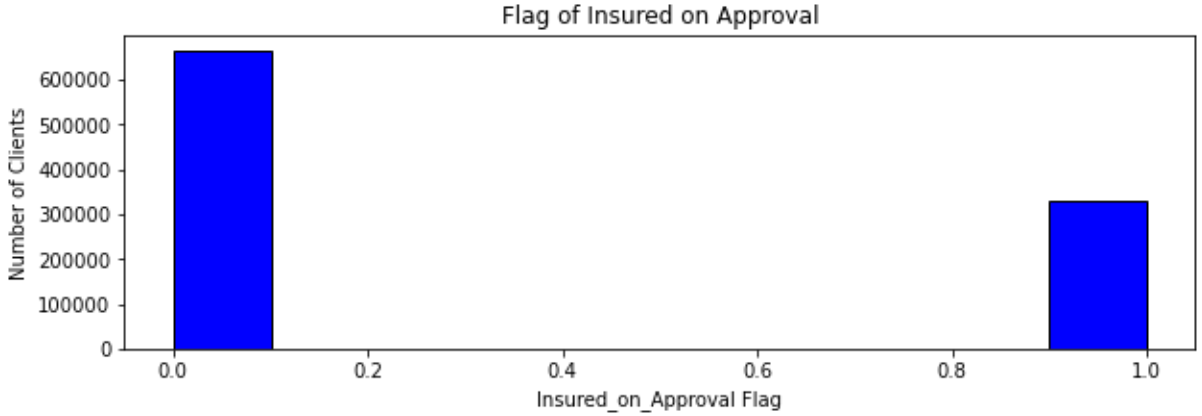


Observations :

- This shows that the banks obtained most of the clients in their previous application through Credit and Cash offices, which is followed by Country-wide.

6.2.9 Univariate Analysis: Nflag_Insured_on_Approval

In [119]:
```
plt.figure(figsize=(10,3))
plt.hist(previous_application['NFLAG_INSURED_ON_APPROVAL'].values, bins
=10, \
         edgecolor='black', color='blue')
plt.title('Flag of Insured on Approval')
plt.xlabel('Insured_on_Approval Flag')
plt.ylabel('Number of Clients')
plt.show()
```



Observations :

- There are much fewer clients who applied for Insurance in the previous application as compared to the number of clients who did not apply for insurance.

6.3 Dealing with Null Values and Outliers

We won't be able to find outliers for Amount and Cash related features except that these values cannot be negative. Except these, we will try and deal with the remaining features.

6.3.1 Days_Decision

This feature basically refers to the number of days relative to the current application that the decision made about the previous application.

In [120]:
```
previous_application['DAYS_DECISION'].describe()
```

Out[120]:
```
count    1.670214e+06
mean    -8.806797e+02
std      7.790997e+02
min     -2.922000e+03
25%     -1.300000e+03
50%     -5.810000e+02
75%     -2.800000e+02
max     -1.000000e+00
Name: DAYS_DECISION, dtype: float64
```

In [121]:
```
print("The maximum Days_Decision (in years) across all applications = "
,\
      - min(previous_application['DAYS_DECISION'].values)/365)

print("The minimum Days_Decision (in years) across all applications = "
,\
      - max(previous_application['DAYS_DECISION'].values)/365)
```

```
The maximum Days_Decision (in years) across all applications =  8.00547
9452054795
The minimum Days_Decision (in years) across all applications =  0.00273
97260273972603
```

Observations :

- This basically means that both the minimum as well as the maxium ages are admissible and there are no outliers present in the 'Days_Decision' column.

6.3.2 Days_First_Drawing

This feature means that relative to application date of the current application when was the first disbursement of the previous application carried out.

In [122]:
```
previous_application['DAYS_FIRST_DRAWING'].describe()
```

Out[122]:
```
count    997149.000000
mean     340114.343750
std       88611.609375
min       -2922.000000
25%      365243.000000
50%      365243.000000
75%      365243.000000
max      365243.000000
Name: DAYS_FIRST_DRAWING, dtype: float64
```

In [123]:
```
print("The maximum Days_First_Drawing (in years) across all application
s = ",\
      max(previous_application['DAYS_FIRST_DRAWING'].values)/365)

print("The minimum Days_First_Drawing (in years) across all application
s = ",\
      min(previous_application['DAYS_FIRST_DRAWING'].values)/365)
```

```
The maximum Days_First_Drawing (in years) across all applications =  10
00.6657534246575
The minimum Days_First_Drawing (in years) across all applications =  -
8.005479452054795
```

Note

This basically means that the maximum Days_First_Drawing in the table is 1000 years before the current application, which is obviously impossible. Also, since the 25th, 50th, and 75th Percentile values of this feature is the same as the maximum value across all features, we will first deal with removing this maximum value.

In [124]:
```
previous_application['DAYS_FIRST_DRAWING'].replace(max(previous_applica
tion['DAYS_FIRST_DRAWING'].values),\
                                                   np.nan, inplace=True)
```

In [125]:
```
previous_application['DAYS_FIRST_DRAWING'].describe()
```

Out[125]:
```
count    62705.000000
mean     -1035.251709
std        922.704407
min      -2922.000000
25%      -1721.000000
50%       -621.000000
75%       -303.000000
max         -2.000000
Name: DAYS_FIRST_DRAWING, dtype: float64
```

Observations :

- Now we can see that both the minimum as well as maximum values across Days_First_Drawing are admissible (both are negative with respect to the current application).

6.3.3 Days_First_Due

Days_First_Due means that relative to application date of current application when was the first due supposed to be of the previous application.

In [126]:
```
previous_application['DAYS_FIRST_DUE'].describe()
```

Out[126]:
```
count    997149.000000
mean      13838.132812
std       72421.296875
min       -2892.000000
25%       -1628.000000
50%        -831.000000
75%        -411.000000
max      365243.000000
Name: DAYS_FIRST_DUE, dtype: float64
```

In [127]:
```
previous_application['DAYS_FIRST_DUE'].replace(np.nan,0, inplace= True)

print("The minimum Days_First_Due (in years) across all applications =
 ",\
      min(previous_application['DAYS_FIRST_DUE'].values)/365)

print("The maximum Days_First_Due (in years) across all applications =
 ",\
      max(previous_application['DAYS_FIRST_DUE'].values)/365)

previous_application['DAYS_FIRST_DUE'].replace(0,np.nan, inplace= True)
```

```
The minimum Days_First_Due (in years) across all applications =  -7.923
287671232877
The maximum Days_First_Due (in years) across all applications =  1000.6
657534246575
```

Note

This basically means that the maximum Days_First_Due in the table is 1000 years before the current application, which is obviously impossible. So we will first deal with removing this maximum value in this feature.

In [128]:
```
previous_application['DAYS_FIRST_DUE'].replace(max(previous_application
['DAYS_FIRST_DUE'].values),\
                                               np.nan, inplace=True)
previous_application['DAYS_FIRST_DUE'].describe()
```

Out[128]:
```
count    956504.000000
mean      -1106.573364
std         790.587769
min       -2892.000000
25%       -1676.000000
50%        -874.000000
75%        -459.000000
max          -2.000000
Name: DAYS_FIRST_DUE, dtype: float64
```

Observations :

- Now we can see that both the minimum as well as maximum values across Days_First_Due are admissible (both are negative with respect to the current application).

6.3.4 Days_Last_Due_1st_Version

This feature means that relative to application date of the current application, when was the first due of the previous application.

In [129]:
```
previous_application['DAYS_LAST_DUE_1ST_VERSION'].describe()
```

Out[129]:
```
count    997149.000000
mean      33764.871094
std      106544.812500
min       -2801.000000
25%       -1242.000000
50%        -361.000000
75%         129.000000
max      365243.000000
Name: DAYS_LAST_DUE_1ST_VERSION, dtype: float64
```

In [130]:
```
previous_application['DAYS_LAST_DUE_1ST_VERSION'].replace(np.nan,0, inp
lace= True)
```

```
print("The minimum Days_Last_Due_1st_Version (in years) across all appl
ications = ",\
      np.round(min(previous_application['DAYS_LAST_DUE_1ST_VERSION'].v
alues)/365,3))

print("The maximum Days_Last_Due_1st_Version (in years) across all appl
ications = ",\
      np.round(max(previous_application['DAYS_LAST_DUE_1ST_VERSION'].v
alues)/365,3))

previous_application['DAYS_LAST_DUE_1ST_VERSION'].replace(0,np.nan, inp
lace= True)
```

```
The minimum Days_Last_Due_1st_Version (in years) across all application
s =  -7.674
The maximum Days_Last_Due_1st_Version (in years) across all application
s =  1000.666
```

Note

Again, this means that the maximum Days_Last_Due_1st_Version in the table is 1000 years before the current application, which is obviously impossible. So we will first deal with removing this maximum value in this feature.

In [131]:
```
previous_application['DAYS_LAST_DUE_1ST_VERSION'].replace(max(previous_
application['DAYS_LAST_DUE_1ST_VERSION'].values),\
                                                          np.nan, inplace=True)
previous_application['DAYS_LAST_DUE_1ST_VERSION'].describe()
```

Out[131]:
```
count    902580.000000
mean       -677.701843
std         923.615356
min       -2801.000000
25%       -1360.000000
50%        -481.000000
75%          -2.000000
```

```
max        2389.000000
Name: DAYS_LAST_DUE_1ST_VERSION, dtype: float64
```

Observations :

- Again, we can see that both the minimum as well as maximum values across Days_Last_Due_1st_Version are admissible (both are negative with respect to the current application).

6.3.5 Days_Last_Due

Days_Last_Due means that relative to the application date of the current application, when was the last due date of the previous application.

In [132]:
```
previous_application['DAYS_LAST_DUE'].describe()
```

Out[132]:
```
count    997149.000000
mean      76829.148438
std      150155.109375
min       -2889.000000
25%       -1314.000000
50%        -537.000000
75%         -74.000000
max      365243.000000
Name: DAYS_LAST_DUE, dtype: float64
```

In [133]:
```
previous_application['DAYS_LAST_DUE'].replace(np.nan,0, inplace= True)

print("The minimum Days_Last_Due (in years) across all applications = "
,\
      np.round(min(previous_application['DAYS_LAST_DUE'].values)/365,3
))

print("The maximum Days_Last_Due (in years) across all applications = "
```

```
,\
       np.round(max(previous_application['DAYS_LAST_DUE'].values)/365,3
))

previous_application['DAYS_LAST_DUE'].replace(0,np.nan, inplace= True)
```

```
The minimum Days_Last_Due (in years) across all applications =  -7.915
The maximum Days_Last_Due (in years) across all applications =  1000.66
6
```

Note

This means that the maximum Days_Last_Due in the table is 1000 years before the current application, which is impossible. So we will first deal with removing this maximum value in this feature.

In [134]:
```
previous_application['DAYS_LAST_DUE'].replace(max(previous_application[
'DAYS_LAST_DUE'].values),\
                                              np.nan, inplace=True)
previous_application['DAYS_LAST_DUE'].describe()
```

Out[134]:
```
count    785928.000000
mean       -996.137085
std         752.599609
min       -2889.000000
25%       -1566.000000
50%        -801.000000
75%        -353.000000
max          -2.000000
Name: DAYS_LAST_DUE, dtype: float64
```

Observations :

- We can see here that both the minimum as well as maximum values across Days_Last_Due are admissible (both are negative with respect to the current application).

6.3.6 Days_Termination

Days_Termination means that relative to the application date of the current application, when was the expected termination of the previous application.

In [135]:
```
previous_application['DAYS_TERMINATION'].describe()
```

Out[135]:
```
count    997149.00000
mean      82314.84375
std      152926.93750
min       -2874.00000
25%       -1270.00000
50%        -499.00000
75%         -44.00000
max      365243.00000
Name: DAYS_TERMINATION, dtype: float64
```

In [136]:
```
previous_application['DAYS_TERMINATION'].replace(np.nan,0, inplace= True)

print("The minimum Days_Termination (in years) across all applications = ",\
      np.round(min(previous_application['DAYS_TERMINATION'].values)/365,3))

print("The maximum Days_Termination (in years) across all applications = ",\
      np.round(max(previous_application['DAYS_TERMINATION'].values)/365,3))

previous_application['DAYS_TERMINATION'].replace(0,np.nan, inplace= True)
```

```
The minimum Days_Termination (in years) across all applications =  -7.874
The maximum Days_Termination (in years) across all applications =  1000.666
```

Note

This means that the maximum Days_Termination in the table is 1000 years before the current application, which is impossible. So we will first deal with removing this maximum value in this feature.

In [137]:
```
previous_application['DAYS_TERMINATION'].replace(max(previous_applicati
on['DAYS_TERMINATION'].values),\
                                                 np.nan, inplace=True)
previous_application['DAYS_TERMINATION'].describe()
```

Out[137]:
```
count    771236.000000
mean       -978.317993
std         749.048645
min       -2874.000000
25%       -1539.000000
50%        -780.000000
75%        -337.000000
max          -2.000000
Name: DAYS_TERMINATION, dtype: float64
```

Observations :

- Both the minimum as well as maximum values across Days_Termination are admissible (both are negative with respect to the current application).

6.4 Feature Engineering on Previous Application

Function to carry out Feature Engineering for Multiple features

In [138]:
```
def FE_previous_application(previous_application):

    prev_app, previous_application_columns = one_hot_encode(previous_ap
plication)
```

```
    prev_app['APPLICATION_CREDIT_DIFF'] = prev_app['AMT_APPLICATION'] -
prev_app['AMT_CREDIT']
    prev_app['APPLICATION_CREDIT_RATIO'] = prev_app['AMT_APPLICATION']
/ prev_app['AMT_CREDIT']
    prev_app['CREDIT_TO_ANNUITY_RATIO'] = prev_app['AMT_CREDIT']/prev_a
pp['AMT_ANNUITY']
    prev_app['DOWN_PAYMENT_TO_CREDIT'] = prev_app['AMT_DOWN_PAYMENT'] /
prev_app['AMT_CREDIT']

    total_payment = prev_app['AMT_ANNUITY'] * prev_app['CNT_PAYMENT']
    prev_app['SIMPLE_INTERESTS'] = (total_payment/prev_app['AMT_CREDIT'
] - 1)/prev_app['CNT_PAYMENT']

    prev_app['DAYS_LAST_DUE_DIFF'] = prev_app['DAYS_LAST_DUE_1ST_VERSIO
N'] - prev_app['DAYS_LAST_DUE']

    numerical_agg_prev = {'AMT_ANNUITY': ['max', 'mean'], 'AMT_APPLICAT
ION': ['max','mean'],\
                          'AMT_CREDIT':['max','mean'], 'AMT_DOWN_PAYMENT': [
'max','mean'],\
                          'AMT_GOODS_PRICE':['mean','sum'], 'HOUR_APPR_PROC
ESS_START' :\
                          ['max','mean'], 'RATE_DOWN_PAYMENT':['max','mean'
], 'RATE_INTEREST_PRIMARY':\
                          ['max','mean'],'RATE_INTEREST_PRIVILEGED':['max',
'mean'], \
                          'DAYS_DECISION': ['max','mean'], 'CNT_PAYMENT' :[
'mean','sum'], \
                          'DAYS_FIRST_DRAWING': ['max','mean'], 'DAYS_TERMI
NATION' : ['max','mean'],\
                          'APPLICATION_CREDIT_RATIO': ['max','mean'], 'DOWN
_PAYMENT_TO_CREDIT' : \
                          ['max','mean'], 'DAYS_LAST_DUE_DIFF': ['max','mea
n']}

    categorical_agg_prev = {}

    for column in previous_application_columns:
        categorical_agg_prev[column] = ['mean']
```

```
    prev_app_agg1 = prev_app.groupby('SK_ID_CURR').agg({**numerical_agg
_prev, \
                                                        **categorical_agg_p
rev})
    col_list_5 =[]

    for col in prev_app_agg1.columns.tolist():
        col_list_5.append('PREV_'+col[0]+'_'+col[1].upper())

    prev_app_agg1.columns = pd.Index(col_list_5)

    prev_app_cs_approved = prev_app[prev_app['NAME_CONTRACT_STATUS_Appr
oved']==1]
    prev_app_agg2 = prev_app_cs_approved.groupby('SK_ID_CURR').agg(nume
rical_agg_prev)

    col_list_6 = []

    for col in prev_app_agg2.columns.tolist():
        col_list_6.append('CS_APP_' + col[0] + '_' + col[1].upper())

    prev_app_agg2.columns = pd.Index(col_list_6)

    prev_app_agg1_join = prev_app_agg1.join(prev_app_agg2, how='left',
on='SK_ID_CURR')

    prev_app_cs_refused = prev_app[prev_app['NAME_CONTRACT_STATUS_Refus
ed']==1]
    prev_app_agg3 = prev_app_cs_refused.groupby('SK_ID_CURR').agg(numer
ical_agg_prev)

    col_list_7 =[]

    for col in prev_app_agg3.columns.tolist():
        col_list_7.append('CS_REF_' + col[0] + '_' + col[1].upper())

    prev_app_agg3.columns = pd.Index(col_list_7)
    prev_app_agg_final = prev_app_agg1_join.join(prev_app_agg3,how='lef
t', on='SK_ID_CURR')
```

```
    del prev_app_agg1_join, prev_app_agg3, prev_app_cs_refused, prev_ap
p_agg1, prev_app_agg2,\
        prev_app_cs_approved
    gc.collect()
    return prev_app_agg_final
```

Function to carry out Feature Engineering using 'Days_Decision'

In [139]:
```
def FE_previous_application_days_decision(data,data_temp,previous_appli
cation):

    temp_1 = FE_previous_application(reduce_memory_usage(previous_appli
cation))
    data = data_temp.merge(temp_1, how='left', on='SK_ID_CURR')
    del temp_1
    gc.collect()

    temp_2 = reduce_memory_usage(previous_application[previous_applicat
ion['DAYS_DECISION']>=-365].reset_index())
    temp_2.drop(['index'], axis=1, inplace=True)
    temp_2 = FE_previous_application(temp_2)
    data = data.join(temp_2, how='left', on='SK_ID_CURR',rsuffix='_yea
r')
    del temp_2
    gc.collect()

    temp_3 = reduce_memory_usage(previous_application[previous_applicat
ion['DAYS_DECISION']>=-182].reset_index())
    temp_3.drop(['index'], axis=1, inplace=True)
    temp_3 = FE_previous_application(temp_3)
    data = data.join(temp_3, how='left', on='SK_ID_CURR', rsuffix='_hal
f_year')
    del temp_3
    gc.collect()

    temp_4 = reduce_memory_usage(previous_application[previous_applicat
ion['DAYS_DECISION']>=-90].reset_index())
```

```
    temp_4.drop(['index'], axis=1, inplace=True)
    temp_4 = FE_previous_application(temp_4)
    data = data.join(temp_4, how='left', on='SK_ID_CURR', rsuffix='_qua
rter')
    del temp_4
    gc.collect()

    temp_5 = reduce_memory_usage(previous_application[previous_applicat
ion['DAYS_DECISION']>=-30].reset_index())
    temp_5.drop(['index'], axis=1, inplace=True)
    temp_5 = FE_previous_application(temp_5)
    data = data.join(temp_5, how='left', on='SK_ID_CURR', rsuffix='_mon
th')
    del temp_5
    gc.collect()

    temp_6 = reduce_memory_usage(previous_application[previous_applicat
ion['DAYS_DECISION']>=-14].reset_index())
    temp_6.drop(['index'], axis=1, inplace=True)
    temp_6 = FE_previous_application(temp_6)
    data = data.join(temp_6, how='left', on='SK_ID_CURR', rsuffix='_for
tnight')
    del temp_6
    gc.collect()

    temp_7 = reduce_memory_usage(previous_application[previous_applicat
ion['DAYS_DECISION']>=-7].reset_index())
    temp_7.drop(['index'], axis=1, inplace=True)
    temp_7 = FE_previous_application(temp_7)
    data = data.join(temp_7, how='left', on='SK_ID_CURR', rsuffix='_wee
k')
    del temp_7
    gc.collect()

    return data
```

Carrying out Feature Engineering using the Functions Defined

In [140]:
```
train_data_temp_2 = FE_previous_application_days_decision(train_data, t
rain_data_temp_2,previous_application)
train_data_temp_2.shape
```

```
Memory usage of dataframe is 309.01 MB
Memory usage after optimization is: 293.08 MB
Decreased by 5.2%
Memory usage of dataframe is 105.06 MB
Memory usage after optimization is: 102.87 MB
Decreased by 2.1%
Memory usage of dataframe is 40.52 MB
Memory usage after optimization is: 39.67 MB
Decreased by 2.1%
Memory usage of dataframe is 16.19 MB
Memory usage after optimization is: 15.77 MB
Decreased by 2.6%
Memory usage of dataframe is 6.82 MB
Memory usage after optimization is: 6.65 MB
Decreased by 2.6%
Memory usage of dataframe is 3.22 MB
Memory usage after optimization is: 3.13 MB
Decreased by 2.6%
Memory usage of dataframe is 1.57 MB
Memory usage after optimization is: 1.53 MB
Decreased by 2.6%
```

Out[140]:
```
(307511, 2084)
```

7. POS Cash Balance Dataset

7.1. Basic Overview of the POS Cash Balance Data

In [141]:
```
pos_cash_balance = reduce_memory_usage(pd.read_csv('home-credit-default
-risk/POS_CASH_balance.csv'))
print('Number of data points : ', pos_cash_balance.shape[0])
```

```
print('Number of features : ', pos_cash_balance.shape[1])
pos_cash_balance.head()
```

```
Memory usage of dataframe is 610.43 MB
Memory usage after optimization is: 238.45 MB
Decreased by 60.9%
Number of data points :  10001358
Number of features :  8
```

Out[141]:

	SK_ID_PREV	SK_ID_CURR	MONTHS_BALANCE	CNT_INSTALMENT	CNT_INSTALMENT_FUTURI
0	1803195	182943	-31	48.0	45.
1	1715348	367990	-33	36.0	35.
2	1784872	397406	-32	12.0	9.
3	1903291	269225	-35	48.0	42.
4	2341044	334279	-35	36.0	35.

In [142]:
```
pos_cash_balance.columns
```

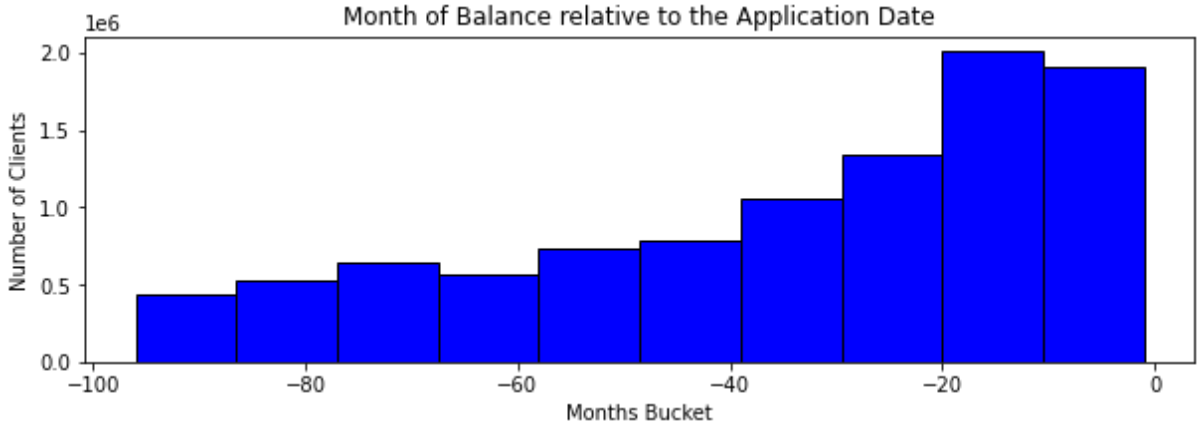
Out[142]:
```
Index(['SK_ID_PREV', 'SK_ID_CURR', 'MONTHS_BALANCE', 'CNT_INSTALMENT',
       'CNT_INSTALMENT_FUTURE', 'NAME_CONTRACT_STATUS', 'SK_DPD',
       'SK_DPD_DEF'],
      dtype='object')
```

7.2 POS Cash Balance Data Analysis

7.2.1 Univariate Analysis : Months_Balance

Months_Balance refers to the Month of balance relative to the application date (0 could mean the same as -1 because as many banks are not updating the information to the Credit Bureau with regularity).

In [143]:
```
plt.figure(figsize=(10,3))
plt.hist(pos_cash_balance['MONTHS_BALANCE'].values, bins=10, edgecolor=
'black', color='blue')
plt.title('Month of Balance relative to the Application Date')
plt.xlabel('Months Bucket')
plt.ylabel('Number of Clients')
plt.show()
```



Observations :

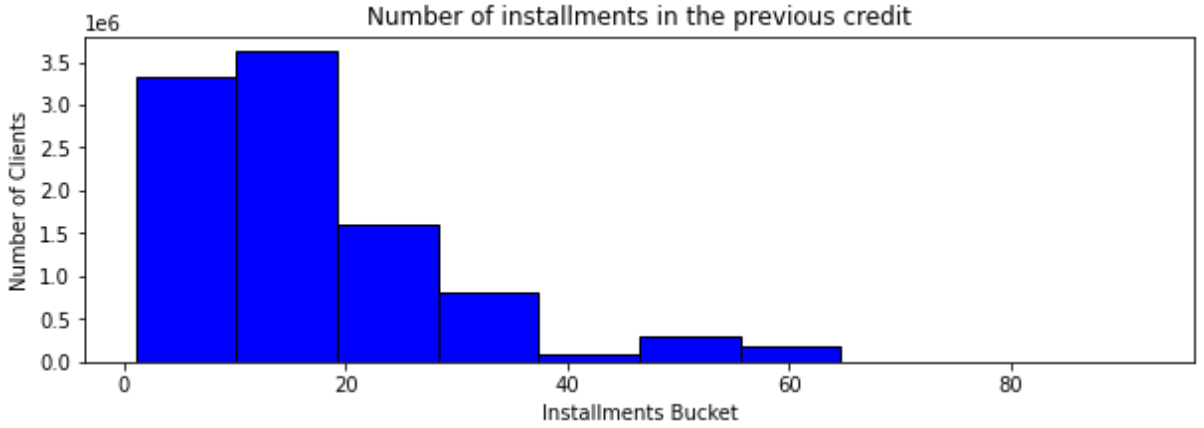
- The Months_Balance for a large number of the clients is between 10 and 20 months before the date of application.
- This is followed by clients with Months_Balance less than 10 months.

7.2.2 Univariate Analysis : Cnt_Instalment

Cnt_Installment refers to the term of the previous credit (which can change over the period of time).

In [144]:
```
plt.figure(figsize=(10,3))
plt.hist(pos_cash_balance['CNT_INSTALMENT'].values, bins=10, edgecolor=
```

```
'black', color='blue')
plt.title('Number of installments in the previous credit')
plt.xlabel('Installments Bucket')
plt.ylabel('Number of Clients')
plt.show()
```



Observations :

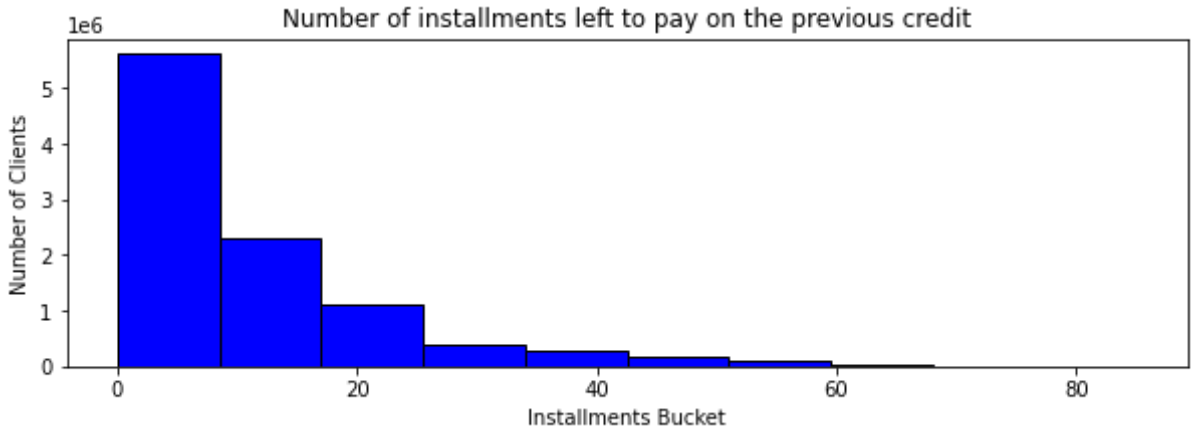
- The Number of installments in the previous credit for most clients lies between 10 and 20.
- This is followed by clients whose installment count is less than 10 months.

7.2.3 Univariate Analysis : Cnt_Instalment_Future

This feature refers to the number of installments left to pay on the previous credit.

In [145]:
```
plt.figure(figsize=(10,3))
plt.hist(pos_cash_balance['CNT_INSTALMENT_FUTURE'].values, bins=10, edg
ecolor='black', color='blue')
plt.title('Number of installments left to pay on the previous credit')
plt.xlabel('Installments Bucket')
```

```
plt.ylabel('Number of Clients')
plt.show()
```



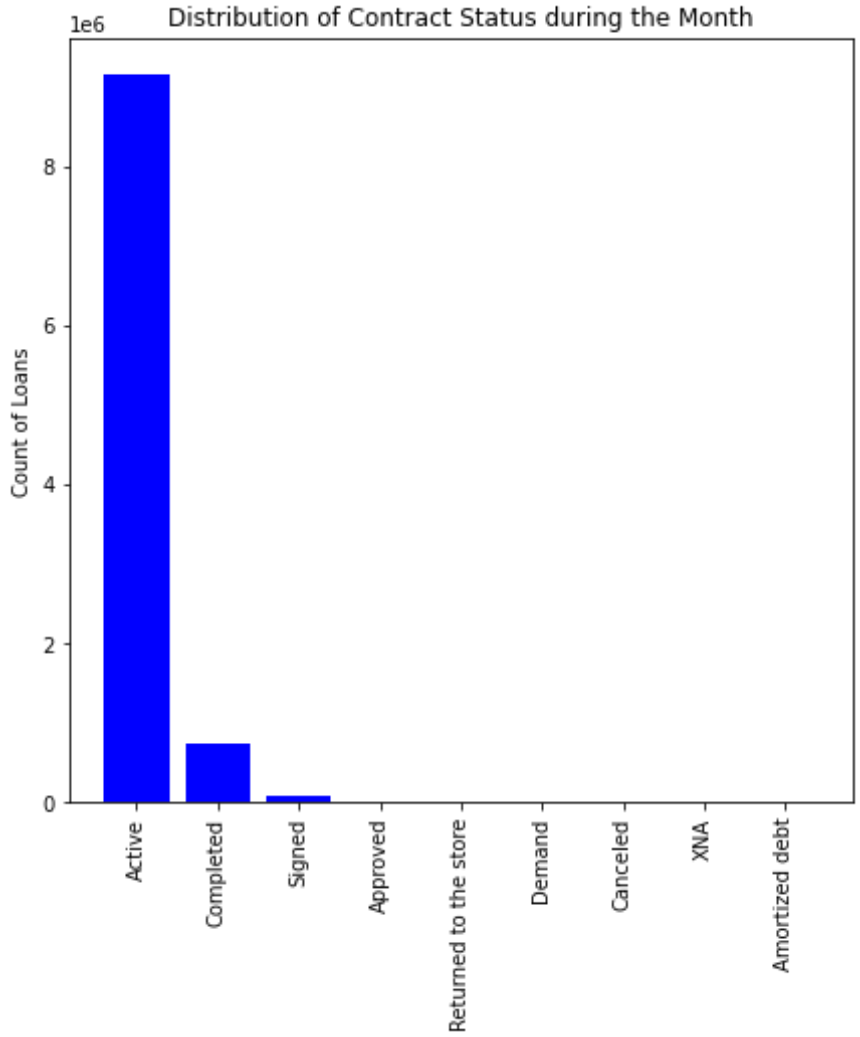
Observations :

- Most of the clients have less than 10 installments left to pay on the previous credit.
- This is followed by clients whose installment count is between 10 and 20.

7.2.4 Univariate Analysis : Name_Contract_Status

In [146]:
```
balance_contract_status = Counter()
for status in pos_cash_balance['NAME_CONTRACT_STATUS'].values:
    balance_contract_status.update(status.split(','))

# dict sort by value python: https://stackoverflow.com/a/613218/4084039
contract_status_dict = dict(balance_contract_status)
sorted_contract_status = dict(sorted(balance_contract_status.items(), \
                                     key=lambda kv: kv[1], reverse=T
rue))

ind_12 = np.arange(len(contract_status_dict))
plt.figure(figsize=(7,7))
```

```
p1 = plt.bar(ind_12, list(contract_status_dict.values()),color='blue')

plt.ylabel('Count of Loans')
plt.title('Distribution of Contract Status during the Month')
plt.xticks(ind_12, list(contract_status_dict.keys()),rotation=90)
plt.show()
```



Observations :

- The Name_Contract_Status is Active in the majority of the cases.

7.3 Feature Engineering on POS Cash Balance Dataset

Function to carry out Feature Engineering for Multiple features

In [147]:
```
def FE_pos_cash_balance(pos_cash_balance):

    pos_balance_data, pos_balance_columns = one_hot_encode(pos_cash_bal
ance)

    pos_balance_data['LATE_PAYMENT'] = pos_balance_data['SK_DPD'].apply
(lambda x:1 \
                                                 if x>0 else 0)
    numerical_agg_pos_balance = {'SK_DPD_DEF': ['max', 'mean','min'],'S
K_DPD': ['max', 'mean','min'],
        'MONTHS_BALANCE': ['max', 'mean', 'size'], 'CNT_INSTALMENT': [
'max','size'],
        'CNT_INSTALMENT_FUTURE': ['max','size','sum']}

    categorical_agg_pos_balance = {}

    for col in pos_balance_columns:
        categorical_agg_pos_balance[col] = ['mean']

    pos_balance_agg = pos_balance_data.groupby('SK_ID_CURR').agg({**num
erical_agg_pos_balance, \
                                                **categorical_agg_p
os_balance})
    col_list_8=[]
    for col in pos_balance_agg.columns.tolist():
        col_list_8.append('POS_'+col[0] + '_' + col[1].upper())

    pos_balance_agg.columns = pd.Index(col_list_8)
```

```
    sort_pos_balance = pos_balance_data.sort_values(by=['SK_ID_PREV',
'MONTHS_BALANCE'])
    pos_group = sort_pos_balance.groupby('SK_ID_PREV')

    pos_final_df = pd.DataFrame()
    pos_final_df['SK_ID_CURR'] = pos_group['SK_ID_CURR'].first()
    pos_final_df['MONTHS_BALANCE_MAX'] = pos_group['MONTHS_BALANCE'].ma
x()

    pos_final_df['POS_LOAN_COMPLETED_MEAN'] = pos_group['NAME_CONTRACT_
STATUS_Completed'].mean()
    pos_final_df['POS_COMPLETED_BEFORE_MEAN'] = pos_group['CNT_INSTALME
NT'].first() - \
                                                pos_group['CNT_INSTALMENT']
.last()
    pos_final_df['POS_COMPLETED_BEFORE_MEAN'] = pos_final_df.apply(lamb
da x: 1 if x['POS_COMPLETED_BEFORE_MEAN'] > 0
                                                        and x['POS_LOAN_COMPLET
ED_MEAN'] > 0 else 0, axis=1)

    pos_final_df['POS_REMAINING_INSTALMENTS'] = pos_group['CNT_INSTALME
NT_FUTURE'].last()
    pos_final_df['POS_REMAINING_INSTALMENTS_RATIO'] = pos_group['CNT_IN
STALMENT_FUTURE'].last()/pos_group['CNT_INSTALMENT'].last()

    pos_final_df_groupby = pos_final_df.groupby('SK_ID_CURR').sum().res
et_index()
    pos_final_df_groupby.drop(['MONTHS_BALANCE_MAX'], axis=1, inplace=
True)
    pos_final_agg = pd.merge(pos_balance_agg, pos_final_df_groupby, on=
'SK_ID_CURR',\
                             how= 'left')

    del pos_balance_agg, pos_final_df_groupby, pos_group, sort_pos_bala
nce
    gc.collect()
    return pos_final_agg
```

Function to carry out Feature Engineering using 'Months_Balance'

In [148]:
```
def FE_pos_cash_balance_months_balance(data, data_temp, pos_cash_balanc
e):

    temp_8 = FE_pos_cash_balance(reduce_memory_usage(pos_cash_balance))
    data = data_temp.merge(temp_8, how='left', on='SK_ID_CURR')
    del temp_8
    gc.collect()

    temp_9 = reduce_memory_usage(pos_cash_balance[pos_cash_balance['MON
THS_BALANCE']>=-12].reset_index())
    temp_9.drop(['index'], axis=1, inplace=True)
    temp_9 = FE_pos_cash_balance(temp_9)
    data = data.join(temp_9, how='left', on='SK_ID_CURR',rsuffix='_yea
r')
    del temp_9
    gc.collect()

    temp_10 = reduce_memory_usage(pos_cash_balance[pos_cash_balance['MO
NTHS_BALANCE']>=-6].reset_index())
    temp_10.drop(['index'], axis=1, inplace=True)
    temp_10 = FE_pos_cash_balance(temp_10)
    data = data.join(temp_10, how='left', on='SK_ID_CURR', rsuffix='_ha
lf_year')
    del temp_10
    gc.collect()

    temp_11 = reduce_memory_usage(pos_cash_balance[pos_cash_balance['MO
NTHS_BALANCE']>=-3].reset_index())
    temp_11.drop(['index'], axis=1, inplace=True)
    temp_11 = FE_pos_cash_balance(temp_11)
    data = data.join(temp_11, how='left', on='SK_ID_CURR', rsuffix='_qu
arter')
    del temp_11
    gc.collect()

    temp_12 = reduce_memory_usage(pos_cash_balance[pos_cash_balance['MO
NTHS_BALANCE']>=-1].reset_index())
```

```
    temp_12.drop(['index'], axis=1, inplace=True)
    temp_12 = FE_pos_cash_balance(temp_12)
    data = data.join(temp_12, how='left', on='SK_ID_CURR', rsuffix='_mo
nth')
    del temp_12
    gc.collect()

    return data
```

Carrying out Feature Engineering using the Functions Defined

In [149]:
```
train_data_temp_2 = FE_pos_cash_balance_months_balance(train_data,train
_data_temp_2, pos_cash_balance)
train_data_temp_2.shape
```

```
Memory usage of dataframe is 238.45 MB
Memory usage after optimization is: 238.45 MB
Decreased by 0.0%
Memory usage of dataframe is 73.51 MB
Memory usage after optimization is: 64.60 MB
Decreased by 12.1%
Memory usage of dataframe is 33.01 MB
Memory usage after optimization is: 29.00 MB
Decreased by 12.1%
Memory usage of dataframe is 14.10 MB
Memory usage after optimization is: 12.39 MB
Decreased by 12.1%
Memory usage of dataframe is 2.99 MB
Memory usage after optimization is: 2.62 MB
Decreased by 12.1%
```

Out[149]:
```
(307511, 2223)
```

8. Installments Payments Dataset

8.1. Basic Overview of the Installments Payments Data

In [150]:
```
installments_payments = reduce_memory_usage(pd.read_csv('home-credit-default-risk/installments_payments.csv'))
print('Number of data points : ', installments_payments.shape[0])
print('Number of features : ', installments_payments.shape[1])
installments_payments.head()
```

```
Memory usage of dataframe is 830.41 MB
Memory usage after optimization is: 311.40 MB
Decreased by 62.5%
Number of data points :  13605401
Number of features :  8
```

Out[150]:

	SK_ID_PREV	SK_ID_CURR	NUM_INSTALMENT_VERSION	NUM_INSTALMENT_NUMBER	DAYS_I
0	1054186	161674	1.0	6	
1	1330831	151639	0.0	34	
2	2085231	193053	2.0	1	
3	2452527	199697	1.0	3	
4	2714724	167756	1.0	2	

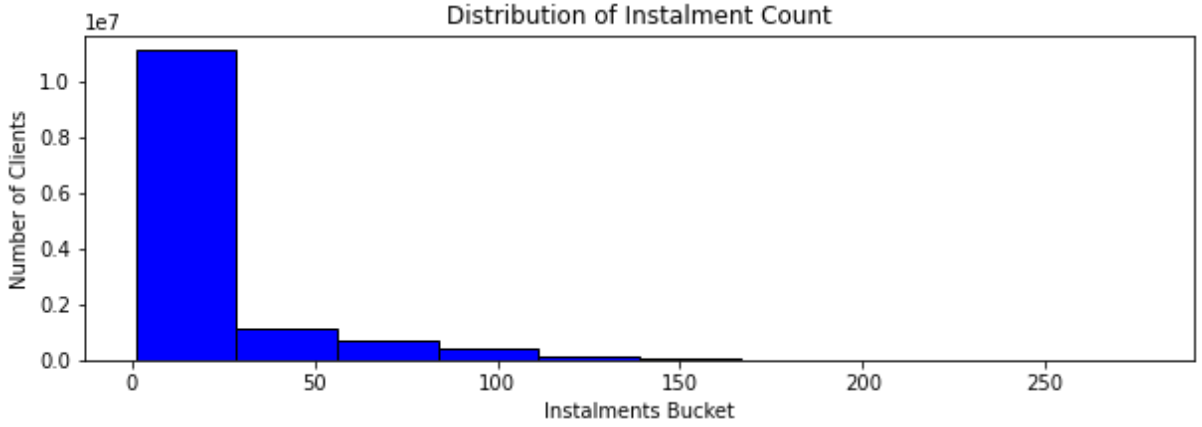
In [151]:
```
installments_payments.columns
```

Out[151]:
```
Index(['SK_ID_PREV', 'SK_ID_CURR', 'NUM_INSTALMENT_VERSION',
       'NUM_INSTALMENT_NUMBER', 'DAYS_INSTALMENT', 'DAYS_ENTRY_PAYMENT',
       'AMT_INSTALMENT', 'AMT_PAYMENT'],
      dtype='object')
```

8.2 Installments Payments Data Analysis

8.2.1 Univariate Analysis : Num_Instalment_Number

In [152]:
```
plt.figure(figsize=(10,3))
plt.hist(installments_payments['NUM_INSTALMENT_NUMBER'].values, bins=10
, edgecolor='black',\
         color='blue')
plt.title('Distribution of Instalment Count')
plt.xlabel('Instalments Bucket')
plt.ylabel('Number of Clients')
plt.show()
```



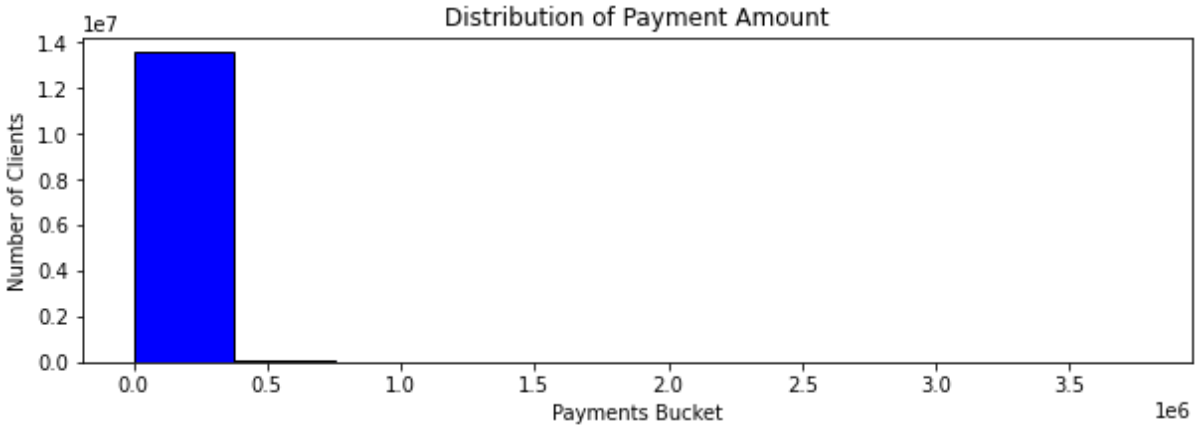
Observations :

- Most of the clients complete their instalment payment before 25 months.

8.2.2 Univariate Analysis : Amt_Payment

In [153]:
```
plt.figure(figsize=(10,3))
plt.hist(installments_payments['AMT_PAYMENT'].values, bins=10, edgecolo
r='black',\
         color='blue')
plt.title('Distribution of Payment Amount')
```

```
plt.xlabel('Payments Bucket')
plt.ylabel('Number of Clients')
plt.show()
```



Observations :

- Most of the clients paid less than 5 lakh on previous credit on the same installment.

8.3 Feature Engineering on Installments Payments Data

Function to carry out Feature Engineering for Multiple features

In [154]:
```
def FE_installments_payments(installments_payments):

    pay1 = installments_payments[['SK_ID_PREV', 'NUM_INSTALMENT_NUMBER'
]+ ['AMT_PAYMENT']]
    pay2 = pay1.groupby(['SK_ID_PREV', 'NUM_INSTALMENT_NUMBER'])['AMT_P
AYMENT'].sum().reset_index()
    pay_final = pay2.rename(columns={'AMT_PAYMENT': 'AMT_PAYMENT_GROUPE
D'})
    payments_final = installments_payments.merge(pay_final,\
```

```
                                on=['SK_ID_PREV','NUM_INSTALMENT_NUMBER'],
how='left')

    payments_final['PAYMENT_DIFFERENCE'] = payments_final['AMT_INSTALME
NT'] - \
                                           payments_final['AMT_PAYMENT_GROU
PED']
    payments_final['PAYMENT_RATIO'] = payments_final['AMT_INSTALMENT']
/ payments_final['AMT_PAYMENT_GROUPED']

    payments_final['PAID_OVER_AMOUNT'] = payments_final['AMT_PAYMENT']
- \
                                         payments_final['AMT_INSTALMENT']
    payments_final['PAID_OVER'] = (payments_final['PAID_OVER_AMOUNT'] >
0).astype(int)

    payments_final['DPD'] = payments_final['DAYS_ENTRY_PAYMENT'] - \
                            payments_final['DAYS_INSTALMENT']
    payments_final['DPD'] = payments_final['DPD'].apply(lambda x: 0 if
x <= 0 else x)

    payments_final['DBD'] = payments_final['DAYS_INSTALMENT'] - \
                            payments_final['DAYS_ENTRY_PAYMENT']
    payments_final['DBD'] = payments_final['DBD'].apply(lambda x: 0 if
x <= 0 else x)
    payments_final['LATE_PAYMENT'] = payments_final['DBD'].apply(lambda
x: 1 if x > 0 else 0)

    payments_final['INSTALMENT_PAYMENT_RATIO'] = payments_final['AMT_PA
YMENT'] / payments_final['AMT_INSTALMENT']
    payments_final['LATE_PAYMENT_RATIO'] = payments_final.apply(lambda
x: x['INSTALMENT_PAYMENT_RATIO'] if x['LATE_PAYMENT'] == 1 else 0, axis
=1)

    payments_final['SIGNIFICANT_LATE_PAYMENT'] = payments_final['LATE_P
AYMENT_RATIO'].apply(lambda x: 1 if x > 0.05 else 0)

    payments_final['DPD_7'] = payments_final['DPD'].apply(lambda x: 1 i
f x >= 7 else 0)
    payments_final['DPD_15'] = payments_final['DPD'].apply(lambda x: 1
```

```
if x >= 15 else 0)
    payments_final['DPD_30'] = payments_final['DPD'].apply(lambda x: 1
if x >= 30 else 0)
    payments_final['DPD_60'] = payments_final['DPD'].apply(lambda x: 1
if x >= 60 else 0)
    payments_final['DPD_90'] = payments_final['DPD'].apply(lambda x: 1
if x >= 90 else 0)
    payments_final['DPD_180'] = payments_final['DPD'].apply(lambda x: 1
if x >= 180 else 0)
    payments_final['DPD_WOF'] = payments_final['DPD'].apply(lambda x: 1
if x >= 720 else 0)

    payments_final, pay_final_columns = one_hot_encode(payments_final)

    numeric_agg_payments = {'LATE_PAYMENT': ['max','mean','min'],'AMT_P
AYMENT': ['min', 'max',\
                        'mean', 'sum'], 'NUM_INSTALMENT_VERSION': ['nuniq
ue'], \
                        'NUM_INSTALMENT_NUMBER':['max'], 'AMT_INSTALMENT'
: ['max', 'mean', 'sum'],
        'PAYMENT_DIFFERENCE': ['max','mean','min','sum'],'DAYS_ENTRY_PA
YMENT': ['max', \
        'mean', 'sum'],  'PAID_OVER_AMOUNT': ['max','mean','min']
               }

    for col in pay_final_columns:
        numeric_agg_payments[col] = ['mean']

    payments_final_agg = payments_final.groupby('SK_ID_CURR').agg(numer
ic_agg_payments)
    col_list_9=[]

    for col in payments_final_agg.columns.tolist():
        col_list_9.append('INS_'+col[0]+'_'+col[1].upper())

    payments_final_agg.columns = pd.Index(col_list_9)
    payments_final_agg['INSTALLATION_COUNT'] = payments_final.groupby(
'SK_ID_CURR').size()

    del payments_final
```

```
    gc.collect()

    return payments_final_agg
```

Function to carry out Feature Engineering using 'Days_Instalment'

In [155]:
```
def FE_installments_payments_days_instalment(data, data_temp, installments_payments):

    installments_payments['DAYS_ENTRY_PAYMENT'].fillna(0, inplace=True)
    installments_payments['AMT_PAYMENT'].fillna(0.0, inplace=True)

    temp_13 = FE_installments_payments(reduce_memory_usage(installments_payments))
    data = data_temp.join(temp_13, how='left', on='SK_ID_CURR')
    del temp_13
    gc.collect()

    temp_14 = reduce_memory_usage(installments_payments[installments_payments['DAYS_INSTALMENT']>=-365].reset_index())
    temp_14.drop(['index'], axis=1, inplace=True)
    temp_14 = FE_installments_payments(temp_14)
    data = data.join(temp_14, how='left', on='SK_ID_CURR', rsuffix='_year')
    del temp_14
    gc.collect()

    temp_15 = reduce_memory_usage(installments_payments[installments_payments['DAYS_INSTALMENT']>=-182].reset_index())
    temp_15.drop(['index'], axis=1, inplace=True)
    temp_15 = FE_installments_payments(temp_15)
    data = data.join(temp_15, how='left', on='SK_ID_CURR', rsuffix='_half_year')
    del temp_15
    gc.collect()

    temp_16 = reduce_memory_usage(installments_payments[installments_payments['DAYS_INSTALMENT']>=-90].reset_index())
```

```
    temp_16.drop(['index'], axis=1, inplace=True)
    temp_16 = FE_installments_payments(temp_16)
    data = data.join(temp_16, how='left', on='SK_ID_CURR', rsuffix='_qu
arter')
    del temp_16
    gc.collect()

    temp_17 = reduce_memory_usage(installments_payments[installments_pa
yments['DAYS_INSTALMENT']>=-30].reset_index())
    temp_17.drop(['index'], axis=1, inplace=True)
    temp_17 = FE_installments_payments(temp_17)
    data = data.join(temp_17, how='left', on='SK_ID_CURR', rsuffix='_mo
nth')
    del temp_17
    gc.collect()

    temp_18 = reduce_memory_usage(installments_payments[installments_pa
yments['DAYS_INSTALMENT']>=-14].reset_index())
    temp_18.drop(['index'], axis=1, inplace=True)
    temp_18 = FE_installments_payments(temp_18)
    data = data.join(temp_18, how='left', on='SK_ID_CURR', rsuffix='_fo
rtnight')
    del temp_18
    gc.collect()

    temp_19 = reduce_memory_usage(installments_payments[installments_pa
yments['DAYS_INSTALMENT']>=-7].reset_index())
    temp_19.drop(['index'], axis=1, inplace=True)
    temp_19 = FE_installments_payments(temp_19)
    data = data.join(temp_19, how='left', on='SK_ID_CURR', rsuffix='_we
ek')
    del temp_19
    gc.collect()

    return data
```

Carrying out Feature Engineering using the Functions Defined

In [156]:
```
train_data_temp_2 = FE_installments_payments_days_instalment(train_data
,train_data_temp_2,installments_payments)
train_data_temp_2.shape
```

```
Memory usage of dataframe is 311.40 MB
Memory usage after optimization is: 311.40 MB
Decreased by 0.0%
Memory usage of dataframe is 105.17 MB
Memory usage after optimization is: 92.03 MB
Decreased by 12.5%
Memory usage of dataframe is 51.33 MB
Memory usage after optimization is: 44.91 MB
Decreased by 12.5%
Memory usage of dataframe is 23.63 MB
Memory usage after optimization is: 20.67 MB
Decreased by 12.5%
Memory usage of dataframe is 6.77 MB
Memory usage after optimization is: 5.92 MB
Decreased by 12.5%
Memory usage of dataframe is 2.49 MB
Memory usage after optimization is: 2.17 MB
Decreased by 12.5%
Memory usage of dataframe is 0.96 MB
Memory usage after optimization is: 0.84 MB
Decreased by 12.5%
```

Out[156]:
```
(307511, 2384)
```

9. Credit Card Balance Dataset

9.1. Basic Overview of the 'Credit Card Balance' Dataset

In [157]:
```
credit_card_balance = reduce_memory_usage(pd.read_csv('home-credit-defa
ult-risk/credit_card_balance.csv'))
print('Number of data points : ', credit_card_balance.shape[0])
```

```
print('Number of features : ', credit_card_balance.shape[1])
credit_card_balance.head()
```

```
Memory usage of dataframe is 673.88 MB
Memory usage after optimization is: 289.33 MB
Decreased by 57.1%
Number of data points :  3840312
Number of features :  23
```

Out[157]:

	SK_ID_PREV	SK_ID_CURR	MONTHS_BALANCE	AMT_BALANCE	AMT_CREDIT_LIMIT_ACTUAL
0	2562384	378907	-6	56.970001	135000
1	2582071	363914	-1	63975.554688	45000
2	1740877	371185	-7	31815.224609	450000
3	1389973	337855	-4	236572.109375	225000
4	1891521	126868	-1	453919.468750	450000

5 rows × 23 columns

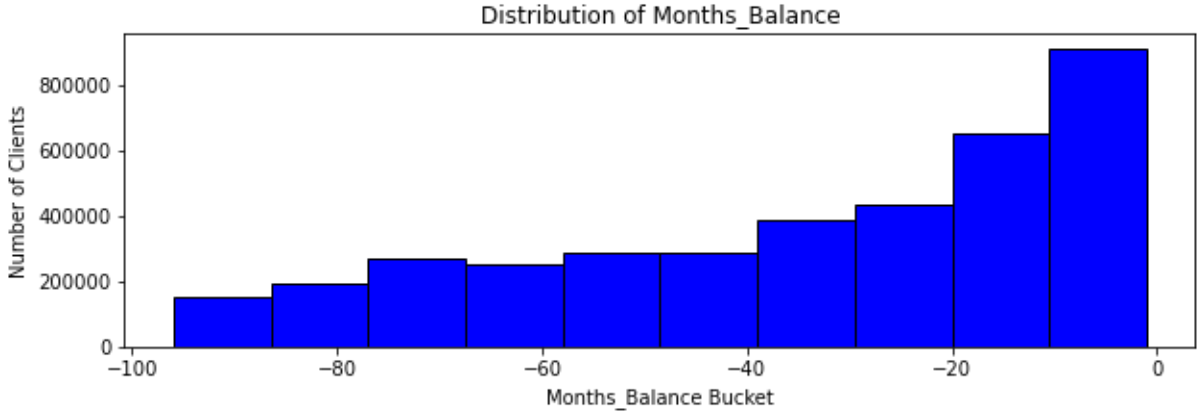
In [158]:
```
credit_card_balance.columns
```

Out[158]:
```
Index(['SK_ID_PREV', 'SK_ID_CURR', 'MONTHS_BALANCE', 'AMT_BALANCE',
       'AMT_CREDIT_LIMIT_ACTUAL', 'AMT_DRAWINGS_ATM_CURRENT',
       'AMT_DRAWINGS_CURRENT', 'AMT_DRAWINGS_OTHER_CURRENT',
       'AMT_DRAWINGS_POS_CURRENT', 'AMT_INST_MIN_REGULARITY',
       'AMT_PAYMENT_CURRENT', 'AMT_PAYMENT_TOTAL_CURRENT',
       'AMT_RECEIVABLE_PRINCIPAL', 'AMT_RECIVABLE', 'AMT_TOTAL_RECEIVAB
LE',
       'CNT_DRAWINGS_ATM_CURRENT', 'CNT_DRAWINGS_CURRENT',
       'CNT_DRAWINGS_OTHER_CURRENT', 'CNT_DRAWINGS_POS_CURRENT',
       'CNT_INSTALMENT_MATURE_CUM', 'NAME_CONTRACT_STATUS', 'SK_DPD',
       'SK_DPD_DEF'],
      dtype='object')
```

9.2 Credit Card Balance EDA

9.2.1 Univariate Analysis : Months_Balance

In [159]:
```
plt.figure(figsize=(10,3))
plt.hist(credit_card_balance['MONTHS_BALANCE'].values, bins=10, edgecol
or='black',\
         color='blue')
plt.title('Distribution of Months_Balance')
plt.xlabel('Months_Balance Bucket')
plt.ylabel('Number of Clients')
plt.show()
```



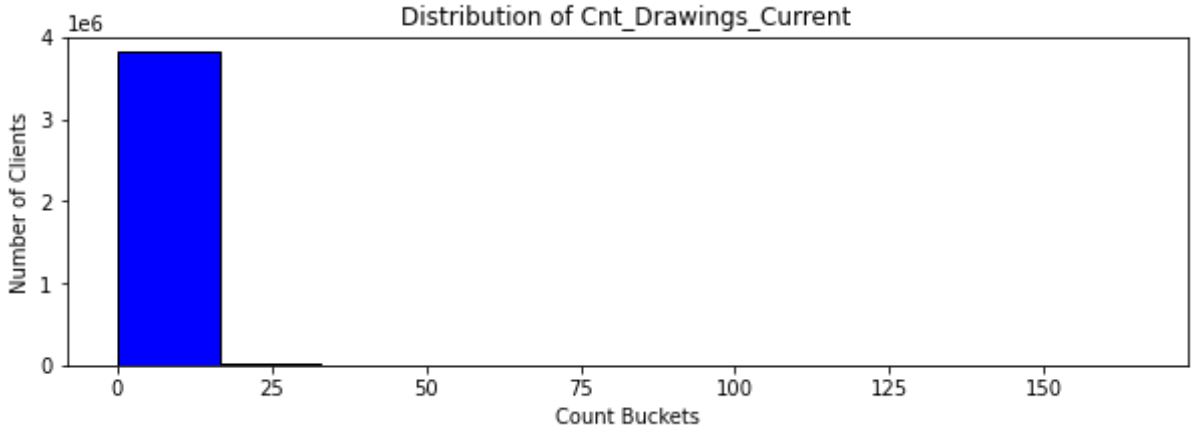
Observations :

- Most of the clients have Months_Balance between 0-10 months before the application date.

9.2.2 Univariate Analysis : Cnt_Drawings_Current

In [160]:
```
plt.figure(figsize=(10,3))
```

```
plt.hist(credit_card_balance['CNT_DRAWINGS_CURRENT'].values, bins=10, e
dgecolor='black',\
         color='blue')
plt.title('Distribution of Cnt_Drawings_Current')
plt.xlabel('Count Buckets')
plt.ylabel('Number of Clients')
plt.show()
```

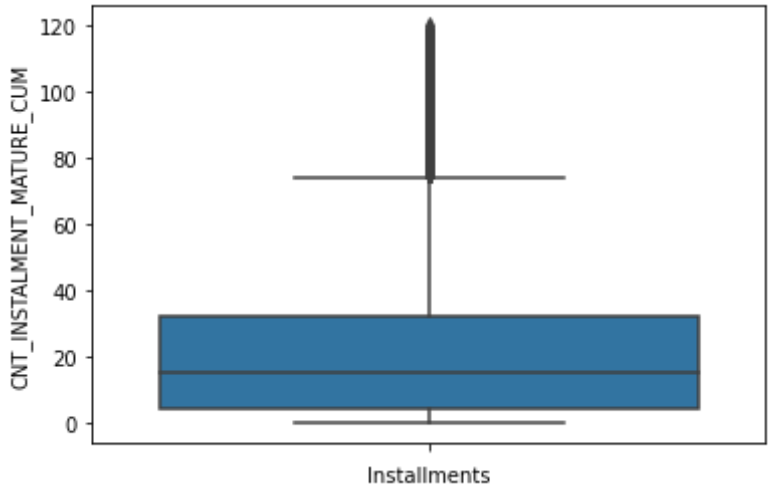


Observations :

- The vast majority of the clients have less than 25 months of drawing in the current month on the previous credit, except a very small number of outliers.

9.2.3 Univariate Analysis : Cnt_Instalment_Mature_Cum

In [161]:
```
sns.boxplot(y='CNT_INSTALMENT_MATURE_CUM', data=credit_card_balance)
plt.xlabel('Installments')
plt.show()
```



In [162]:
```
credit_card_balance['CNT_INSTALMENT_MATURE_CUM'].describe()
```

Out[162]:
```
count    3535076.0
mean           NaN
std            0.0
min            0.0
25%            4.0
50%           15.0
75%           32.0
max          120.0
Name: CNT_INSTALMENT_MATURE_CUM, dtype: float64
```

Observations :

- As we can see from the Boxplot and the detailed Feature Description of 'Cnt_Instalment_Mature_Cum' (number of paid instalments on the previous credit), the minimum value is 0 whereas the maximum value is 120.
- 75% of the total values lying are less than 32.

9.3 Feature Engineering on Credit Card Balance Data

Function to carry out Feature Engineering for Multiple features

In [163]:
```
def FE_credit_card_balance(credit_card_balance):

    cc_balance_data, cc_balance_columns = one_hot_encode(credit_card_ba
lance)
    cc_balance_data.rename(columns={'AMT_RECIVABLE': 'AMT_RECEIVABLE'},
inplace=True)

    cc_balance_data['LIMIT_USE'] = cc_balance_data['AMT_BALANCE'] / cc_
balance_data['AMT_CREDIT_LIMIT_ACTUAL']
    cc_balance_data['PAYMENT_DIV_MIN'] = cc_balance_data['AMT_PAYMENT_C
URRENT'] / cc_balance_data['AMT_INST_MIN_REGULARITY']
    cc_balance_data['LATE_PAYMENT'] = cc_balance_data['SK_DPD'].apply(l
ambda x: 1 if x > 0 else 0)

    cc_balance_data['DRAWING_LIMIT_RATIO'] = cc_balance_data['AMT_DRAWI
NGS_ATM_CURRENT'] / cc_balance_data['AMT_CREDIT_LIMIT_ACTUAL']

    cc_balance_data.drop(['SK_ID_PREV'], axis= 1, inplace = True)
    cc_balance_data_agg = cc_balance_data.groupby('SK_ID_CURR').agg(['m
ax', 'mean', 'sum', 'var'])

    col_list_9=[]

    for col in cc_balance_data_agg.columns.tolist():
        col_list_9.append('CR_'+col[0]+'_'+col[1].upper())

    cc_balance_data_agg.columns = pd.Index(col_list_9)

    cc_balance_data_agg['CREDIT_COUNT'] = cc_balance_data.groupby('SK_I
D_CURR').size()

    del cc_balance_data, cc_balance_columns
    gc.collect()

    return cc_balance_data_agg
```

Function to carry out Feature Engineering using 'Months_Balance'

In [164]:
```
def FE_credit_card_balance_months_balance(data,data_temp,credit_card_balance):

    temp_20 = FE_credit_card_balance(reduce_memory_usage(credit_card_balance))
    data = data_temp.join(temp_20, how='left', on='SK_ID_CURR')
    del temp_20
    gc.collect()

    temp_21 = reduce_memory_usage(credit_card_balance[credit_card_balance['MONTHS_BALANCE']>=-12].reset_index())
    temp_21.drop(['index'], axis=1, inplace=True)
    temp_21 = FE_credit_card_balance(temp_21)
    data = data.join(temp_21, how='left', on='SK_ID_CURR', rsuffix='_year')
    del temp_21
    gc.collect()

    temp_22 = reduce_memory_usage(credit_card_balance[credit_card_balance['MONTHS_BALANCE']>=-6].reset_index())
    temp_22.drop(['index'], axis=1, inplace=True)
    temp_22 = FE_credit_card_balance(temp_22)
    data = data.join(temp_22, how='left', on='SK_ID_CURR', rsuffix='_half_year')
    del temp_22
    gc.collect()

    temp_23 = reduce_memory_usage(credit_card_balance[credit_card_balance['MONTHS_BALANCE']>=-3].reset_index())
    temp_23.drop(['index'], axis=1, inplace=True)
    temp_23 = FE_credit_card_balance(temp_23)
    data = data.join(temp_23, how='left', on='SK_ID_CURR', rsuffix='_quarter')
    del temp_23
    gc.collect()
```

```
    temp_24 = reduce_memory_usage(credit_card_balance[credit_card_balan
ce['MONTHS_BALANCE']>=-1].reset_index())
    temp_24.drop(['index'], axis=1, inplace=True)
    temp_24 = FE_credit_card_balance(temp_24)
    data = data.join(temp_24, how='left', on='SK_ID_CURR', rsuffix='_mo
nth')
    del temp_24
    gc.collect()

    return data
```

Carrying out Feature Engineering using the Functions Defined

In [165]:
```
train_data_temp_2 = FE_credit_card_balance_months_balance(train_data,tr
ain_data_temp_2,credit_card_balance)
train_data_temp_2.shape
```

```
Memory usage of dataframe is 289.33 MB
Memory usage after optimization is: 289.33 MB
Decreased by 0.0%
Memory usage of dataframe is 88.60 MB
Memory usage after optimization is: 84.53 MB
Decreased by 4.6%
Memory usage of dataframe is 46.35 MB
Memory usage after optimization is: 44.22 MB
Decreased by 4.6%
Memory usage of dataframe is 21.35 MB
Memory usage after optimization is: 20.37 MB
Decreased by 4.6%
Memory usage of dataframe is 5.17 MB
Memory usage after optimization is: 4.94 MB
Decreased by 4.6%
```

Out[165]: (307511, 2981)

* Duplicate Feature Removal

In [166]:
```
#Removing any duplicate features, if any are present in the final datas
et
train_data = train_data_temp_2.loc[:,~train_data_temp_2.columns.duplica
ted()]
train_data.shape
```

Out[166]:
```
(307511, 2981)
```

10. Featurization on the Test Data

In [167]:
```
start = datetime.now()

test_data = fix_nulls_outliers(test_data)

test_data_temp_1 = FE_application_data(test_data)
bureau_data_balance_final = FE_bureau_data_2(bureau_data, bureau_balanc
e,bureau_data_columns,\
                                                bureau_balance_columns)
test_data_temp_2 = test_data_temp_1.join(bureau_data_balance_final, how
='left', on='SK_ID_CURR')

print("="*100)
test_data_temp_2 = FE_previous_application_days_decision(test_data, tes
t_data_temp_2,previous_application)
print("="*100)
print(" "*100)

print("="*100)
test_data_temp_2 = FE_pos_cash_balance_months_balance(test_data,test_da
ta_temp_2, pos_cash_balance)
print("="*100)
print(" "*100)

print("="*100)
test_data_temp_2 = FE_installments_payments_days_instalment(test_data,t
est_data_temp_2,installments_payments)
```

```
print("="*100)
print(" "*100)

print("="*100)
test_data_temp_2 = FE_credit_card_balance_months_balance(test_data,test
_data_temp_2,credit_card_balance)
print("="*100)
print(" "*100)

#Removing any duplicate features, if any are present in the final datas
et
test_data = test_data_temp_2.loc[:,~test_data_temp_2.columns.duplicated
()]

print(" "*100)
print("Time taken to run this cell :", datetime.now() - start)
```

```
====================================================================
============================
Memory usage of dataframe is 293.08 MB
Memory usage after optimization is: 293.08 MB
Decreased by 0.0%
Memory usage of dataframe is 105.06 MB
Memory usage after optimization is: 102.87 MB
Decreased by 2.1%
Memory usage of dataframe is 40.52 MB
Memory usage after optimization is: 39.67 MB
Decreased by 2.1%
Memory usage of dataframe is 16.19 MB
Memory usage after optimization is: 15.77 MB
Decreased by 2.6%
Memory usage of dataframe is 6.82 MB
Memory usage after optimization is: 6.65 MB
Decreased by 2.6%
Memory usage of dataframe is 3.22 MB
Memory usage after optimization is: 3.13 MB
Decreased by 2.6%
Memory usage of dataframe is 1.57 MB
Memory usage after optimization is: 1.53 MB
Decreased by 2.6%
```

```
================================================================================
==============================

================================================================================
==============================
Memory usage of dataframe is 238.45 MB
Memory usage after optimization is: 238.45 MB
Decreased by 0.0%
Memory usage of dataframe is 73.51 MB
Memory usage after optimization is: 64.60 MB
Decreased by 12.1%
Memory usage of dataframe is 33.01 MB
Memory usage after optimization is: 29.00 MB
Decreased by 12.1%
Memory usage of dataframe is 14.10 MB
Memory usage after optimization is: 12.39 MB
Decreased by 12.1%
Memory usage of dataframe is 2.99 MB
Memory usage after optimization is: 2.62 MB
Decreased by 12.1%
================================================================================
==============================

================================================================================
==============================
Memory usage of dataframe is 311.40 MB
Memory usage after optimization is: 311.40 MB
Decreased by 0.0%
Memory usage of dataframe is 105.17 MB
Memory usage after optimization is: 92.03 MB
Decreased by 12.5%
Memory usage of dataframe is 51.33 MB
Memory usage after optimization is: 44.91 MB
Decreased by 12.5%
Memory usage of dataframe is 23.63 MB
Memory usage after optimization is: 20.67 MB
Decreased by 12.5%
Memory usage of dataframe is 6.77 MB
Memory usage after optimization is: 5.92 MB
Decreased by 12.5%
```

```
Memory usage of dataframe is 2.49 MB
Memory usage after optimization is: 2.17 MB
Decreased by 12.5%
Memory usage of dataframe is 0.96 MB
Memory usage after optimization is: 0.84 MB
Decreased by 12.5%
================================================================================
==============================

================================================================================
==============================
Memory usage of dataframe is 289.33 MB
Memory usage after optimization is: 289.33 MB
Decreased by 0.0%
Memory usage of dataframe is 88.60 MB
Memory usage after optimization is: 84.53 MB
Decreased by 4.6%
Memory usage of dataframe is 46.35 MB
Memory usage after optimization is: 44.22 MB
Decreased by 4.6%
Memory usage of dataframe is 21.35 MB
Memory usage after optimization is: 20.37 MB
Decreased by 4.6%
Memory usage of dataframe is 5.17 MB
Memory usage after optimization is: 4.94 MB
Decreased by 4.6%
================================================================================
==============================


Time taken to run this cell : 0:16:02.958643
```

10. Before Building the ML Models

10.1. Train Test Split of the Data

- Before we build actual ML Models on our dataset, we try and carry out a Train-Test Split. The number of datapoints that we obtain in each of these datasets is as shown.

In [168]:
```
print('Shape of the Train Data: {}'.format(train_data.shape))
print('Shape of the Test Data: {}'.format(test_data.shape))
```

```
Shape of the Train Data: (307511, 2981)
Shape of the Test Data: (48744, 2979)
```

In [169]:
```
X_data_train = train_data.drop(['TARGET'], axis=1)
#For the X_data_train we select all the columns except the TARGET colum
n that has the class labels

Y_data_train = train_data['TARGET']
#For the Y_data_train, we select only the TARGET column
```

In [170]:
```
X_train_final, X_cv_final, Y_train_final, Y_cv_final = train_test_split
(X_data_train, Y_data_train, test_size=0.20, \

stratify=Y_data_train)
print(X_train_final.shape, Y_train_final.shape)
print(X_cv_final.shape, Y_cv_final.shape)
```

```
(246008, 2980) (246008,)
(61503, 2980) (61503,)
```

10.2. Pickling Dataframes obtained for Future Use

In [10]:
```
#Pickling the Dataframes obtained for future use:-

import pandas as pd
import pickle
import os

if not os.path.isfile('pickles/train_data.pkl'):
```

```
    train_data.to_pickle('pickles/train_data.pkl')
train_data = pd.read_pickle('pickles/train_data.pkl')

if not os.path.isfile('pickles/test_data'):
    test_data.to_pickle('pickles/test_data')
test_data = pd.read_pickle('pickles/test_data')

if not os.path.isfile('pickles/X_train_final_hcdr_new'):
    X_train_final.to_pickle('pickles/X_train_final_hcdr_new')
X_train_final_hcdr_new = pd.read_pickle('pickles/X_train_final_hcdr_ne
w')

if not os.path.isfile('pickles/Y_train_final_hcdr_new'):
    Y_train_final.to_pickle('pickles/Y_train_final_hcdr_new')
Y_train_final_hcdr_new = pd.read_pickle('pickles/Y_train_final_hcdr_ne
w')

if not os.path.isfile('pickles/X_cv_final_hcdr_new'):
    X_cv_final.to_pickle('pickles/X_cv_final_hcdr_new')
X_cv_final_hcdr_new = pd.read_pickle('pickles/X_cv_final_hcdr_new')

if not os.path.isfile('pickles/Y_cv_final_hcdr_new'):
    Y_cv_final.to_pickle('pickles/Y_cv_final_hcdr_new')
Y_cv_final_hcdr_new = pd.read_pickle('pickles/Y_cv_final_hcdr_new')

if not os.path.isfile('pickles/X_data_train'):
    X_data_train.to_pickle('pickles/X_data_train')
X_data_train = pd.read_pickle('pickles/X_data_train')

if not os.path.isfile('pickles/Y_data_train'):
    Y_data_train.to_pickle('pickles/Y_data_train')
Y_data_train = pd.read_pickle('pickles/Y_data_train')
```

10.3. Obtaining Dataframe from only Top 500 Most Important Features

In [11]:
```
X_train_final_arr = np.nan_to_num(X_train_final_hcdr_new)
```

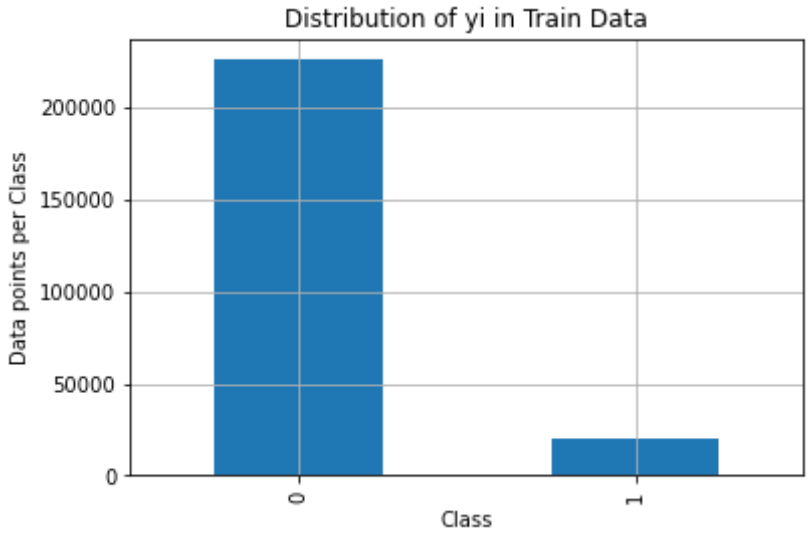
```
X_cv_final_arr = np.nan_to_num(X_cv_final_hcdr_new)

S = SelectKBest(f_classif, k=500)

X_train_k_best = S.fit_transform(X_train_final_arr, Y_train_final_hcdr_
new)
X_cv_k_best = S.transform(X_cv_final_arr)

# Get columns to keep and create new dataframe with those only
cols = S.get_support(indices=True)

features_top_df_train = X_train_final_hcdr_new.iloc[:,cols]
features_top_df_cv = X_cv_final_hcdr_new.iloc[:,cols]
```

In [12]:
```
if not os.path.isfile('pickles/features_top_df_train.pkl'):
    features_top_df_train.to_pickle('pickles/features_top_df_train.pkl'
)
features_top_df_train = pd.read_pickle('pickles/features_top_df_train.p
kl')
```

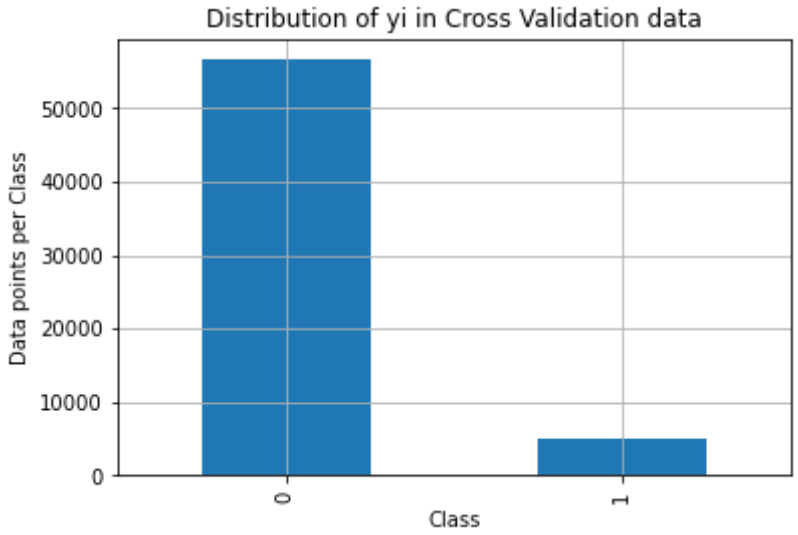
10.4. Distribution of y_i's in Train and Cross Validation datasets

In [5]:
```
# it returns a dict, keys as class labels and values as the number of d
ata points in that class
Y_train_final_hcdr_new = Y_train_final_hcdr_new.to_frame()
Y_cv_final_hcdr_new = Y_cv_final_hcdr_new.to_frame()

train_class_distribution = Y_train_final_hcdr_new['TARGET'].value_count
s().sort_index()
cv_class_distribution = Y_cv_final_hcdr_new['TARGET'].value_counts().so
rt_index()

my_colors = 'rgbkymc'
train_class_distribution.plot(kind='bar')
plt.xlabel('Class')
plt.ylabel('Data points per Class')
```

```
plt.title('Distribution of yi in Train Data')
plt.grid()
plt.show()

# ref: argsort https://docs.scipy.org/doc/numpy/reference/generated/num
py.argsort.html
# -(train_class_distribution.values): the minus sign will give us in de
creasing order
sorted_yi = np.argsort(-train_class_distribution.values)
for i in sorted_yi:
    print('Number of data points in class', i, ':',train_class_distribu
tion.values[i], \
          '(', np.round((train_class_distribution.values[i]/Y_train_final
_hcdr_new.shape[0]*100), 3), '%)')
print('-'*80)

my_colors = 'rgbkymc'
cv_class_distribution.plot(kind='bar')
plt.xlabel('Class')
plt.ylabel('Data points per Class')
plt.title('Distribution of yi in Cross Validation data')
plt.grid()
plt.show()

# ref: argsort https://docs.scipy.org/doc/numpy/reference/generated/num
py.argsort.html
# -(train_class_distribution.values): the minus sign will give us in de
creasing order
sorted_yi = np.argsort(-train_class_distribution.values)
for i in sorted_yi:
    print('Number of data points in class', i, ':',cv_class_distributio
n.values[i], \
            '(', np.round((cv_class_distribution.values[i]/Y_cv_final_hcd
r_new.shape[0]*100), 3), '%)')
```




```
Number of data points in class 0 : 226148 ( 91.927 %)
Number of data points in class 1 : 19860 ( 8.073 %)
----------------------------------------------------------------------
---------
```




```
Number of data points in class 0 : 56538 ( 91.927 %)
Number of data points in class 1 : 4965 ( 8.073 %)
```

10.5. Standardizing the Final Dataset Obtained

In [13]:
```
scaler = StandardScaler()

features_top_df_test = test_data[features_top_df_train.columns]
features_top_df_test_final = np.nan_to_num(features_top_df_test)
features_top_df_cv_final = np.nan_to_num(features_top_df_cv)

scaler.fit(features_top_df_train)
scaler_train = scaler.transform(features_top_df_train)
scaler_cv = scaler.transform(features_top_df_cv_final)
scaler_test = scaler.transform(features_top_df_test_final)

Y_train_final_hcdr_new = np.nan_to_num(Y_train_final_hcdr_new)
Y_cv_final_hcdr_new = np.nan_to_num(Y_cv_final_hcdr_new)

scaler_train = np.nan_to_num(scaler_train)
```

```
scaler_cv = np.nan_to_num(scaler_cv)
scaler_test = np.nan_to_num(scaler_test)
```

10.6. Additional Functions

Function to obtain predictions by Batches

In [14]:
```
def batch_predict(clf, data):

    y_data_pred = []
    loop_count = data.shape[0] - data.shape[0]%1000

    for i in range(0, loop_count, 1000):
        y_data_pred.extend(clf.predict_proba(data[i:i+1000])[:,1])

    y_data_pred.extend(clf.predict_proba(data[loop_count:])[:,1])

    return y_data_pred
```

Function to obtain Ideal value of Threshold

In [15]:
```
def obtain_threshold(thresholds,tpr,fpr):

    obtain_threshold.best_tradeoff = tpr*(1-fpr)
    ideal_threshold = thresholds[obtain_threshold.best_tradeoff.argmax
()]

    return ideal_threshold
```

Function to obtain the Confusion, Precision & Recall Matrices

In [16]:
```
def plot_confusion_matrix(test_y, predict_y):
```

```
    C = confusion_matrix(test_y, predict_y)

    A =(((C.T)/(C.sum(axis=1))).T)

    B =(C/C.sum(axis=0))

    plt.figure(figsize=(20,4))

    labels = [0,1]

    cmap=sns.light_palette("blue")
    plt.subplot(1, 3, 1)
    sns.set(font_scale=1.1)
    sns.set_style(style='white')

    sns.heatmap(C, annot=True, cmap=cmap, fmt=".10f", xticklabels=label
s, \
                yticklabels=labels)
    plt.xlabel('Predicted Class')
    plt.ylabel('Original Class')
    plt.title("Confusion matrix")

    plt.subplot(1, 3, 2)
    sns.heatmap(B,annot=True, cmap=cmap, fmt=".10f", xticklabels=labels
, \
                yticklabels=labels)
    plt.xlabel('Predicted Class')
    plt.ylabel('Original Class')
    plt.title("Precision matrix")

    plt.subplot(1, 3, 3)
    # representing B in heatmap format
    sns.heatmap(A, annot=True, cmap=cmap, fmt=".10f", xticklabels=label
s, \
                yticklabels=labels)
    plt.xlabel('Predicted Class')
    plt.ylabel('Original Class')
    plt.title("Recall matrix")

    plt.show()
```

11. Machine Learning Models

11.1. Logistic Regression

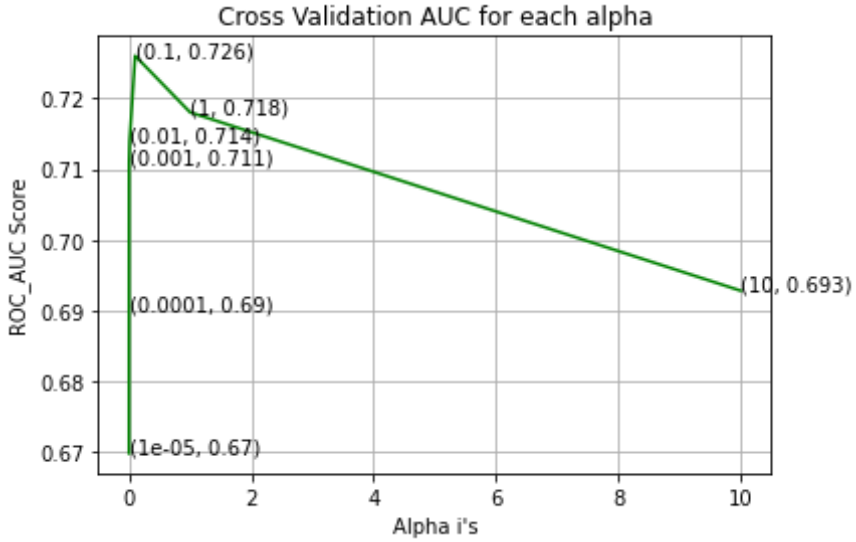
11.1.1 Hyperparameter Tuning on the CV Data

In [9]:
```
alpha = [10 ** x for x in range(-5, 2)] # hyperparameter for SGD classi
fier.

train_auc=[]
cv_auc=[]

roc_auc_array=[]

start = datetime.now()

roc_auc_array=[]
for i in alpha:
    clf = SGDClassifier(alpha=i, penalty='l2', loss='log', random_state
=42,\
                        class_weight='balanced')
    clf.fit(scaler_train, Y_train_final_hcdr_new)

    sig_clf = CalibratedClassifierCV(clf, method="sigmoid")
    sig_clf.fit(scaler_train, Y_train_final_hcdr_new)

    Y_train_pred = batch_predict(sig_clf, scaler_train)
    Y_cv_pred = batch_predict(sig_clf, scaler_cv)

    train_auc.append(roc_auc_score(Y_train_final_hcdr_new,Y_train_pred
))
    cv_auc.append(roc_auc_score(Y_cv_final_hcdr_new,Y_cv_pred))
```

```
    roc_auc_array.append(roc_auc_score(Y_cv_final_hcdr_new, Y_cv_pred))
    print('For values of alpha = ', i, "the roc_auc score is:", roc_auc
_score(Y_cv_final_hcdr_new,Y_cv_pred))

fig, ax = plt.subplots()
ax.plot(alpha, roc_auc_array,c='g')
for i, txt in enumerate(np.round(roc_auc_array,3)):
    ax.annotate((alpha[i],np.round(txt,3)), (alpha[i],roc_auc_array[i
]))
plt.grid()
plt.title("Cross Validation AUC for each alpha")
plt.xlabel("Alpha i's")
plt.ylabel("ROC_AUC Score")
plt.show()

best_alpha = np.argmax(roc_auc_array)
clf = SGDClassifier(alpha=alpha[best_alpha], penalty='l2', loss='log',
random_state=42)

clf.fit(scaler_train,  Y_train_final_hcdr_new)
sig_clf = CalibratedClassifierCV(clf, method="sigmoid")
sig_clf.fit(scaler_train, Y_train_final_hcdr_new)

predict_y_train = batch_predict(sig_clf,scaler_train)
print('For values of best alpha = ', alpha[best_alpha], "The train roc_
auc is:",\
                                    roc_auc_score(Y_train_final_hcdr_new,
predict_y_train))

predict_y_cv = batch_predict(sig_clf,scaler_cv)
print('For values of best alpha = ', alpha[best_alpha], "The cv roc_auc
is:",\
                                    roc_auc_score(Y_cv_final_hcdr_new, pr
edict_y_cv))

print(" "*100)
print("Time taken to run this cell :", datetime.now() - start)
```

```
For values of alpha =  1e-05 the roc_auc score is: 0.6697228970261497
For values of alpha =  0.0001 the roc_auc score is: 0.6900543002973485
```

```
For values of alpha =  0.001 the roc_auc score is: 0.7107942623017103
For values of alpha =  0.01 the roc_auc score is: 0.713965675110114
For values of alpha =  0.1 the roc_auc score is: 0.7260538973208654
For values of alpha =  1 the roc_auc score is: 0.7180474221955613
For values of alpha =  10 the roc_auc score is: 0.6927722149424976
```




```
For values of best alpha =  0.1 The train roc_auc is: 0.7452581142622051
For values of best alpha =  0.1 The cv roc_auc is: 0.7202674478539632

Time taken to run this cell : 0:11:59.721300
```

Observations :-

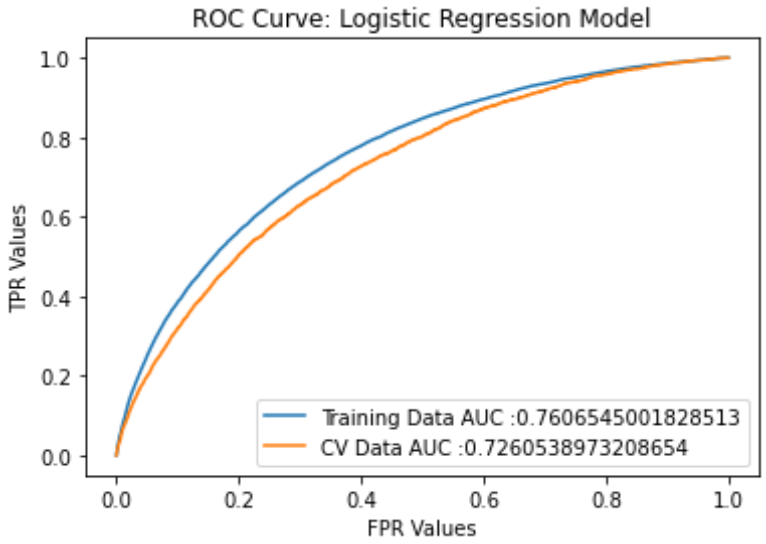
- In the roc_auc values that we obtained, since the values are quite close to each other for Train and CV, we can say with assurance that our model is not overfitting).

- We could have said that the model was underfitting if the Train and CV roc_auc values were approximately same as each other and had we trained a more complex model such as GBDT and this complex model gave an roc_auc score higher than 0.7202. (Since Logistic Regression is a very simple linear classifier it has a much higher chance of underfitting).

11.1.2 Obtaining ROC Curves on Train and CV Datasets

In [10]:
```
start = datetime.now()

clf = SGDClassifier(alpha=0.1, penalty='l2', loss='log', random_state=4
2,class_weight='balanced')
clf.fit(scaler_train, Y_train_final_hcdr_new)

sig_clf = CalibratedClassifierCV(clf, method="sigmoid")
sig_clf.fit(scaler_train, Y_train_final_hcdr_new)

train_fpr1, train_tpr1, tr_thresholds1 = roc_curve(Y_train_final_hcdr_n
ew,sig_clf.predict_proba(scaler_train)[:,1])
cv_fpr1,cv_tpr1, cv_thresholds1 = roc_curve(Y_cv_final_hcdr_new, sig_cl
f.predict_proba(scaler_cv)[:,1])

plt.plot(train_fpr1,train_tpr1, label ="Training Data AUC :" + str(auc(
train_fpr1,train_tpr1)))
plt.plot(cv_fpr1,cv_tpr1,label="CV Data AUC :" + str(auc(cv_fpr1,cv_tpr
1)))
plt.legend()

plt.xlabel("FPR Values")
plt.ylabel("TPR Values")
plt.title('ROC Curve: Logistic Regression Model')

plt.grid(False)
plt.show()

print('Ideal Threshold for the CV Dataset =', obtain_threshold(cv_thres
```

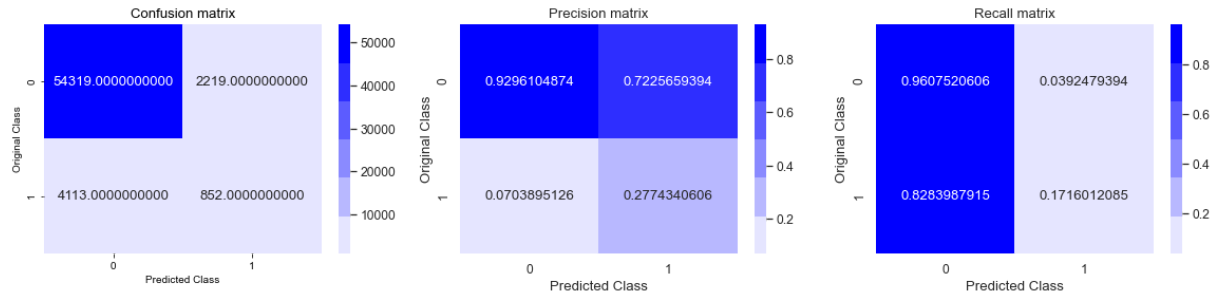
```
holds1,cv_tpr1,cv_fpr1))
print("Time taken to run this cell :", datetime.now() - start)
```




```
Ideal Threshold for the CV Dataset = 0.16685396420321394
Time taken to run this cell : 0:00:23.126684
```

11.1.3 Plotting Confusion, Precision & Recall Matrices on CV Data

In [11]:
```
Y_cv_final_plot = Y_cv_final_hcdr_new.flatten()
Y_cv_final_pred = sig_clf.predict(scaler_cv)

plot_confusion_matrix(Y_cv_final_plot, Y_cv_final_pred)
```



Observations :-

- Here, according to the Confusion Matrix we can see that the Accuracy in this case :- (54319 + 852)/61503 ie 89.70%.
- Precision Matrix :- In your Precision matrix the column sum = 1. It is saying that of all the points that are predicted to belong to class 0, 92.96% of them actually belong to class 0 and 7.03% of them belong to class 1. Similarly of all the points that are predicted to belong to class 1, 27.74% of the points belong to class 1 and 72.25% of the points actually belong to class 0.
- Recall Matrix :- In your recall matrix, the row sum = 1. Hence, here it says that for all the points that belong to class 0 our model predicted 96.07% of them belonging to class 0 and 3.9% of them belonging to class 1. Similarly of all the points that originally belong to class 1, 17.16% of those points have been predicted by the model to belong to class 1 and 82.83% to belong to class 0. {This class 1 Value of Recall is the major cause of concern in this case}. The diagonal values that you see in the Recall Matrix are the Recall Values for Class 0 and the Recall values for class 1.

11.1.4 Evaluating on the Test Dataset

In [12]:
```
from sklearn.linear_model import LogisticRegression

for col in features_top_df_test.columns:
    if col=='TARGET':
        features_top_df_test = features_top_df_test.drop(['TARGET'],axi
```

```
s=1)
    if col =='SK_ID_CURR':
        features_top_df_test = features_top_df_test.drop(['SK_ID_CURR'
],axis=1)

logistic_regression = LogisticRegression(C=0.1, class_weight='balanced'
,random_state=42,penalty='l2')
logistic_regression.fit(scaler_train, Y_train_final_hcdr_new)

scaler_test = np.nan_to_num(scaler_test)
lr_test_predict = logistic_regression.predict_proba(scaler_test)[:,1]
```

In [13]:
```
features_top_df_test['SK_ID_CURR'] = test_data['SK_ID_CURR']
features_top_df_test['TARGET'] = lr_test_predict
features_top_df_test['SK_ID_CURR'] = features_top_df_test['SK_ID_CURR']
.apply(lambda x: np.int32(x))
features_top_df_test[['SK_ID_CURR', 'TARGET']].to_csv('hcdr_logistic_re
gression_500f.csv', index= False)
```

Observations :-

- We can see from above that the AUC Value obtained on the Test Data using the Logistic Regression Model in the Kaggle Competition = 0.71840.

11.2. Linear SVM

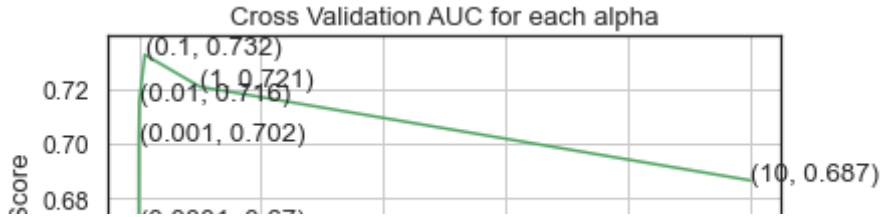
11.2.1 Hyperparameter Tuning on the CV Data

In [14]:
```
alpha = [10 ** x for x in range(-5, 2)] # hyperparam for SGD classifie
r.
```

```
train_auc=[]
cv_auc=[]

roc_auc_array=[]

start = datetime.now()

roc_auc_array=[]
for i in alpha:
    clf = SGDClassifier(alpha=i, penalty='l2', loss='hinge', random_sta
te=42,\
                        class_weight='balanced')
    clf.fit(scaler_train, Y_train_final_hcdr_new)

    sig_clf = CalibratedClassifierCV(clf, method="sigmoid")
    sig_clf.fit(scaler_train, Y_train_final_hcdr_new)

    Y_train_pred = batch_predict(sig_clf, scaler_train)
    Y_cv_pred = batch_predict(sig_clf, scaler_cv)

    train_auc.append(roc_auc_score(Y_train_final_hcdr_new,Y_train_pred
))
    cv_auc.append(roc_auc_score(Y_cv_final_hcdr_new,Y_cv_pred))

    roc_auc_array.append(roc_auc_score(Y_cv_final_hcdr_new, Y_cv_pred))
    print('For values of alpha = ', i, "the roc_auc score is:", roc_auc
_score(Y_cv_final_hcdr_new,Y_cv_pred))

fig, ax = plt.subplots()
ax.plot(alpha, roc_auc_array,c='g')
for i, txt in enumerate(np.round(roc_auc_array,3)):
    ax.annotate((alpha[i],np.round(txt,3)), (alpha[i],roc_auc_array[i
]))
plt.grid()
plt.title("Cross Validation AUC for each alpha")
plt.xlabel("Alpha i's")
plt.ylabel("ROC_AUC Score")
plt.show()

best_alpha = np.argmax(roc_auc_array)
```

```
clf = SGDClassifier(alpha=alpha[best_alpha], penalty='l2', loss='log',
random_state=42)

clf.fit(scaler_train,  Y_train_final_hcdr_new)
sig_clf = CalibratedClassifierCV(clf, method="sigmoid")
sig_clf.fit(scaler_train, Y_train_final_hcdr_new)

predict_y_train = batch_predict(sig_clf,scaler_train)
print('For values of best alpha = ', alpha[best_alpha], "The train roc_
auc is:",\
                                    roc_auc_score(Y_train_final_hcdr_new,
predict_y_train))

predict_y_cv = batch_predict(sig_clf,scaler_cv)
print('For values of best alpha = ', alpha[best_alpha], "The cv roc_auc
is:",\
                                    roc_auc_score(Y_cv_final_hcdr_new, pr
edict_y_cv))

print(" "*100)
print("Time taken to run this cell :", datetime.now() - start)
```

```
For values of alpha =  1e-05 the roc_auc score is: 0.5879952301149969
For values of alpha =  0.0001 the roc_auc score is: 0.6700757971262774
For values of alpha =  0.001 the roc_auc score is: 0.7015780134434978
For values of alpha =  0.01 the roc_auc score is: 0.7156245830901562
For values of alpha =  0.1 the roc_auc score is: 0.7323251938994804
For values of alpha =  1 the roc_auc score is: 0.7206848697898269
For values of alpha =  10 the roc_auc score is: 0.6866182311163463
```

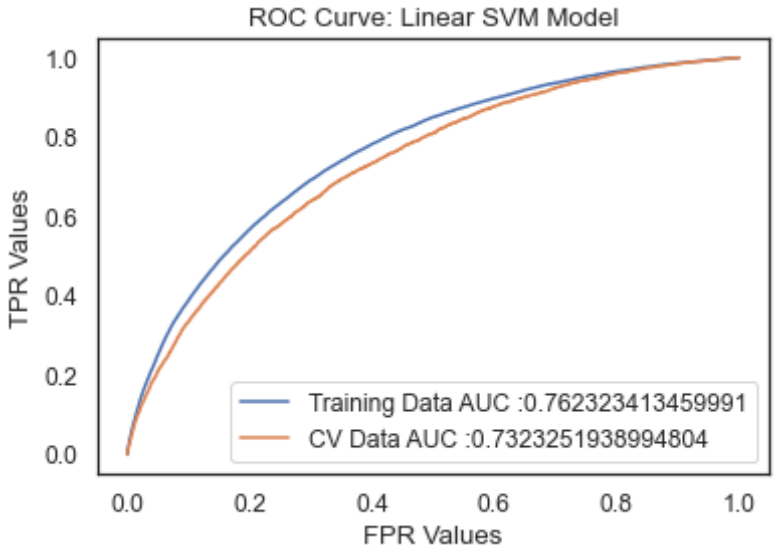



```
For values of best alpha =  0.1 The train roc_auc is: 0.745258114262205
1
For values of best alpha =  0.1 The cv roc_auc is: 0.7202674478539632

Time taken to run this cell : 0:12:15.194681
```

11.2.2 Obtaining ROC Curves on Train and Test Datasets

In [15]:
```
start = datetime.now()

svm = SGDClassifier(alpha=0.1, penalty='l2', loss='hinge', random_state
=42, class_weight='balanced')
svm.fit(scaler_train, Y_train_final_hcdr_new)

sig_svm = CalibratedClassifierCV(svm, method="sigmoid")
sig_svm.fit(scaler_train, Y_train_final_hcdr_new)

train_fpr2, train_tpr2, tr_thresholds2 = roc_curve(Y_train_final_hcdr_n
ew,sig_svm.predict_proba(scaler_train)[:,1])
cv_fpr2, cv_tpr2, cv_thresholds2 = roc_curve(Y_cv_final_hcdr_new, sig_s
vm.predict_proba(scaler_cv)[:,1])

plt.plot(train_fpr2,train_tpr2, label ="Training Data AUC :" + str(auc(
train_fpr2,train_tpr2)))
plt.plot(cv_fpr2,cv_tpr2,label="CV Data AUC :" + str(auc(cv_fpr2,cv_tpr
2)))
plt.legend()

plt.xlabel("FPR Values")
plt.ylabel("TPR Values")
```

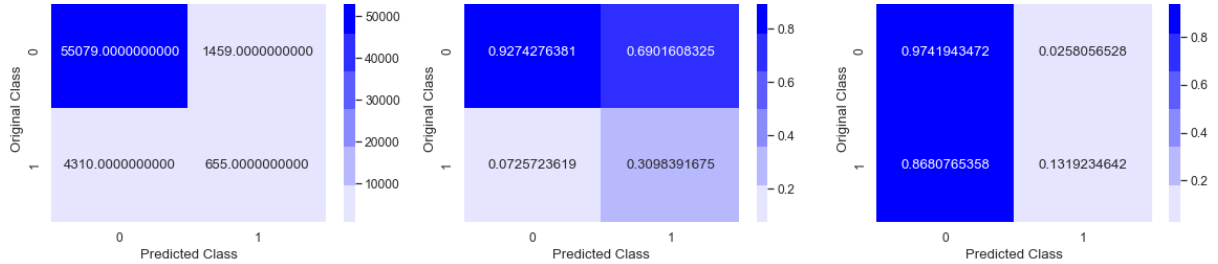
```
plt.title('ROC Curve: Linear SVM Model')

plt.grid(False)
plt.show()

print('Ideal Threshold for the CV Dataset =', obtain_threshold(cv_thres
holds2,cv_tpr2,cv_fpr2))
print("Time taken to run this cell :", datetime.now() - start)
```




```
Ideal Threshold for the CV Dataset = 0.13066317552738244
Time taken to run this cell : 0:00:23.222285
```

11.2.3 Plotting Confusion, Precision & Recall Matrices on Test Data

In [16]:
```
Y_cv_final_plot_2 = Y_cv_final_hcdr_new.flatten()
Y_cv_final_pred_2 = sig_svm.predict(scaler_cv)

plot_confusion_matrix(Y_cv_final_plot_2, Y_cv_final_pred_2)
```



Observations :-

- Here, according to the Confusion Matrix we can see that the Accuracy in this case :- (55079 + 665)/61503 ie 90.63%.
- Precision Matrix :- In your Precision matrix the column sum = 1. It is saying that of all the points that are predicted to belong to class 0, 92.74% of them actually belong to class 0 and 7.25% of them belong to class 1.
- Recall Matrix :- In your recall matrix, the row sum = 1. Hence, here it says that for all the points that belong to class 0, our model predicted 97.41% of them belonging to class 0 and 2.58% of them belonging to class 1. Similarly of all the points that originally belong to class 1, 13.19% of those points have been predicted by the model to belong to class 1 and 86.80% to belong to class 0. {This class 1 Value of Recall is the major cause of concern in this case}.

11.2.4 Evaluating on the Test Dataset

In [17]:
```
#Working with Linear SVM.

start = datetime.now()

for col in features_top_df_test.columns:
    if col=='TARGET':
        features_top_df_test = features_top_df_test.drop(['TARGET'],axi
s=1)
    if col =='SK_ID_CURR':
```

```
        features_top_df_test = features_top_df_test.drop(['SK_ID_CURR'
],axis=1)

from sklearn.svm import LinearSVC

linear_svm = LinearSVC(C=0.1, penalty='l2' ,class_weight='balanced', ra
ndom_state=42)
clf = CalibratedClassifierCV(linear_svm)
clf.fit(scaler_train, Y_train_final_hcdr_new)

scaler_test = np.nan_to_num(scaler_test)
svm_test_predict = clf.predict_proba(scaler_test)[:,1]

print("Time taken to run this cell :", datetime.now() - start)
```

```
Time taken to run this cell : 0:18:03.122834
```

In [18]:
```
features_top_df_test['SK_ID_CURR'] = test_data['SK_ID_CURR']
features_top_df_test['TARGET'] = svm_test_predict
features_top_df_test['SK_ID_CURR'] = features_top_df_test['SK_ID_CURR']
.apply(lambda x: np.int32(x))
features_top_df_test[['SK_ID_CURR', 'TARGET']].to_csv('hcdr_linear_svm_
500f.csv', index= False)
```

Observations :-

- We can see from above that the AUC Value obtained on the Test Data using the Linear SVM in the Kaggle Competition = 0.72262.

11.3. Random Forest Classifier

11.3.1 Hyperparameter Tuning using Random Search

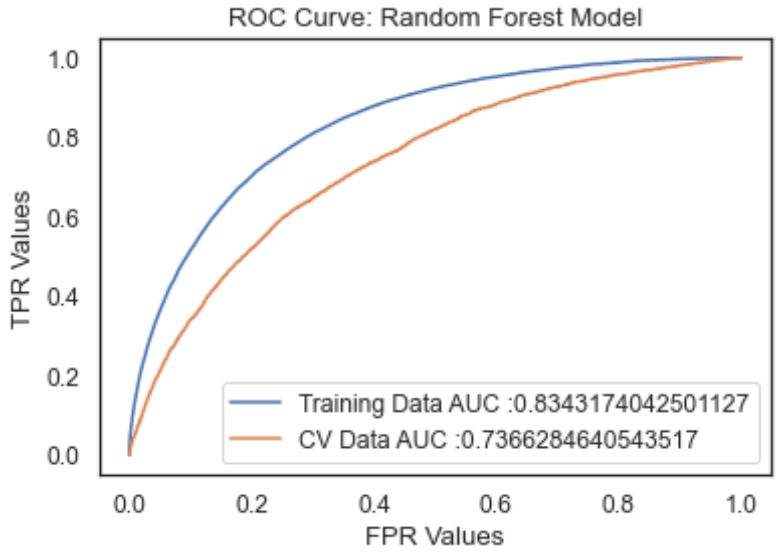
In [19]:
```
start = datetime.now()

rf_model = RandomForestClassifier(class_weight='balanced', random_state
=42)

params = {'n_estimators' : sp_randint(200,1000),
          'max_depth' : sp_randint(5,20)
          }

rf_random_search = RandomizedSearchCV(rf_model, param_distributions=par
ams, \
                                      scoring='roc_auc',\
                                     n_jobs=3, cv=3, verbose=10)
rf_random_search.fit(scaler_train,Y_train_final_hcdr_new)

print('\n Best hyperparameters:')
print(rf_random_search.best_params_)

#Best cross validation log loss obtained from hyperparameter tuning
print("Best ROC_AUC obtained on Cross Validation data using hyperparame
ter tuning: ",\
      rf_random_search.best_score_)

print("Time taken to run this cell :", datetime.now() - start)
```

```
Fitting 3 folds for each of 10 candidates, totalling 30 fits
[Parallel(n_jobs=3)]: Using backend LokyBackend with 3 concurrent worke
rs.
[Parallel(n_jobs=3)]: Done   2 tasks      | elapsed: 16.9min
[Parallel(n_jobs=3)]: Done   7 tasks      | elapsed: 60.2min
[Parallel(n_jobs=3)]: Done  12 tasks      | elapsed: 79.2min
[Parallel(n_jobs=3)]: Done  19 tasks      | elapsed: 166.6min
[Parallel(n_jobs=3)]: Done  30 out of  30 | elapsed: 602.8min finished

 Best hyperparameters:
{'max_depth': 10, 'n_estimators': 728}
Best ROC_AUC obtained on Cross Validation data using hyperparameter tun
ing:  0.7434272771192566
Time taken to run this cell : 10:44:12.658486
```

Time taken to run this cell : 10:44:12.050480

11.3.2 Obtaining ROC Curves on Train and CV Datasets

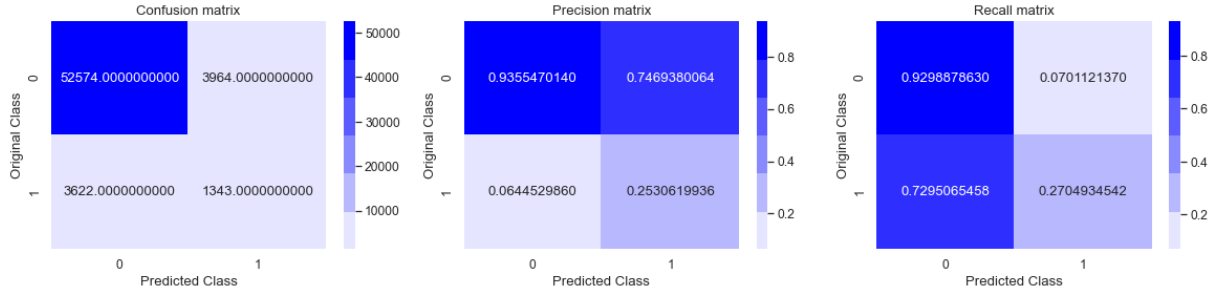
In [20]:
```
start = datetime.now()

rf_model = RandomForestClassifier(n_estimators=728, max_depth=10,class_
weight='balanced',\
                                  random_state=42)
rf_model.fit(scaler_train, Y_train_final_hcdr_new)

train_fpr3, train_tpr3, tr_thresholds3 = roc_curve(Y_train_final_hcdr_n
ew,rf_model.predict_proba(scaler_train)[:,1])
cv_fpr3,cv_tpr3, cv_thresholds3 = roc_curve(Y_cv_final_hcdr_new, rf_mod
el.predict_proba(scaler_cv)[:,1])

plt.plot(train_fpr3,train_tpr3, label ="Training Data AUC :" + str(auc(
train_fpr3,train_tpr3)))
plt.plot(cv_fpr3,cv_tpr3,label="CV Data AUC :" + str(auc(cv_fpr3,cv_tpr
3)))
plt.legend()

plt.xlabel("FPR Values")
plt.ylabel("TPR Values")
plt.title('ROC Curve: Random Forest Model')

plt.grid(False)
plt.show()

print('Ideal Threshold for the CV Dataset =', obtain_threshold(cv_thres
holds3,cv_tpr3,cv_fpr3))
print("Time taken to run this cell :", datetime.now() - start)
```




```
Ideal Threshold for the CV Dataset = 0.40767493794302595
Time taken to run this cell : 0:16:47.318354
```

11.3.3 Plotting Confusion, Precision & Recall Matrices on Test Data

In [21]:
```
Y_cv_final_plot_3 = Y_cv_final_hcdr_new.flatten()
Y_cv_final_pred_3 = rf_model.predict(scaler_cv)

plot_confusion_matrix(Y_cv_final_plot_3, Y_cv_final_pred_3)
```

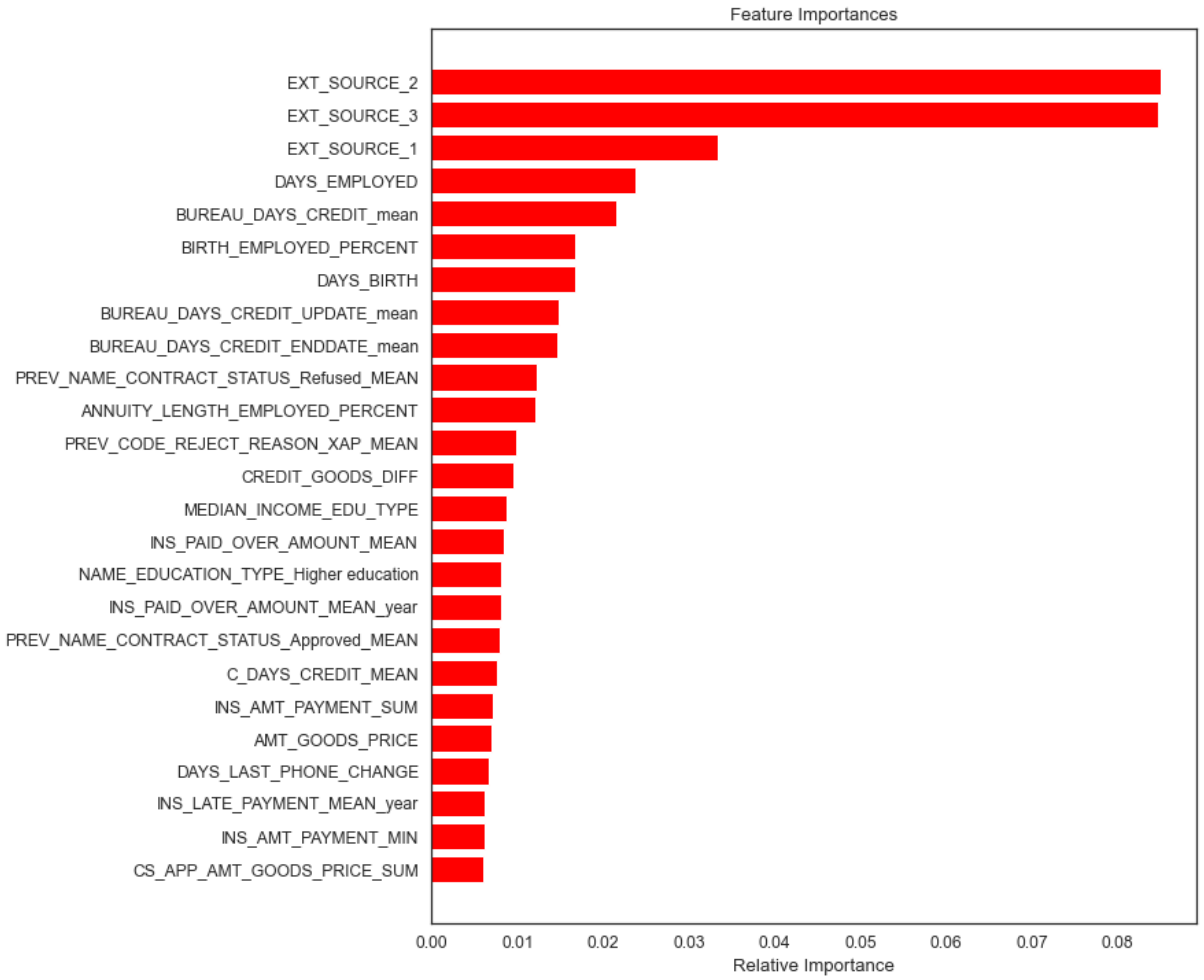


Observations :-

- Here, according to the Confusion Matrix we can see that the Accuracy in this case :- (52574 + 1343)/61503 ie 87.66%.
- Precision Matrix :- In your Precision matrix the column sum = 1. It is saying that of all the points that are predicted to belong to class 0, 93.55% of them actually belong to class 0 and 6.44% of them belong to class 1. Similarly of all the points that are predicted to belong to class 1, 25.30% of the points belong to class 1 and 74.69% of the points actually belong to class 0.
- Recall Matrix :- In your recall matrix, the row sum = 1. Hence, here it says that for all the points that belong to class 0, our model predicted 92.98% of them belonging to class 0 and 7.01% of them belonging to class 1. Similarly of all the points that originally belong to class 1, 27.04% of those points have been predicted by the model to belong to class 1 and 72.95% to belong to class 0.

11.3.4 Top 25 Features obtained from Random Forest Model

In [22]:
```
#Feature Importances for our Features

features = features_top_df_train.columns
importances = rf_model.feature_importances_
indices = (np.argsort(importances))[-25:]
plt.figure(figsize=(10,12))
plt.title('Feature Importances')
plt.barh(range(len(indices)), importances[indices], color='red', align=
'center')
plt.yticks(range(len(indices)), [features[i] for i in indices])
plt.xlabel('Relative Importance')
plt.show()
```



Observations :-

- According to the Feature Importances obtained from the Random Forest Classifier after Hyperparameter Tuning, 'EXT_SOURCE_2', 'EXT_SOURCE_3' and 'EXT_SOURCE_1' are the most important features in deciding the Loan repayment capability of a borrower.

11.3.5 Evaluating on the Test Dataset

In [25]:
```
#Working with Random Forest.

start = datetime.now()

for col in features_top_df_test.columns:
    if col=='TARGET':
        features_top_df_test = features_top_df_test.drop(['TARGET'],axi
s=1)
    if col =='SK_ID_CURR':
        features_top_df_test = features_top_df_test.drop(['SK_ID_CURR'
],axis=1)

random_forest = RandomForestClassifier(n_estimators=728, max_depth=10,\
                                       class_weight='balanced', random_s
tate=42)
random_forest.fit(scaler_train, Y_train_final_hcdr_new)

scaler_test = np.nan_to_num(scaler_test)
rf_test_predict = random_forest.predict_proba(scaler_test)[:,1]

print("Time taken to run this cell :", datetime.now() - start)
```

```
Time taken to run this cell : 0:16:14.616904
```

In [26]:
```
features_top_df_test['SK_ID_CURR'] = test_data['SK_ID_CURR']
features_top_df_test['TARGET'] = rf_test_predict
features_top_df_test['SK_ID_CURR'] = features_top_df_test['SK_ID_CURR']
.apply(lambda x: np.int32(x))
features_top_df_test[['SK_ID_CURR', 'TARGET']].to_csv('hcdr_random_fore
st_500f.csv', index= False)
```

Observations :-

- We can see from above that the AUC Value obtained on the Test Data using the Random Forest Classifier in the Kaggle Competition = 0.73097.

11.4. XGBoost Classifier

11.4.1 Obtaining Model Performance without any Hyperparameter Tuning

In [27]:
```
start = datetime.now()

import xgboost as xgb
xgb = xgb.XGBClassifier(objective='binary:logistic',eval_metric = 'auc'
, n_jobs=-1)
xgb.fit(scaler_train,Y_train_final_hcdr_new)

predict_y_train2 = batch_predict(xgb, scaler_train)
predict_y_cv2 = batch_predict(xgb, scaler_cv)

print("The Train roc_auc is:", roc_auc_score(Y_train_final_hcdr_new, pr
edict_y_train2))
print("The CV roc_auc is:", roc_auc_score(Y_cv_final_hcdr_new, predict_
y_cv2))
print(" "*100)

print("Time taken to run this cell :", datetime.now() - start)
```

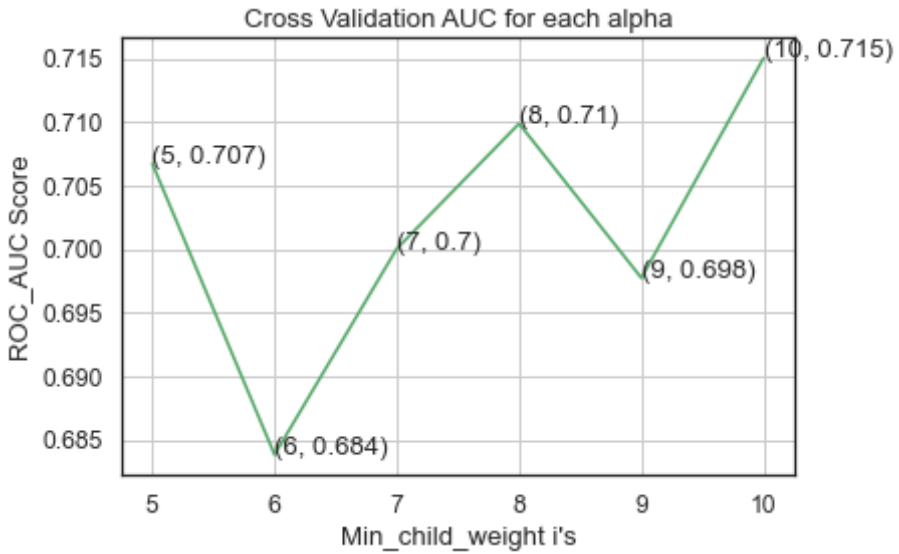
```
The Train roc_auc is: 0.8843125399561438
The CV roc_auc is: 0.648025867656068

Time taken to run this cell : 0:14:59.142044
```

11.4.2 Hyperparameter Tuning on min_child_weight on the CV Data

In [13]:
```
import xgboost as xgb
min_child_weight = [5,6,7,8,9,10] # hyperparam for SGD classifier.

train_auc=[]
cv_auc=[]

roc_auc_array=[]

start = datetime.now()

roc_auc_array=[]

for i in min_child_weight:

    import xgboost as xgb
    xgb1 = xgb.XGBClassifier(objective='binary:logistic', min_child_wei
ght=i,n_jobs=3)
    xgb1.fit(scaler_train,Y_train_final_hcdr_new)

    Y_train_pred = batch_predict(xgb1, scaler_train)
    Y_cv_pred = batch_predict(xgb1, scaler_cv)

    train_auc.append(roc_auc_score(Y_train_final_hcdr_new,Y_train_pred
))
    cv_auc.append(roc_auc_score(Y_cv_final_hcdr_new,Y_cv_pred))

    roc_auc_array.append(roc_auc_score(Y_cv_final_hcdr_new, Y_cv_pred))
    print('For values of min_child_weight = ', i, "the roc_auc score i
s:", roc_auc_score(Y_cv_final_hcdr_new,Y_cv_pred))

fig, ax = plt.subplots()
ax.plot(min_child_weight, roc_auc_array,c='g')
for i, txt in enumerate(np.round(roc_auc_array,3)):
    ax.annotate((min_child_weight[i],np.round(txt,3)), (min_child_weigh
t[i],roc_auc_array[i]))
plt.grid()
plt.title("Cross Validation AUC for each alpha")
plt.xlabel("Min_child_weight i's")
plt.ylabel("ROC_AUC Score")
plt.show()
```

```
best_weight = np.argmax(roc_auc_array)

xgb1 = xgb.XGBClassifier(objective='binary:logistic', min_child_weight=
best_weight,\
                         n_jobs=3)
xgb1.fit(scaler_train,Y_train_final_hcdr_new)

predict_y_train = batch_predict(xgb1,scaler_train)
print('For values of best min_child_weight = ', min_child_weight[best_w
eight], "The train roc_auc is:",\
                                  roc_auc_score(Y_train_final_hcdr_new,
predict_y_train))

predict_y_cv = batch_predict(xgb1,scaler_cv)
print('For values of best min_child_weight = ', min_child_weight[best_w
eight], "The cv roc_auc is:",\
                                  roc_auc_score(Y_cv_final_hcdr_new, pr
edict_y_cv))

print(" "*100)
print("Time taken to run this cell :", datetime.now() - start)
```

```
For values of min_child_weight =  5 the roc_auc score is: 0.70676433894
66831
For values of min_child_weight =  6 the roc_auc score is: 0.68383829542
65767
For values of min_child_weight =  7 the roc_auc score is: 0.70005872940
50322
For values of min_child_weight =  8 the roc_auc score is: 0.70990636567
82878
For values of min_child_weight =  9 the roc_auc score is: 0.69775157682
53896
For values of min_child_weight =  10 the roc_auc score is: 0.7150751500
198584
```



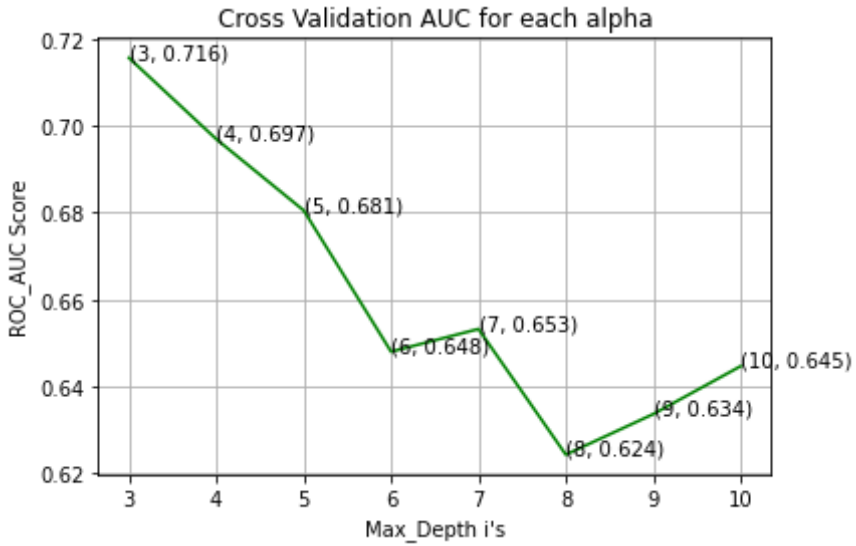

```
For values of best min_child_weight =  10 The train roc_auc is: 0.87578
68327803797
For values of best min_child_weight =  10 The cv roc_auc is: 0.70676433
89466831

Time taken to run this cell : 1:56:25.388689
```

11.4.3 Hyperparameter Tuning on max_depth on the CV Data

In [10]:
```
import xgboost as xgb
max_depth = [3,4,5,6,7,8,9,10] # hyperparam for SGD classifier.

train_auc=[]
cv_auc=[]

roc_auc_array=[]

start = datetime.now()

roc_auc_array=[]

for i in max_depth:

    import xgboost as xgb
    xgb2 = xgb.XGBClassifier(objective='binary:logistic', max_depth=i,\
                             n_jobs=3)
    xgb2.fit(scaler_train,Y_train_final_hcdr_new)

    Y_train_pred = batch_predict(xgb2, scaler_train)
    Y_cv_pred = batch_predict(xgb2, scaler_cv)

    train_auc.append(roc_auc_score(Y_train_final_hcdr_new,Y_train_pred
))
    cv_auc.append(roc_auc_score(Y_cv_final_hcdr_new,Y_cv_pred))

    roc_auc_array.append(roc_auc_score(Y_cv_final_hcdr_new, Y_cv_pred))
    print('For values of max_depth = ', i, "the roc_auc score is:", roc
_auc_score(Y_cv_final_hcdr_new,Y_cv_pred))

fig, ax = plt.subplots()
ax.plot(max_depth, roc_auc_array,c='g')
for i, txt in enumerate(np.round(roc_auc_array,3)):
    ax.annotate((max_depth[i],np.round(txt,3)), (max_depth[i],roc_auc_a
rray[i]))
plt.grid()
plt.title("Cross Validation AUC for each alpha")
plt.xlabel("Max_Depth i's")
plt.ylabel("ROC_AUC Score")
plt.show()
```

```
best_depth = np.argmax(roc_auc_array)

xgb2 = xgb.XGBClassifier(objective='binary:logistic', max_depth=best_de
pth,\
                         n_jobs=3)
xgb2.fit(scaler_train,Y_train_final_hcdr_new)

predict_y_train = batch_predict(xgb2,scaler_train)
print('For values of best max_depth = ', max_depth[best_depth], "The tr
ain roc_auc is:",\
                                    roc_auc_score(Y_train_final_hcdr_new,
predict_y_train))

predict_y_cv = batch_predict(xgb2,scaler_cv)
print('For values of best max_depth = ', max_depth[best_depth], "The cv
roc_auc is:",\
                                    roc_auc_score(Y_cv_final_hcdr_new, pr
edict_y_cv))

print(" "*100)
print("Time taken to run this cell :", datetime.now() - start)
```

```
For values of max_depth =  3 the roc_auc score is: 0.7156286406415534
For values of max_depth =  4 the roc_auc score is: 0.6969198571613663
For values of max_depth =  5 the roc_auc score is: 0.6805181211705968
For values of max_depth =  6 the roc_auc score is: 0.648025867656068
For values of max_depth =  7 the roc_auc score is: 0.6532371155732777
For values of max_depth =  8 the roc_auc score is: 0.62431790120785
For values of max_depth =  9 the roc_auc score is: 0.6336436629864071
For values of max_depth =  10 the roc_auc score is: 0.644670534485678
```

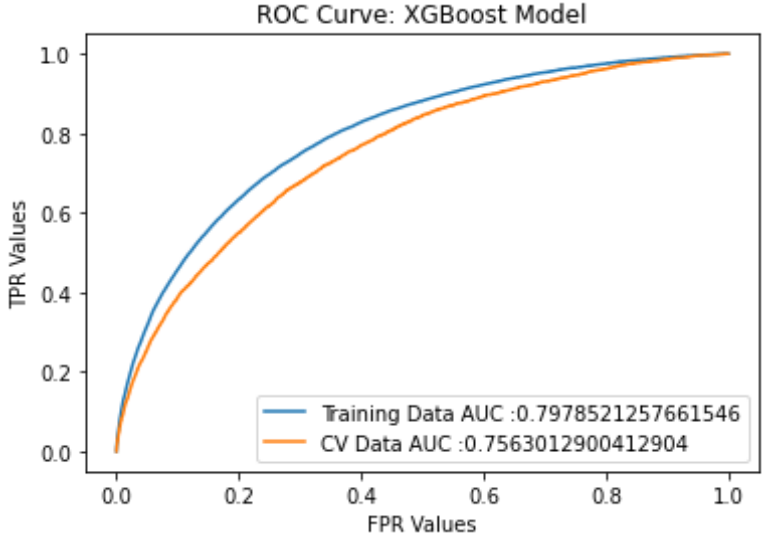



```
For values of best max_depth =  3 The train roc_auc is: 0.5
For values of best max_depth =  3 The cv roc_auc is: 0.5

Time taken to run this cell : 1:13:50.168236
```

11.4.4 Obtaining ROC Curves on Train and CV Datasets

In [11]:
```
start = datetime.now()

xgb_model = xgb.XGBClassifier(objective='binary:logistic',eval_metric =
'auc',\
                              min_child_weight=10, max_depth=3, n_jobs=-1)
xgb_model.fit(scaler_train, Y_train_final_hcdr_new)
```

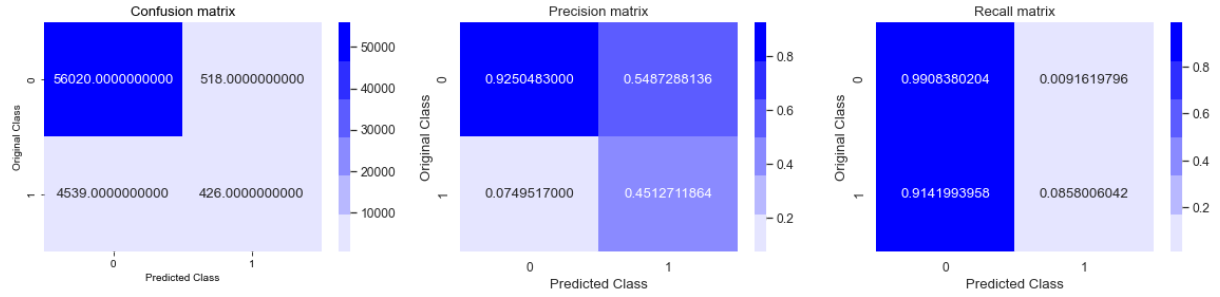
```
train_fpr4, train_tpr4, tr_thresholds4 = roc_curve(Y_train_final_hcdr_n
ew,xgb_model.predict_proba(scaler_train)[:,1])
cv_fpr4,cv_tpr4, cv_thresholds4 = roc_curve(Y_cv_final_hcdr_new, xgb_mo
del.predict_proba(scaler_cv)[:,1])

plt.plot(train_fpr4,train_tpr4, label ="Training Data AUC :" + str(auc(
train_fpr4,train_tpr4)))
plt.plot(cv_fpr4,cv_tpr4,label="CV Data AUC :" + str(auc(cv_fpr4,cv_tpr
4)))
plt.legend()

plt.xlabel("FPR Values")
plt.ylabel("TPR Values")
plt.title('ROC Curve: XGBoost Model')

plt.grid(False)
plt.show()

print('Ideal Threshold for the CV Dataset =', obtain_threshold(cv_thres
holds4,cv_tpr4,cv_fpr4))
print("Time taken to run this cell :", datetime.now() - start)
```



```
Ideal Threshold for the CV Dataset = 0.11629721
Time taken to run this cell : 0:09:23.631217
```

11.4.5 Plotting Confusion, Precision & Recall Matrices on CV Data

In [12]:
```
Y_cv_final_plot_4 = Y_cv_final_hcdr_new
Y_cv_final_pred_4 = xgb_model.predict(scaler_cv)

plot_confusion_matrix(Y_cv_final_plot_4, Y_cv_final_pred_4)
```

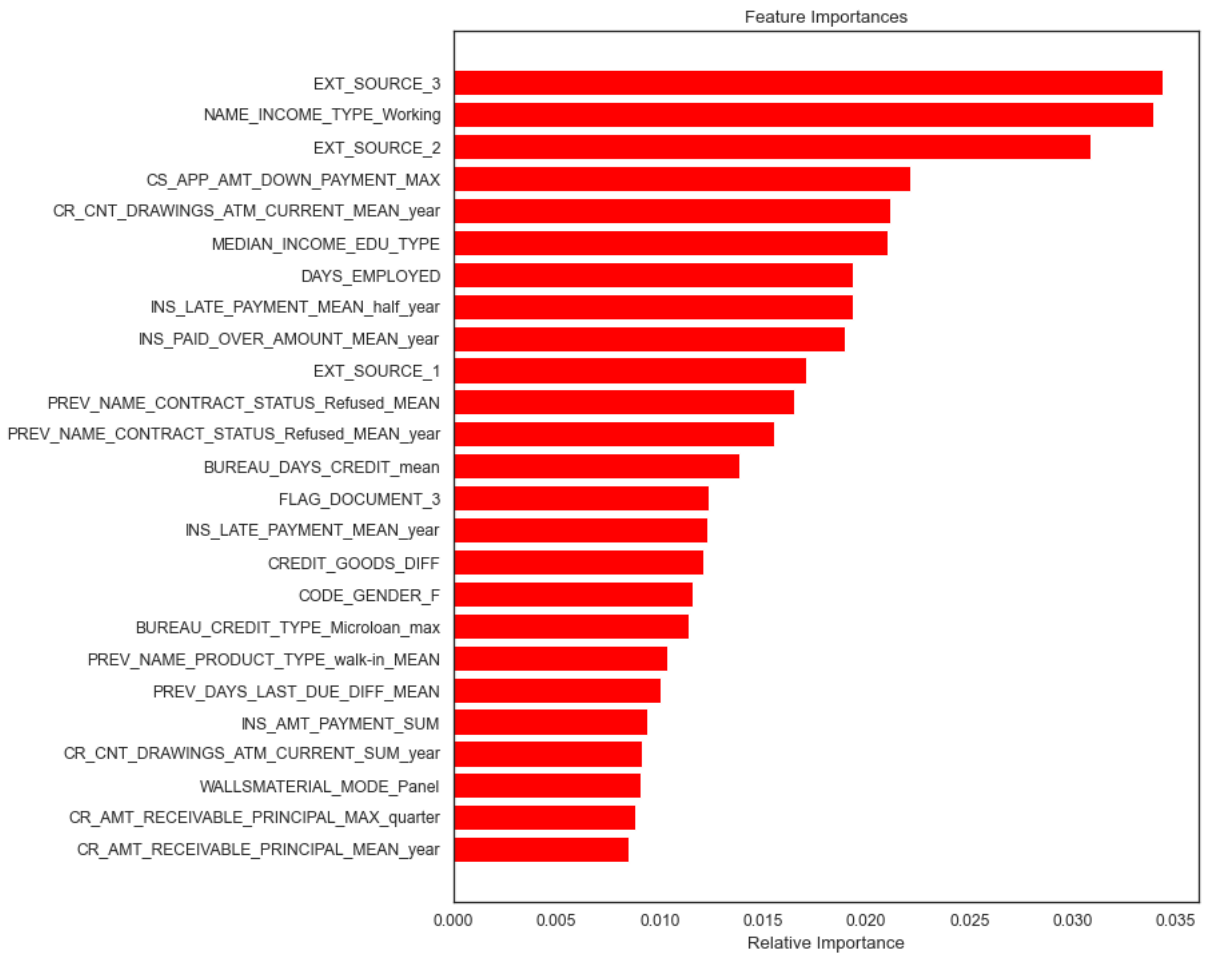


Observations :-

- Here, according to the Confusion Matrix we can see that the Accuracy in this case :- (56020 + 426)/61503 ie 91.77%.
- Precision Matrix :- In your Precision matrix the column sum = 1. It is saying that of all the points that are predicted to belong to class 0, 92.50% of them actually belong to class 0 and 7.49% of them belong to class 1. Similarly of all the points that are predicted to belong to class 1, 45.12% of the points belong to class 1 and 54.87% of the points actually belong to class 0.
- Recall Matrix :- In your recall matrix, the row sum = 1. Hence, here it says that for all the points that belong to class 0, our model predicted 99.08% of them belonging to class 0 and 0.9% of them belonging to class 1. Similarly of all the points that originally belong to class 1, 8.58% of those points have been predicted by the model to belong to class 1 and 91.41% to belong to class 0. {This class 1 Value of Recall is the major cause of concern in this case as well}.

11.4.6 Top 25 Features obtained from XGBoost Model

In [13]:
```
#Feature Importances for our Features

features = features_top_df_train.columns
importances = xgb_model.feature_importances_
indices = (np.argsort(importances))[-25:]
plt.figure(figsize=(10,12))
plt.title('Feature Importances')
plt.barh(range(len(indices)), importances[indices], color='red', align=
'center')
plt.yticks(range(len(indices)), [features[i] for i in indices])
plt.xlabel('Relative Importance')
plt.show()
```



Observations :-

- According to the Feature Importances obtained from the XGBoost Classifier after Hyperparameter Tuning, 'EXT_SOURCE_3', 'NAME_INCOME_TYPE_Working' and 'EXT_SOURCE_2' are the most important features in deciding the Loan repayment capability of a borrower.

11.4.7 Evaluating on the Test Dataset

In [14]:
```
#Working with XGBoost

start = datetime.now()

for col in features_top_df_test.columns:
    if col=='TARGET':
        features_top_df_test = features_top_df_test.drop(['TARGET'],axi
s=1)
    if col =='SK_ID_CURR':
        features_top_df_test = features_top_df_test.drop(['SK_ID_CURR'
],axis=1)

xgb_model = xgb.XGBClassifier(objective='binary:logistic',eval_metric =
'auc',\
                              min_child_weight=10, max_depth=3, n_jobs=-1)
xgb_model.fit(scaler_train, Y_train_final_hcdr_new)

scaler_test = np.nan_to_num(scaler_test)
xgb_test_predict = xgb_model.predict_proba(scaler_test)[:,1]

print("Time taken to run this cell :", datetime.now() - start)
```

```
Time taken to run this cell : 0:08:16.694771
```

In [15]:
```
features_top_df_test['SK_ID_CURR'] = test_data['SK_ID_CURR']
features_top_df_test['TARGET'] = xgb_test_predict
features_top_df_test['SK_ID_CURR'] = features_top_df_test['SK_ID_CURR']
.apply(lambda x: np.int32(x))
features_top_df_test[['SK_ID_CURR', 'TARGET']].to_csv('hcdr_xgboost_500
f.csv', index= False)
```

Observations :-

- We can see from above that the AUC Value obtained on the Test Data using the XGB Classifier in the Kaggle Competition = 0.73805.

11.5. Light GBM

11.5.1 Hyperparameter Tuning using Bayesian Optimization

In [9]:
```
start = datetime.now()

y_f = Y_train_final_hcdr_new.tolist()
y = [item for sublist in y_f for item in sublist]
features_top_df_train = features_top_df_train.rename(columns = \
                                                     lambda x:re.sub('[^A-Za-z0-9_]
+', '', x))
data = lgb.Dataset(data=features_top_df_train, label=y)

def parameters(num_iterations,bagging_fraction,lambda_l1,lambda_l2,num_
leaves,\
               learning_rate, feature_fraction, max_depth, min_split_ga
in, \
               min_child_weight, min_child_samples,subsample,scale_pos_
weight):

    params = {'application':'binary','early_stopping_round':300, 'metri
c':'auc',\
              'verbose':-1}
    params['num_iterations']=int(round(num_iterations))
    params['bagging_fraction'] = bagging_fraction
    params['lambda_l1'] = lambda_l1
    params['lambda_l2'] = lambda_l2
    params["num_leaves"] = int(round(num_leaves))
    params["learning_rate"] = learning_rate
    params['feature_fraction'] = max(min(feature_fraction, 1), 0)
    params['max_depth'] = int(round(max_depth))
    params['min_split_gain'] = min_split_gain
```

```
    params['min_child_weight'] = min_child_weight
    params['subsample'] = subsample
    params['scale_pos_weight'] = scale_pos_weight

    cv_result = lgb.cv(params, data, nfold=5,seed=6, stratified=True, m
etrics=['auc'])
    return max(cv_result['auc-mean'])

optimizer = BayesianOptimization(parameters, {'num_iterations':(7500,13
000),
                                              'bagging_fraction':(0.7,
1.0),
                                              'lambda_l1': (0, 5),
                                             'lambda_l2': (0, 3),
                                             'num_leaves':(12,55),
                                             'learning_rate':(0.009,0.05
),
                                             'feature_fraction':(0.01,0.
6),
                                             'max_depth': (5,22),
                                             'min_split_gain':(0.001,0.3
),
                                             'min_child_weight': (5,60),
                                             'min_child_samples':(30,65
),
                                             'subsample':(0.7,1.0),
                                             'scale_pos_weight':(0.7,11.
38)
                                              })

optimizer.maximize(init_points=20, n_iter=15)

print(optimizer.max)
print(" "*80)
print("Time taken to run this cell :", datetime.now() - start)
```

```
|   iter    |  target   | baggin... | featur... | lambda_l1 | lambda_l2
| learni... | max_depth | min_ch... | min_ch... | min_sp... | num_it...
| num_le... | scale_... | subsample |
-------------------------------------------------------------------------
```

```
-------------------------------------------------------------------
---------------------------------------
[LightGBM] [Warning] bagging_fraction is set=0.8155287920818188, subsam
ple=0.7514269219032764 will be ignored. Current value: bagging_fraction
=0.8155287920818188
|  1        |  0.7809   |  0.8155   |  0.1113   |  3.585    |  0.9447
|  0.02755  |  6.788    |  32.07    |  21.9     |  0.04056  |  9.186e+0
|  35.68    |  1.53     |  0.7514   |
|  2        |  0.7798   |  0.7653   |  0.08429  |  1.962    |  2.993
|  0.01583  |  9.61     |  55.94    |  59.82    |  0.2523   |  1.288e+0
|  37.48    |  6.604    |  0.9606   |
|  3        |  0.7793   |  0.8802   |  0.4889   |  0.7492   |  2.299
|  0.04506  |  16.96    |  38.71    |  39.17    |  0.2769   |  1.138e+0
|  39.58    |  9.917    |  0.7251   |
|  4        |  0.7813   |  0.7766   |  0.2689   |  2.395    |  1.61
|  0.03154  |  7.235    |  46.06    |  23.08    |  0.1481   |  9.472e+0
|  29.85    |  3.908    |  0.819    |
|  5        |  0.7795   |  0.9485   |  0.5123   |  0.2951   |  0.6499
|  0.0154   |  6.771    |  52.33    |  17.89    |  0.2977   |  1.005e+0
|  50.72    |  8.379    |  0.9726   |
|  6        |  0.7799   |  0.8022   |  0.5487   |  4.7      |  0.09615
|  0.04038  |  17.75    |  33.26    |  53.63    |  0.09421  |  1.21e+04
|  24.88    |  11.12    |  0.7778   |
|  7        |  0.7811   |  0.9181   |  0.5305   |  0.3665   |  2.887
|  0.02673  |  19.18    |  55.56    |  16.5     |  0.02525  |  1.165e+0
|  34.9     |  1.676    |  0.9377   |
|  8        |  0.7819   |  0.9256   |  0.5509   |  3.484    |  0.07585
|  0.01255  |  10.89    |  47.34    |  20.12    |  0.2904   |  9.373e+0
|  35.48    |  1.724    |  0.9337   |
|  9        |  0.7811   |  0.8229   |  0.4518   |  2.974    |  1.502
|  0.02851  |  21.41    |  43.24    |  26.45    |  0.2844   |  9.243e+0
|  25.2     |  4.384    |  0.8717   |
|  10       |  0.7798   |  0.9931   |  0.4713   |  1.092    |  1.487
|  0.03088  |  6.319    |  31.67    |  47.71    |  0.2982   |  9.162e+0
|  42.19    |  11.07    |  0.8323   |
|  11       |  0.781    |  0.908    |  0.2735   |  3.911    |  1.657
|  0.04263  |  14.11    |  47.79    |  52.01    |  0.1228   |  8.406e+0
|  35.64    |  2.251    |  0.8349   |
|  12       |  0.7806   |  0.7014   |  0.3987   |  0.8929   |  2.587
|  0.03999  |  18.32    |  44.83    |  49.75    |  0.02101  |  8.485e+0
```

37.86	5.689	0.73			
13	0.7822	0.8238	0.4176	1.206	2.089
0.01105	15.64	46.72	49.15	0.02466	1.266e+0
24.5	2.238	0.9548			
14	0.7793	0.8229	0.08138	1.909	1.73
0.02679	9.467	63.89	5.579	0.1504	1.093e+0
43.99	5.822	0.8794			
15	0.7798	0.8573	0.451	1.008	0.5429
0.0391	9.145	56.81	51.02	0.04689	1.286e+0
38.67	9.186	0.9872			
16	0.7791	0.7211	0.5396	2.128	2.615
0.03389	6.471	62.5	35.98	0.04231	7.907e+0
46.15	10.13	0.7171			
17	0.7814	0.879	0.4014	2.056	1.192
0.01178	21.92	44.59	36.94	0.085	1.118e+0
51.05	6.775	0.7378			
18	0.7805	0.9261	0.5667	3.8	2.521
0.03287	6.805	41.22	27.65	0.1361	7.648e+0
45.32	4.548	0.7162			
19	0.7811	0.957	0.4904	0.06897	2.503
0.01623	7.932	50.67	6.586	0.2595	8.57e+03
52.97	1.829	0.868			
20	0.7813	0.8397	0.3233	4.776	1.509
0.04224	18.42	60.79	43.59	0.2745	1.252e+0
12.56	3.27	0.9886			
21	0.7803	0.9169	0.4447	0.7388	2.218
0.02292	21.38	44.22	40.35	0.135	1.117e+0
50.11	10.58	0.7696			
22	0.7812	0.7653	0.2551	2.709	0.5068
0.02587	20.75	44.82	18.38	0.2682	1.136e+0
31.25	4.713	0.831			
23	0.7825	0.7327	0.1724	4.096	1.687
0.009023	20.1	49.91	51.97	0.192	1.084e+0
34.8	6.014	0.938			
24	0.7824	0.7627	0.5741	3.907	1.97
0.01227	5.23	38.16	46.35	0.2111	1.217e+0
48.39	1.093	0.8684			
25	0.7792	0.7112	0.5927	4.028	2.126
0.04347	17.48	50.33	6.808	0.07839	9.915e+0
42.56	10.36	0.7421			

```
|  26       |  0.7691   |  0.7759   |  0.03168  |  0.1264   |  0.3755
|  0.02729  |  15.02    |  48.8     |  27.13    |  0.1832   |  1.151e+0
|  50.83    |  7.434    |  0.9862   |
|  27       |  0.7808   |  0.7646   |  0.1659   |  1.298    |  0.4545
|  0.04776  |  12.48    |  44.78    |  43.42    |  0.2844   |  1.266e+0
|  21.58    |  3.825    |  0.9173   |
|  28       |  0.7807   |  0.7941   |  0.5177   |  1.305    |  2.243
|  0.03859  |  19.31    |  41.9     |  56.43    |  0.03998  |  1.268e+0
|  41.16    |  3.279    |  0.9446   |
|  29       |  0.7803   |  0.922    |  0.3362   |  2.132    |  0.08639
|  0.02309  |  6.026    |  47.71    |  22.37    |  0.2923   |  9.471e+0
|  33.31    |  8.814    |  0.7901   |
|  30       |  0.7806   |  0.8658   |  0.277    |  1.976    |  2.39
|  0.03676  |  9.889    |  42.98    |  31.01    |  0.1992   |  7.651e+0
|  50.66    |  3.016    |  0.7733   |
|  31       |  0.7819   |  0.9344   |  0.4755   |  1.278    |  1.993
|  0.01471  |  21.57    |  54.55    |  53.98    |  0.1755   |  1.265e+0
|  43.72    |  2.588    |  0.7542   |
|  32       |  0.7822   |  0.7111   |  0.2593   |  2.905    |  2.743
|  0.02136  |  15.02    |  46.69    |  58.44    |  0.1432   |  1.268e+0
|  40.91    |  2.804    |  0.7287   |
|  33       |  0.7815   |  0.9699   |  0.5541   |  2.281    |  2.016
|  0.01024  |  10.68    |  38.4     |  29.45    |  0.06337  |  7.651e+0
|  54.26    |  3.59     |  0.8362   |
|  34       |  0.7822   |  0.7618   |  0.1508   |  4.276    |  0.8881
|  0.009587 |  8.331    |  34.82    |  16.5     |  0.1039   |  9.186e+0
|  38.5     |  4.992    |  0.9598   |
|  35       |  0.7818   |  0.905    |  0.2205   |  0.3664   |  2.469
|  0.01944  |  21.04    |  43.48    |  49.69    |  0.1708   |  8.488e+0
|  34.38    |  4.501    |  0.764    |
=============================================================================
=============================================================================
=======================================
{'target': 0.7825084368002166, 'params': {'bagging_fraction': 0.7327318
230470493, 'feature_fraction': 0.1723525427151981, 'lambda_l1': 4.09552
8031147774, 'lambda_l2': 1.6872110950694539, 'learning_rate': 0.0090231
14465955174, 'max_depth': 20.099988455574476, 'min_child_samples': 49.9
1293761533297, 'min_child_weight': 51.96557354962844, 'min_split_gain':
0.19202267895386566, 'num_iterations': 10837.176310846433, 'num_leave
s': 34.804849657579354, 'scale_pos_weight': 6.014158781875895, 'subsamp
```

```
le': 0.938049283306656}}

Time taken to run this cell : 13:08:32.954992
```

11.5.2 Concatenating Train and Test Datasets and Filtering Data by Taking only Top 500 Features

In [17]:
```
if not os.path.isfile('pickles/full_data'):
    full_data = pd.concat([train_data,test_data])
    full_data.to_pickle('pickles/full_data')
full_data = pd.read_pickle('pickles/full_data')

print(full_data.shape)
```

```
(356255, 2981)
```

In [18]:
```
full_data_x = full_data.drop(['TARGET','SK_ID_CURR'], axis=1)
full_data_top_df = full_data_x.iloc[:,cols]

full_data_top_df = reduce_memory_usage(full_data_top_df)

print('Initial shape of Full Data after Selecting Top 500 Features: {}'
.format(full_data_top_df.shape))
full_data_top_df['TARGET'] = full_data['TARGET']
full_data_top_df['SK_ID_CURR'] = full_data['SK_ID_CURR']
print('Final shape of Train Data: {}'.format(full_data_top_df.shape))
```

```
Memory usage of dataframe is 996.83 MB
Memory usage after optimization is: 452.21 MB
Decreased by 54.6%
Initial shape of Full Data after Selecting Top 500 Features: (356255, 5
00)
Final shape of Train Data: (356255, 502)
```

In [19]:
```
train_df = full_data_top_df[full_data_top_df['TARGET'].notnull()]
test_df = full_data_top_df[full_data_top_df['TARGET'].isnull()]
```

```
print(train_df.shape)
print(test_df.shape)
```

```
(307511, 502)
(48744, 502)
```

11.5.3 Carrying out 5-Fold CV

In [20]:
```
f =  KFold(n_splits=5,shuffle=True,random_state=0) #K fold cross valida
tion
lgbm_df = pd.DataFrame()

train_df = train_df.rename(columns = lambda x:re.sub('[^A-Za-z0-9_]+',
'', x))
test_df = test_df.rename(columns = lambda x:re.sub('[^A-Za-z0-9_]+', ''
, x))
y = train_df['TARGET']

feats = [f for f in train_df.columns if f not in ['TARGET','SK_ID_CURR'
,'SK_ID_BUREAU',\
                                                  'SK_ID_PREV','index'
]]
lgbm_df['feat']=feats

#hold outputs of training, testing and cv data
train_predict = np.zeros(train_df.shape[0])
cv_predict = np.zeros(train_df.shape[0])
test_predict = np.zeros(test_df.shape[0])

best_threshold_train = np.zeros(X_train_final_hcdr_new.shape[0])
```

11.5.4 Obtaining the CV and Test Data Predictions, and the Ideal Threshold Value

In [15]:
```
start = datetime.now()
```

```
a=0
for i,(train, cv) in enumerate(f.split(train_df[feats],y)):
    X_train, Y_train = train_df[feats].iloc[train], y.iloc[train]
    X_valid, Y_valid = train_df[feats].iloc[cv], y.iloc[cv]

    lgb = LGBMClassifier(
            n_estimators=10837, \
            bagging_fraction= 0.7327318230470493, \
            lambda_l1=4.095528031147774,  \
            lambda_l2=1.6872110950694539, \
            learning_rate=0.009023114465955174, \
            min_child_samples=50, \
            num_leaves=35, \
            feature_fraction=0.1723525427151981, \
            max_depth=20, \
            min_split_gain=0.19202267895386566, \
            min_child_weight=52,
            scale_pos_weight=6.014158781875895, \
            subsample=0.938049283306656, \
            n_jobs=2, \
            verbose=-1)

    lgb.fit(X_train, Y_train, eval_set=[(X_train, Y_train), (X_valid, Y
_valid)], \
            eval_metric= 'auc', verbose=-1, early_stopping_rounds=11000
)
    a=a+1

    train_predict[train] = lgb.predict_proba(X_train, num_iteration=lgb
.best_iteration_)[:, 1]
    cv_predict[cv]=lgb.predict_proba(X_valid, num_iteration=lgb.best_it
eration_)[:, 1]

    fpr_t, tpr_t, thresh = roc_curve(Y_train, train_predict[train])
    best_stat = tpr_t - fpr_t
    best_thresh_index = np.argmax(best_stat)
    best_threshold_train += thresh[best_thresh_index]/5

    test_predict += lgb.predict_proba(test_df[feats], num_iteration=lgb
```

```
.best_iteration_)[:, 1] / 5
    with open('lgbm/lgbm_model_500f_'+ str(a) +'.pickle', 'wb') as hand
le:
        pickle.dump(lgb, handle)

    lgbm_df["imp"] = lgb.feature_importances_
    lgbm_df["fold"] = i + 1

    print(" "*100)
    print('Fold ',i + 1,' Train AUC : ',roc_auc_score(Y_train, train_pr
edict[train]))
    print('Fold ',i + 1,' CV AUC : ',roc_auc_score(Y_valid, cv_predict[
cv]))
    print("="*80)
    print(" "*100)


print('Full Train AUC score ',roc_auc_score(y, train_predict))
print('Full CV AUC score ',roc_auc_score(y, cv_predict))

with open('lgbm/lgbm_train_predict_500f.pkl','wb') as f:
     pickle.dump(train_predict, f)

with open('lgbm/lgbm_cv_predict_500f.pkl','wb') as f:
    pickle.dump(cv_predict, f)

with open('lgbm/lgbm_test_predict_500f.pkl','wb') as f:
    pickle.dump(test_predict, f)

with open('lgbm/lgbm_best_threshold_500f_api.pkl','wb') as f:
    pickle.dump(best_threshold_train, f)

print(" "*80)
print("Time taken to run this cell :", datetime.now() - start)
```

```
Training until validation scores don't improve for 11000 rounds
Did not meet early stopping. Best iteration is:
[10837] training's auc: 0.962937        training's binary_logloss: 0.25
8202    valid_1's auc: 0.780891 valid_1's binary_logloss: 0.328496
```

```
Fold  1  Train AUC :  0.9629365803011315
Fold  1  CV AUC :  0.7808910494438999
=========================================================================
=========

[LightGBM] [Warning] lambda_l1 is set=4.095528031147774, reg_alpha=0.0
will be ignored. Current value: lambda_l1=4.095528031147774
[LightGBM] [Warning] feature_fraction is set=0.1723525427151981, colsam
ple_bytree=1.0 will be ignored. Current value: feature_fraction=0.17235
25427151981
[LightGBM] [Warning] bagging_fraction is set=0.7327318230470493, subsam
ple=0.938049283306656 will be ignored. Current value: bagging_fraction=
0.7327318230470493
[LightGBM] [Warning] lambda_l2 is set=1.6872110950694539, reg_lambda=0.
0 will be ignored. Current value: lambda_l2=1.6872110950694539
Training until validation scores don't improve for 11000 rounds
Did not meet early stopping. Best iteration is:
[10837] training's auc: 0.96291 training's binary_logloss: 0.256995
valid_1's auc: 0.776727 valid_1's binary_logloss: 0.328677

Fold  2  Train AUC :  0.9629103510164205
Fold  2  CV AUC :  0.7767265085870596
=========================================================================
=========

[LightGBM] [Warning] lambda_l1 is set=4.095528031147774, reg_alpha=0.0
will be ignored. Current value: lambda_l1=4.095528031147774
[LightGBM] [Warning] feature_fraction is set=0.1723525427151981, colsam
ple_bytree=1.0 will be ignored. Current value: feature_fraction=0.17235
25427151981
[LightGBM] [Warning] bagging_fraction is set=0.7327318230470493, subsam
ple=0.938049283306656 will be ignored. Current value: bagging_fraction=
0.7327318230470493
[LightGBM] [Warning] lambda_l2 is set=1.6872110950694539, reg_lambda=0.
0 will be ignored. Current value: lambda_l2=1.6872110950694539
Training until validation scores don't improve for 11000 rounds
Did not meet early stopping. Best iteration is:
[10837] training's auc: 0.96339 training's binary_logloss: 0.256113
valid_1's auc: 0.775115 valid_1's binary_logloss: 0.329178
```

```
Fold  3  Train AUC :  0.9633902099194458
Fold  3  CV AUC :  0.7751152852158589
=========================================================================
=========

[LightGBM] [Warning] lambda_l1 is set=4.095528031147774, reg_alpha=0.0
will be ignored. Current value: lambda_l1=4.095528031147774
[LightGBM] [Warning] feature_fraction is set=0.1723525427151981, colsam
ple_bytree=1.0 will be ignored. Current value: feature_fraction=0.17235
25427151981
[LightGBM] [Warning] bagging_fraction is set=0.7327318230470493, subsam
ple=0.938049283306656 will be ignored. Current value: bagging_fraction=
0.7327318230470493
[LightGBM] [Warning] lambda_l2 is set=1.6872110950694539, reg_lambda=0.
0 will be ignored. Current value: lambda_l2=1.6872110950694539
Training until validation scores don't improve for 11000 rounds
Did not meet early stopping. Best iteration is:
[10837] training's auc: 0.963571        training's binary_logloss: 0.25
5598    valid_1's auc: 0.772978 valid_1's binary_logloss: 0.331585

Fold  4  Train AUC :  0.9635708593364005
Fold  4  CV AUC :  0.7729780322733407
=========================================================================
=========

[LightGBM] [Warning] lambda_l1 is set=4.095528031147774, reg_alpha=0.0
will be ignored. Current value: lambda_l1=4.095528031147774
[LightGBM] [Warning] feature_fraction is set=0.1723525427151981, colsam
ple_bytree=1.0 will be ignored. Current value: feature_fraction=0.17235
25427151981
[LightGBM] [Warning] bagging_fraction is set=0.7327318230470493, subsam
ple=0.938049283306656 will be ignored. Current value: bagging_fraction=
0.7327318230470493
[LightGBM] [Warning] lambda_l2 is set=1.6872110950694539, reg_lambda=0.
0 will be ignored. Current value: lambda_l2=1.6872110950694539
Training until validation scores don't improve for 11000 rounds
Did not meet early stopping. Best iteration is:
[10837] training's auc: 0.963701        training's binary_logloss: 0.25
5398    valid_1's auc: 0.776575 valid_1's binary_logloss: 0.329185
```

```
Fold  5  Train AUC :  0.9637010568618634
Fold  5  CV AUC :  0.7765749681571059
================================================================================
=========

Full Train AUC score  0.9637142494650246
Full CV AUC score  0.776405555086621

Time taken to run this cell : 4:07:06.749940
```

11.5.5 Obtaining Final Data in Kaggle Competition Submission Format

In [21]:
```
with open('lgbm/lgbm_train_predict_500f.pkl','rb') as f:
    train_predict = pickle.load(f)

with open('lgbm/lgbm_cv_predict_500f.pkl','rb') as f:
    cv_predict = pickle.load(f)

with open('lgbm/lgbm_test_predict_500f.pkl','rb') as f:
    test_predict = pickle.load(f)

with open('lgbm/lgbm_test_predict_500f.pkl','rb') as f:
    test_predict = pickle.load(f)

with open('lgbm/lgbm_best_threshold_500f_api.pkl','rb') as f:
    threshold = pickle.load(f)
```

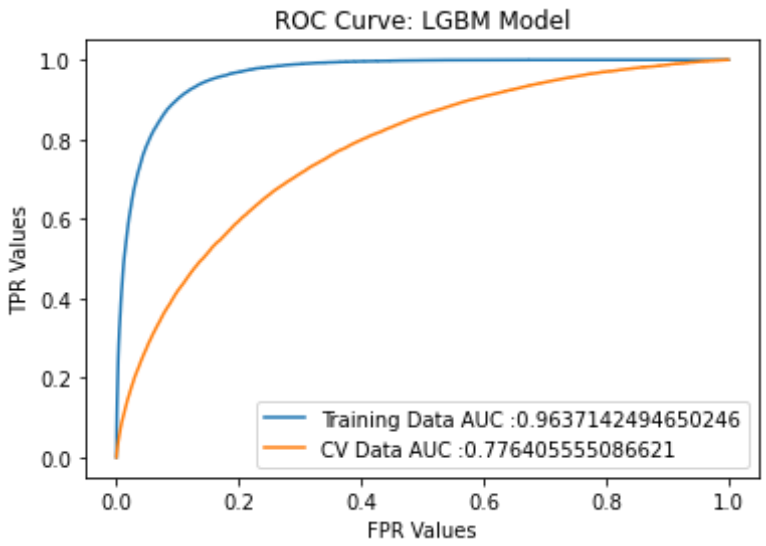
In [18]:
```
features_top_df_test['SK_ID_CURR'] = test_data['SK_ID_CURR']
features_top_df_test['TARGET'] = test_predict
features_top_df_test['SK_ID_CURR'] = features_top_df_test['SK_ID_CURR']
.apply(lambda x: np.int32(x))
features_top_df_test[['SK_ID_CURR', 'TARGET']].to_csv('hcdr_lgbm_500f_f
inal.csv', index= False)
```

Observations :-

- We can see from above that the AUC Value obtained on the Test Data using the LGBM Classifier in the Kaggle Competition = 0.78101.

11.5.6 Obtaining ROC Curves on Train and Test Datasets

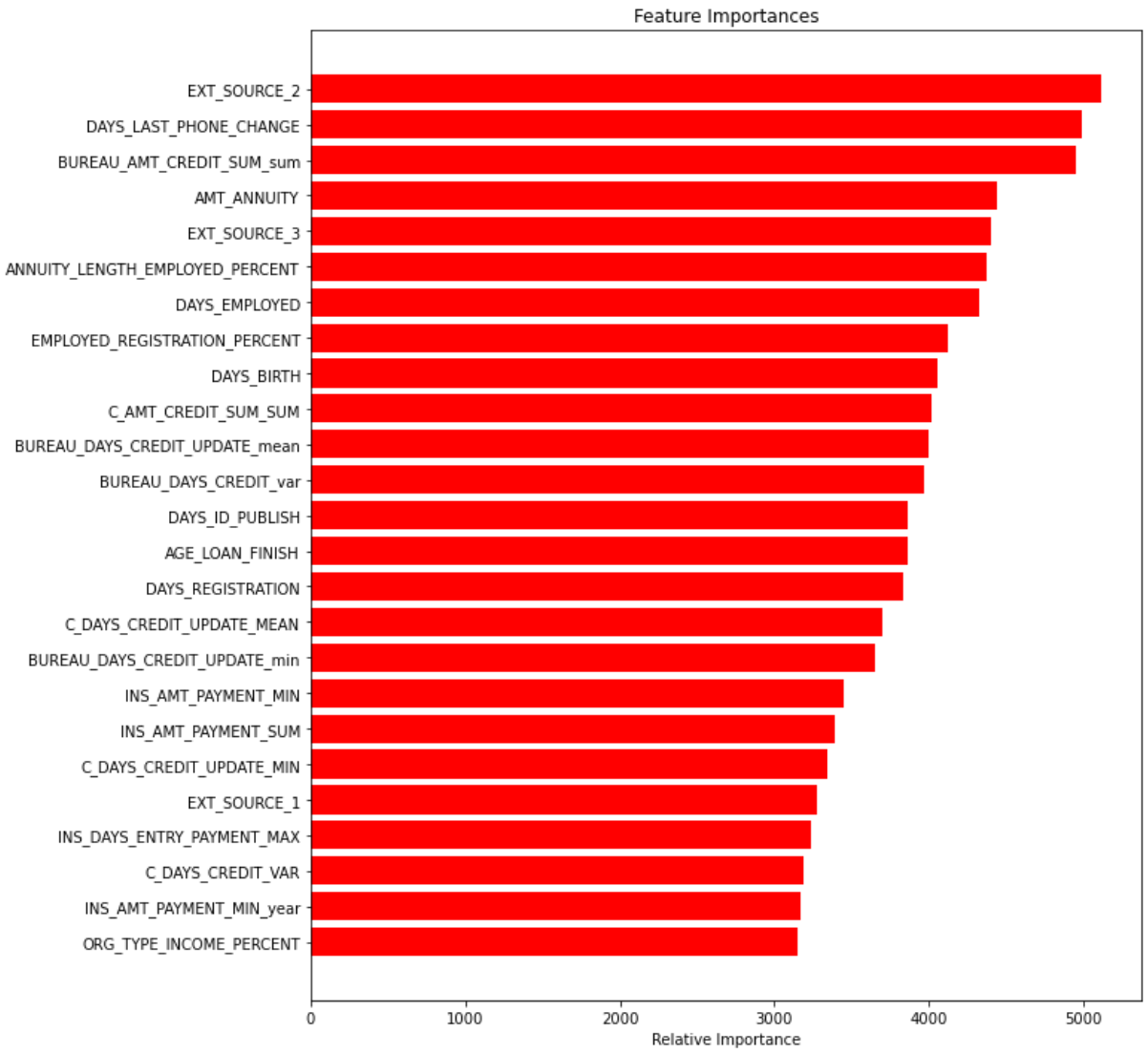
In [22]:
```
start = datetime.now()

train_fpr5, train_tpr5, tr_thresholds5 = roc_curve(y, train_predict)
cv_fpr5,cv_tpr5, cv_thresholds5 = roc_curve(y, cv_predict)

plt.plot(train_fpr5,train_tpr5, label ="Training Data AUC :" + str(auc(
train_fpr5,train_tpr5)))
plt.plot(cv_fpr5,cv_tpr5,label="CV Data AUC :" + str(auc(cv_fpr5,cv_tpr
5)))
plt.legend()

plt.xlabel("FPR Values")
plt.ylabel("TPR Values")
plt.title('ROC Curve: LGBM Model')

plt.grid(False)
plt.show()

print('Ideal Threshold for the CV Dataset =', set(threshold))
print("Time taken to run this cell :", datetime.now() - start)
```




```
Ideal Threshold for the CV Dataset = {0.3741018248484985}
Time taken to run this cell : 0:00:00.708510
```

11.5.7 Top 25 Features obtained from Light GBM Model

In [20]:
```
#plot imp features
features = lgbm_df["feat"]
importances = lgbm_df["imp"]
indices = (np.argsort(importances))[-25:]
plt.figure(figsize=(10,12))
plt.title('Feature Importances')
plt.barh(range(len(indices)), importances[indices], color='red', align=
'center')
plt.yticks(range(len(indices)), [features[i] for i in indices])
plt.xlabel('Relative Importance')
plt.show()
```



Observations :-

- According to the Feature Importances obtained from the Light GBM Classifier after Hyperparameter Tuning, 'EXT_SOURCE_2', 'DAYS_LAST_PHONE_CHANGE' and

'BUREAU_AMT_CREDIT_SUM_sum' are the most important features in deciding the Loan repayment capability of a borrower.

12. Conclusion

12.1. Summary

The problem is that there are a lot of people who apply for loans in Banks and similar financial institutions whereas only a few of them get approved. This is primarily because of insufficient or non-existent credit histories of the applicant, whereas this population is taken advantage of by untrustworthy lenders. In order to make sure that these applicants have a positive loan taking experience, Home Credit uses a lot of data (including telco data and transactional data) to predict the applicants' loan repayment abilities. Improving on this overall process basically ensures that the clients capable of loan repayment do not have their applications rejected.

In order to work with this problem, we first carry out extensive EDA on each of the data tables provided in order to understand our data better and in order to fix the null values as well as outliers present in our data. This is followed by extensive Feature Engineering and join the data tables and at the end of all this, we get a total of 2948 features, which obviously is a very high number to work with. So we take only the Top 500 features filtered using 'SelectKBest' and develop our models, summary of which is as follows, where we can see that the best model obtained is LGBM with Hyperparameter Tuning using Bayesian Optimization after 5-fold Cross Validation.

In [23]:
```
x=PrettyTable()
x.field_names=["Model","Test AUC"]

x.add_row(["Logistic Regression", "0.71840"])
x.add_row(["Linear SVM", "0.72262"])
x.add_row(["Random Forest Classifier","0.73097"])
x.add_row(["XGBoost","0.73805"])
x.add_row(["Light GBM","0.78101"])
```

```
print(x)
```

```
+--------------------------+----------+
|          Model           | Test AUC |
+--------------------------+----------+
|   Logistic Regression    | 0.71840  |
|        Linear SVM        | 0.72262  |
| Random Forest Classifier | 0.73097  |
|         XGBoost          | 0.73805  |
|        Light GBM         | 0.78101  |
+--------------------------+----------+
```