Home Credit Default Risk Part 2: Data Cleaning and Feature Engineering

In this section, we will use the knownledge from exploratory data analysis to do data cleaninig and feature engineering.

Table of contents

Loading libries

In [45]:

```
#import usefull libries
import pandas as pd
import numpy as np
from scipy.stats import uniform

#import plotting libraries
import matplotlib.pyplot as plt
import seaborn as sns
from prettytable import PrettyTable

#import Misc Libraries
import os
import gc
import pickle
import warnings
warnings.filterwarnings('ignore')
from datetime import datetime

#for 100% jupyter notebook cell width
from IPython.core.display import display, HTML
display(HTML("<style>.container { width: 100% !important; }</style>"))
```

```
#sklearn
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.model_selection import StratifiedKFold
from sklearn.model_selection import RandomizedSearchCV
from sklearn.metrics import roc_auc_score
from sklearn.metrics import precision_score
from sklearn.metrics import recall_score
from sklearn.metrics import roc_curve
from sklearn.metrics import confusion_matrix
from sklearn.neighbors import KNeighborsClassifier
from sklearn.calibration import CalibratedClassifierCV

from xgboost import XGBRegressor
from lightgbm import LGBMRegressor
from lightgbm import LGBMClassifier
```

1. Defining Utility Functions and Classes

- `reduce_mem_usage` :

This function is used to iterating through all the columns of a dataframe and modify the data type to reduce memory usage.

- `relational_tables_prepare` :

Function to pickle the relational tables which would need to be merged during production with the test datapoint

In [9]:

```
print(np.iinfo(np.int8))
print(np.iinfo(np.int16))
print(np.finfo(np.float16))
```

```
Machine parameters for int8
---------------------------------------------------------------
min = -128
max = 127
---------------------------------------------------------------

Machine parameters for int16
---------------------------------------------------------------
min = -32768
max = 32767
---------------------------------------------------------------

Machine parameters for float16
---------------------------------------------------------------
precision =   3   resolution = 1.00040e-03
machep =    -10   eps =        9.76562e-04
negep =     -11   epsneg =     4.88281e-04
minexp =    -14   tiny =       6.10352e-05
maxexp =     16   max =        6.55040e+04
nexp =        5   min =        -max
```

```
----------------------------------------------------------------
```

In [10]:

```
def reduce_mem_usage(data, verbose=True):
    #reference: https://www.kaggle.com/gemartin/load-data-reduce-memory-usage
    '''
    This function is used to iterating through all the columns of a dataframe and modif
    the data type to reduce memory usage.
    '''

    start_mem = data.memory_usage().sum() / 1024**2
    if verbose:
        print('-' * 100)
        print(f'Memory usage of dataframe is {start_mem} MB')

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

        if col_type != object:
            c_min = data[col].min()
            c_max = data[col].max()
            if str(col_type) == 'int':
                if c_min > np.iinfo(np.int8).min and c_max < np.iinfo(np.int8).max:
                    data[col] = data[col].astype(np.int8)
                elif c_min > np.iinfo(np.int16).min and c_max < np.iinfo(np.int16).max:
                    data[col] = data[col].astype(np.int16)
                elif c_min > np.iinfo(np.int32).min and c_max < np.iinfo(np.int32).max:
                    data[col] = data[col].astype(np.int32)
                elif c_min > np.iinfo(np.int64).min and c_max < np.iinfo(np.int64).max:
                    data[col] = data[col].astype(np.int64)
            else:
                if c_min > np.finfo(np.float16).min and c_max < np.finfo(np.float16).ma
                    data[col] = data[col].astype(np.float16)
                elif c_min > np.finfo(np.float32).min and c_max < np.finfo(np.float32).
                    data[col] = data[col].astype(np.float32)
                else:
                    data[col] = data[col].astype(np.float64)

    end_mem = data.memory_usage().sum() / 1024**2
    if verbose:
        print(f'Memory usage of after optimization is {end_mem} MB')
        print(f'Decreased by {100 * (start_mem - end_mem) / start_mem}')
        print('-' * 100)

    return data
```

In [13]:

```
def relational_tables_prepare(file_directory='', verbose=True):
    '''
    Function to pickle the relational tables which would need to be merged during
    production with the test datapoint

    Inputs:
        file_directory: str, default = ''
            The directory in which files are saved
        verbose: bool, default = True
            Whether to keep verbosity or not

    Returns:
```

```
    None
'''

if verbose:
    print('Loading the tables into memory...')
    start = datetime.now()

#loading all the tables in memory, for dimensionality reduction
with open(file_directory + 'bureau_merged_preprocessed.pkl', 'rb') as f:
    bureau_aggregated = reduce_mem_usage(pickle.load(f), verbose = False)
with open(file_directory + 'previous_application_preprocessed.pkl', 'rb') as f:
    previous_aggregated = reduce_mem_usage(pickle.load(f), verbose = False)
with open(file_directory + 'installments_payments_preprocessed.pkl', 'rb') as f:
    installments_aggregated = reduce_mem_usage(pickle.load(f), verbose = False)
with open(file_directory + 'POS_CASH_balance_preprocessed.pkl', 'rb') as f:
    pos_aggregated = reduce_mem_usage(pickle.load(f), verbose = False)
with open(file_directory + 'credit_card_balance_preprocessed.pkl', 'rb') as f:
    cc_aggregated = reduce_mem_usage(pickle.load(f), verbose = False)
with open(file_directory + 'application_train_preprocessed.pkl', 'rb') as f:
    application_train = reduce_mem_usage(pickle.load(f), verbose = False)
with open(file_directory + 'application_test_preprocessed.pkl', 'rb') as f:
    application_test = reduce_mem_usage(pickle.load(f), verbose = False)
with open('Final_XGBOOST_Selected_features.pkl', 'rb') as f:
    final_cols = pickle.load(f)

if verbose:
    print('Done')
    printnt(f'Time elapsed {datetime.now() - start}')
    start2 = datetime.now()
    print('\nRemoving the non-useful features...')

#removing non-useful columns from pre-processed previous_application table
previous_app_columns_to_keep = set(previous_aggregated.columns).intersection(set(fi
                                set([ele for ele in previous_aggregated.columns if
                                   [ele for ele in previous_aggregated.columns if '
previous_aggregated = previous_aggregated[previous_app_columns_to_keep]

#removing non-useful columns from pre-processed credit_card_balance table
credit_card_balance_columns_to_keep = set(cc_aggregated.columns).intersection(set(f
                                        set([ele for ele in cc_aggregated.columns i
                                            [ele for ele in cc_aggregated.columns i
                                            [ele for ele in cc_aggregated.columns i
cc_aggregated = cc_aggregated[credit_card_balance_columns_to_keep]

#removing non-useful columns from pre-processed installments_payments table
installments_payments_columns_to_keep = set(installments_aggregated.columns).inters
                                        set([ele for ele in installments_aggregated
                                             'RATIO' not in ele and 'DIFF' not in e
                                            ['AMT_INSTALMENT_MEAN_MAX', 'AMT_INSTAL
installments_aggregated = installments_aggregated[installments_payments_columns_to_

#Removing non-useful columns from pre-processed bureau-aggregated table
bureau_columns_to_keep = set(bureau_aggregated.columns).intersection(set(final_cols
                         set([ele for ele in bureau_aggregated.columns if 'DAYS_CRE
                             [ele for ele in bureau_aggregated.columns if 'AMT_CRED
                             [ele for ele in bureau_aggregated.columns if 'AMT_ANNU
bureau_aggregated = bureau_aggregated[bureau_columns_to_keep]

if verbose:
    print('Done.')
```

```
        print(f'Tile elapsed {datetime.now() - start2}')
        print("\nMerging all the tables, and saving to pickle file 'relational_table.pk

    #merging all the tables
    relational_table = cc_aggregated.merge(bureau_aggregated, on = 'SK_ID_CURR', how =
    relational_table = relational_table.merge(previous_aggregated, on = 'SK_ID_CURR', h
    relational_table = relational_table.merge(installments_aggregated, on = 'SK_ID_CURR
    relational_table = relational_table.merge(pos_aggregated, on = 'SK_ID_CURR', how =
    relational_table = reduce_mem_usage(relational_table, verbose = False)

    with open(file_directory + 'relational_table.pkl', 'wb') as f:
        pickle.dump(relational_table, f)

    if verbose:
        print("Done.")
        print(f"Total Time taken = {datetime.now() - start}")
```

2. Data Clearning and Feature Engineering

The data contains several number of relational tables. We'll process each one of them separately, and then finally in the end, merge all of them together.

2.1 Preprocessing Tables

2.1.1 bureau_balance.csv and bureau.csv

These tables contain the information related to the client's previous credits which were not with Home Credit Group, and were reported by Credit Bureau Department.

bureau_balance

1. First of all, the bureau_balance table contains three fields, i.e. SK_ID_BUREAU, MONTHS_BALANCE and STATUS.
2. Since the Status follows somewhat ordinal behaviour, we start by label encoding it.
3. Next, some features are created such as weighted status, which is obtained by dividing the status by the MONTHS_BALANCE.
4. Since the data contains the timeseries, we also calculate the Exponential Weighted Moving Average of the Status and Weighted Status fields.
5. Finally, we aggregate the data over SK_ID_BUREAU, in such a way that we first aggregate it over all the data, and after that we also aggregate over the last 2 years. These 2 years would depict the more recent behaviour of the clients.
6. The aggregations performed are based on Domain Knowledge, such as mean, min, max, sum, count, etc. For EDA features, we only take the last/most recent values, as they somewhat contain the trend of all the previous values.

bureau

1. Firstly, we merge the bureau table with the aggregated bureau_balance table from previous step, on SK_ID_BUREAU.
2. We replace some erroneous values with NaN values. We saw some loans dating back to as long as 100 years ago. We believe they wouldn't really tell much about client's recent behaviour, so we remove them and only keep the loans in the period of 50 years.
3. We create some features by multiplications, divisions, subtractions of raw features, based on domain knowledge, such as Credit duration, annutiy to credit ratio, etc.
4. The categorical features are one-hot encoded.
5. To merge these to main table, i.e. application_train, we aggregate this table over SK_ID_CURR. We perform the aggregations again in two ways. We aggregate the credits based on the CREDIT_ACTIVE category, where we aggregate for two most popular categories separately, i.e. Active, and Closed. Later we aggregate for the remaining categories too, and merge these. We aggregated the whole data overall too. The aggregations performed are sum, mean, min, max, last, etc.

In [17]:
```
class preprocess_bureau_balance_and_bureau:
    '''
    Preprocess the tables bureau_balance and bureau.
    Contains 4 member functions:
        1. init method
        2. preprocess_bureau_balance method
        3. preprocess_bureau method
        4. main method
    '''
    def __init__(self, file_directory='', verbose=True, dump_to_pickle=False):
        '''
        This function is used to initialize the class members

        Inputs:
            self
            file_directory: Path, str, default = ''
                The path where the file exists. Include a '/' at the end of the path in
            verbose: bool, default = True
                Whether to enable verbosity or not
            dump_to_pickle: bool, default = False
                Whether to pickle the final preprocessed table or not

        Returns:
            None
        '''
        self.file_directory = file_directory
        self.verbose = verbose
        self.dump_to_pickle = dump_to_pickle
        self.start = datetime.now()

    def preprocess_bureau_balance(self):
        '''
        Function to preprocess bureau_balance table.
        This function first loads the table into memory, does some feature engineering,
        and finally aggregates the data over SK_ID_BUREAU

        Inputs:
```

```
            self

        Returns:
            preprocessed and aggregated bureau_balance table.
        '''
        if self.verbose:
            print('#######################################################')
            print('#          Pre-processing bureau_balance.csv          #')
            print('#######################################################')
            print("\nLoading the DataFrame, bureau_balance.csv, into memory...")

        bureau_balance = pd.read_csv(self.file_directory + 'bureau_balance.csv')

        if self.verbose:
            print("Loaded bureau_balance.csv")
            print(f"Time Taken to load = {datetime.now() - self.start}")
            print("\nStarting Data Cleaning...")

        #as we saw from EDA, bureau_balance has a variable called STATUS, which describ
        #about the status of loan.
        #it has 7 labels, we will label encode them
        #so we give C as 0, and rest increasing
        #also we will give X the benefit of doubt and keep it as middle value
        dict_for_status = { 'C': 0, '0': 1, '1': 2, '2': 3, 'X': 4, '3': 5, '4': 6, '5'
        bureau_balance['STATUS'] = bureau_balance['STATUS'].map(dict_for_status)

        #weighting the status with the months_balance
        #converting month to positive
        bureau_balance['MONTHS_BALANCE'] = np.abs(bureau_balance['MONTHS_BALANCE'])
        bureau_balance['WEIGHTED_STATUS'] = bureau_balance.STATUS / (bureau_balance.MON

        #sorting the bureau_balance in ascending order of month and by the bureau_sk_ID
        #this is done as to make the rolling exponential average easily for previous mo
        bureau_balance = bureau_balance.sort_values(by=['SK_ID_BUREAU', 'MONTHS_BALANCE
        #we will do exponential weighted average on the encoded status
        #this is because if a person had a bad status 2 years ago, it should be given l
        # we keep the latent variable alpha = 0.8
        #doing this for both weighted status and the status itself
        bureau_balance['EXP_WEIGHTED_STATUS'] = \
                      bureau_balance.groupby('SK_ID_BUREAU')['WEIGHTED_STATUS'].transfo
        bureau_balance['EXP_ENCODED_STATUS'] = \
                      bureau_balance.groupby('SK_ID_BUREAU')['STATUS'].transform(lambda

        if self.verbose:
            print("Halfway through. A little bit more patience...")
            print(f"Total Time Elapsed = {datetime.now() - self.start}")

        #we can see that these datapoints are for 96 months i.e. 8 years.
        #so we will extract the means, and exponential averages for each year separatel
        #first we convert month to year
        bureau_balance['MONTHS_BALANCE'] = bureau_balance['MONTHS_BALANCE'] // 12

        #defining our aggregations
        aggregations_basic = {
            'MONTHS_BALANCE': ['mean', 'max'],
            'STATUS': ['mean', 'max', 'first'],
            'WEIGHTED_STATUS': ['mean', 'sum', 'first'],
            'EXP_ENCODED_STATUS': ['last'],
            'EXP_WEIGHTED_STATUS': ['last']
        }
```

```
        #we will find aggregates for each year too
        aggregations_for_year = {
            'STATUS': ['mean', 'max', 'last', 'first'],
            'WEIGHTED_STATUS': ['mean', 'max', 'first', 'last'],
            'EXP_ENCODED_STATUS': ['last'],
            'EXP_WEIGHTED_STATUS': ['last']
        }
        #aggregating over whole dataset first
        aggregated_bureau_balance = bureau_balance.groupby('SK_ID_BUREAU').agg(aggregat
        aggregated_bureau_balance.columns = ['_'.join(ele).upper() for ele in aggregate

        #aggregating some of the features separately for latest 2 years
        aggregated_bureau_years = pd.DataFrame()
        for year in range(2):
            year_group = bureau_balance[bureau_balance['MONTHS_BALANCE'] == year].group
            year_group.columns = ['_'.join(ele).upper() + '_YEAR_' + str(year) for ele

            if year == 0:
                aggregated_bureau_years = year_group
            else:
                aggregated_bureau_years = aggregated_bureau_years.merge(year_group, on=

        #aggregating for rest of years
        aggregated_bureau_rest_years = bureau_balance[bureau_balance.MONTHS_BALANCE > y
        aggregated_bureau_rest_years.columns = ['_'.join(ele).upper() + '_YEAR_REST' fo

        #merging with rest of years
        aggregated_bureau_years = aggregated_bureau_years.merge(aggregated_bureau_rest_
        aggregated_bureau_balance = aggregated_bureau_balance.merge(aggregated_bureau_y

        #filling the missing values obtained after greggations with 0
        aggregated_bureau_balance.fillna(0, inplace=True)

        if self.verbose:
            print('Done preprocessing bureau_balance.')
            print(f"\nInitial Size of bureau_balance: {bureau_balance.shape}")
            print(f'Size of bureau_balance after Pre-Processing, Feature Engineering an
            print(f'\nTotal Time Taken = {datetime.now() - self.start}')

        if self.dump_to_pickle:
            if self.verbose:
                print('\nPickling pre-processed bureau_balance to bureau_balance_prepro
            with open(self.file_directory + 'bureau_balance_preprocessed.pkl', 'wb') as
                pickle.dump(aggregated_bureau_balance, f)
            if self.verbose:
                print('Done.')

        return aggregated_bureau_balance

    def preprocess_bureau(self, aggregated_bureau_balance):
        '''
        Function to preprocess the bureau table and merge it with the aggregated bureau
        Finally aggregates the data over SK_ID_CURR for it to be merged with applicatio

        Inputs:
            self
            aggregated_bureau_balance: DataFrame of aggregated bureau_balance table

        Returns:
            Final preprocessed, merged and aggregated bureau table
```

```
        '''
        if self.verbose:
            start2 = datetime.now()
            print('\n##############################################')
            print('#          Pre-processing bureau.csv         #')
            print('##############################################')
            print("\nLoading the DataFrame, bureau.csv, into memory...")

        bureau = pd.read_csv(self.file_directory + 'bureau.csv')

        if self.verbose:
            print("Loaded bureau.csv")
            print(f"Time Taken to load = {datetime.now() - start2}")
            print("\nStarting Data Cleaning and Feature Engineering...")

        #merging it with aggregated bureau_balance on 'SK_ID_BUREAU'
        bureau_merged = bureau.merge(aggregated_bureau_balance, on='SK_ID_BUREAU', how=

        #from the EDA we saw some erroneous values in DAYS Fields, we will remove those
        #there are some loans which ended about very long ago, around 100 years ago.
        #Thus we will only keep those loans which have ended in past 50 years.
        bureau_merged['DAYS_CREDIT_ENDDATE'][bureau_merged['DAYS_CREDIT_ENDDATE'] > -50
        bureau_merged['DAYS_ENDDATE_FACT'][bureau_merged['DAYS_ENDDATE_FACT'] > -50*365
        #there is also a feature which tells about the number of days ago the credit re
        bureau_merged['DAYS_CREDIT_UPDATE'][bureau_merged['DAYS_CREDIT_UPDATE'] > -50*3

        #engineering some features based on domain knowledge
        bureau_merged['CREDIT_DURATION'] = np.abs(bureau_merged['DAYS_CREDIT'] - bureau
        bureau_merged['FLAG_OVERDUE_RECENT'] = [0 if ele == 0 else 1 for ele in bureau_
        bureau_merged['MAX_AMT_OVERDUE_DURATION_RATIO'] = bureau_merged['AMT_CREDIT_MAX
        bureau_merged['CURRENT_AMT_OVERDUE_DURATION_RATIO'] = bureau_merged['AMT_CREDIT

        bureau_merged['AMT_OVERDUE_DURATION_LEFT_RATIO'] = bureau_merged['AMT_CREDIT_SU
        bureau_merged['CNT_PROLONGED_MAX_OVERDUE_MUL'] = bureau_merged['CNT_CREDIT_PROL
        bureau_merged['CNT_PROLONGED_DURATION_RATIO'] = bureau_merged['CNT_CREDIT_PROLO
        bureau_merged['CURRENT_DEBT_TO_CREDIT_RATIO'] = bureau_merged['AMT_CREDIT_SUM_D
        bureau_merged['CURRENT_CREDIT_DEBT_DIFF'] = bureau_merged['AMT_CREDIT_SUM'] - b
        bureau_merged['AMT_ANNUITY_CREDIT_RATIO'] = bureau_merged['AMT_ANNUITY'] / (bur
        bureau_merged['CREDIT_ENDDATE_UPDATE_DIFF'] = bureau_merged['DAYS_CREDIT_UPDATE

        #now we will be aggregating the bureau_merged df with respect to 'SK_ID_CURR' s
        #application_train later
        #firstly we will aggregate the columns based on the category of CREDIT_ACTIVE
        aggregations_CREDIT_ACTIVE = {
            'DAYS_CREDIT': ['mean', 'min', 'max', 'last'],
            'CREDIT_DAY_OVERDUE': ['mean', 'max'],
            'DAYS_CREDIT_ENDDATE': ['mean', 'max'],
            'DAYS_ENDDATE_FACT': ['mean', 'min'],
            'AMT_CREDIT_MAX_OVERDUE': ['max', 'sum'],
            'CNT_CREDIT_PROLONG': ['max', 'sum'],
            'AMT_CREDIT_SUM': ['sum', 'max'],
            'AMT_CREDIT_SUM_DEBT': ['sum'],
            'AMT_CREDIT_SUM_LIMIT': ['max', 'sum'],
            'AMT_CREDIT_SUM_OVERDUE': ['max', 'sum'],
            'DAYS_CREDIT_UPDATE': ['mean', 'min'],
            'AMT_ANNUITY': ['mean', 'sum', 'max'],
            'CREDIT_DURATION': ['max', 'mean'],
            'FLAG_OVERDUE_RECENT': ['sum'],
            'MAX_AMT_OVERDUE_DURATION_RATIO': ['max', 'sum'],
            'CURRENT_AMT_OVERDUE_DURATION_RATIO': ['max', 'sum'],
```

```
            'AMT_OVERDUE_DURATION_LEFT_RATIO': ['max', 'mean'],
            'CNT_PROLONGED_MAX_OVERDUE_MUL': ['mean', 'max'],
            'CNT_PROLONGED_DURATION_RATIO': ['mean', 'max'],
            'CURRENT_DEBT_TO_CREDIT_RATIO': ['mean', 'min'],
            'CURRENT_CREDIT_DEBT_DIFF': ['mean', 'min'],
            'AMT_ANNUITY_CREDIT_RATIO': ['mean', 'max', 'min'],
            'CREDIT_ENDDATE_UPDATE_DIFF': ['max', 'min'],
            'STATUS_MEAN': ['mean', 'max'],
            'WEIGHTED_STATUS_MEAN': ['mean', 'max']
        }
        #we saw from EDA that the two most common type of CREDIT ACTIVE were 'Closed' a
        #So we will aggregate them two separately and the remaining categories separate
        categories_to_aggregate_on = ['Closed','Active']
        bureau_merged_aggregated_credit = pd.DataFrame()
        for i, status in enumerate(categories_to_aggregate_on):
            group = bureau_merged[bureau_merged['CREDIT_ACTIVE'] == status].groupby('SK
            group.columns = ['_'.join(ele).upper() + '_CREDITACTIVE_' + status.upper()

            if i == 0:
                bureau_merged_aggregated_credit = group
            else:
                bureau_merged_aggregated_credit = bureau_merged_aggregated_credit.merge
        #aggregating for remaining categories
        bureau_merged_aggregated_credit_rest = bureau_merged[(bureau_merged['CREDIT_ACT
                                                             (bureau_merged['CREDIT_ACT
                                                            groupby('SK_ID_CURR').agg(a
        bureau_merged_aggregated_credit_rest.columns = ['_'.join(ele).upper() + 'CREDIT
                                                        ele in bureau_merged_aggregated

        #merging with other categories
        bureau_merged_aggregated_credit = bureau_merged_aggregated_credit.merge(bureau_

        #encoding the categorical columns in one-hot form
        currency_ohe = pd.get_dummies(bureau_merged['CREDIT_CURRENCY'], prefix='CURRENC
        credit_active_ohe = pd.get_dummies(bureau_merged['CREDIT_ACTIVE'], prefix='CRED
        credit_type_ohe = pd.get_dummies(bureau_merged['CREDIT_TYPE'], prefix='CREDIT_T

        #merging the one-hot encoded columns
        bureau_merged = pd.concat([bureau_merged.drop(['CREDIT_CURRENCY', 'CREDIT_ACTIV
                                   currency_ohe, credit_active_ohe, credit_type_ohe], a

        #aggregating the bureau_merged over all the columns
        bureau_merged_aggregated = bureau_merged.drop('SK_ID_BUREAU', axis=1).groupby('
        bureau_merged_aggregated.columns = [ele + '_MEAN_OVERALL' for ele in bureau_mer
        #merging it with aggregates over categories
        bureau_merged_aggregated = bureau_merged_aggregated.merge(bureau_merged_aggrega

        if self.verbose:
            print('Done preprocessing bureau and bureau_balance.')
            print(f"\nInitial Size of bureau: {bureau.shape}")
            print(f'Size of bureau and bureau_balance after Merging, Pre-Processing, Fe
            print(f'\nTotal Time Taken = {datetime.now() - self.start}')

        if self.dump_to_pickle:
            if self.verbose:
                print('\nPickling pre-processed bureau and bureau_balance to bureau_mer
            with open(self.file_directory + 'bureau_merged_preprocessed.pkl', 'wb') as
                pickle.dump(bureau_merged_aggregated, f)
            if self.verbose:
                print('Done.')
```

```
        if self.verbose:
            print('-'*100)

        return bureau_merged_aggregated

    def main(self):
        '''
        Function to be called for complete preprocessing and aggregation of the bureau

        Inputs:
            self

        Returns:
            Final pre=processed and merged bureau and burea_balance tables
        '''
        #preprocessing the bureau_balance first
        aggregated_bureau_balance = self.preprocess_bureau_balance()
        #preprocessing the bureau table next, by combining it with the aggregated burea
        bureau_merged_aggregated = self.preprocess_bureau(aggregated_bureau_balance)

        return bureau_merged_aggregated
```

In [19]:

```
# this line take long time for every preprocess__ function, I can split it into two ste
bureau_aggregated = preprocess_bureau_balance_and_bureau(file_directory='./data/',dump_
```

```
#######################################################
#          Pre-processing bureau_balance.csv          #
#######################################################

Loading the DataFrame, bureau_balance.csv, into memory...
Loaded bureau_balance.csv
Time Taken to load = 0:00:03.513000

Starting Data Cleaning...
Halfway through. A little bit more patience...
Total Time Elapsed = 0:04:06.339000
Done preprocessing bureau_balance.

Initial Size of bureau_balance: (27299925, 6)
Size of bureau_balance after Pre-Processing, Feature Engineering and Aggregation: (81739
5, 40)

Total Time Taken = 0:04:13.362999

Pickling pre-processed bureau_balance to bureau_balance_preprocessed.pkl
Done.

##############################################
#          Pre-processing bureau.csv         #
##############################################

Loading the DataFrame, bureau.csv, into memory...
Loaded bureau.csv
Time Taken to load = 0:00:01.643000

Starting Data Cleaning and Feature Engineering...
Done preprocessing bureau and bureau_balance.
```

```
Initial Size of bureau: (1716428, 17)
Size of bureau and bureau_balance after Merging, Pre-Processing, Feature Engineering and
Aggregation: (305811, 242)

Total Time Taken = 0:04:25.299262

Pickling pre-processed bureau and bureau_balance to bureau_merged_preprocessed.pkl
Done.
----------------------------------------------------------------------------------------
------------
```

In [21]:

```
list(bureau_aggregated.columns)
```

Out[21]:

```
['DAYS_CREDIT_MEAN_OVERALL',
 'CREDIT_DAY_OVERDUE_MEAN_OVERALL',
 'DAYS_CREDIT_ENDDATE_MEAN_OVERALL',
 'DAYS_ENDDATE_FACT_MEAN_OVERALL',
 'AMT_CREDIT_MAX_OVERDUE_MEAN_OVERALL',
 'CNT_CREDIT_PROLONG_MEAN_OVERALL',
 'AMT_CREDIT_SUM_MEAN_OVERALL',
 'AMT_CREDIT_SUM_DEBT_MEAN_OVERALL',
 'AMT_CREDIT_SUM_LIMIT_MEAN_OVERALL',
 'AMT_CREDIT_SUM_OVERDUE_MEAN_OVERALL',
 'DAYS_CREDIT_UPDATE_MEAN_OVERALL',
 'AMT_ANNUITY_MEAN_OVERALL',
 'MONTHS_BALANCE_MEAN_MEAN_OVERALL',
 'MONTHS_BALANCE_MAX_MEAN_OVERALL',
 'STATUS_MEAN_MEAN_OVERALL',
 'STATUS_MAX_MEAN_OVERALL',
 'STATUS_FIRST_MEAN_OVERALL',
 'WEIGHTED_STATUS_MEAN_MEAN_OVERALL',
 'WEIGHTED_STATUS_SUM_MEAN_OVERALL',
 'WEIGHTED_STATUS_FIRST_MEAN_OVERALL',
 'EXP_ENCODED_STATUS_LAST_MEAN_OVERALL',
 'EXP_WEIGHTED_STATUS_LAST_MEAN_OVERALL',
 'STATUS_MEAN_YEAR_0_MEAN_OVERALL',
 'STATUS_MAX_YEAR_0_MEAN_OVERALL',
 'STATUS_LAST_YEAR_0_MEAN_OVERALL',
 'STATUS_FIRST_YEAR_0_MEAN_OVERALL',
 'WEIGHTED_STATUS_MEAN_YEAR_0_MEAN_OVERALL',
 'WEIGHTED_STATUS_MAX_YEAR_0_MEAN_OVERALL',
 'WEIGHTED_STATUS_FIRST_YEAR_0_MEAN_OVERALL',
 'WEIGHTED_STATUS_LAST_YEAR_0_MEAN_OVERALL',
 'EXP_ENCODED_STATUS_LAST_YEAR_0_MEAN_OVERALL',
 'EXP_WEIGHTED_STATUS_LAST_YEAR_0_MEAN_OVERALL',
 'STATUS_MEAN_YEAR_1_MEAN_OVERALL',
 'STATUS_MAX_YEAR_1_MEAN_OVERALL',
 'STATUS_LAST_YEAR_1_MEAN_OVERALL',
 'STATUS_FIRST_YEAR_1_MEAN_OVERALL',
 'WEIGHTED_STATUS_MEAN_YEAR_1_MEAN_OVERALL',
 'WEIGHTED_STATUS_MAX_YEAR_1_MEAN_OVERALL',
 'WEIGHTED_STATUS_FIRST_YEAR_1_MEAN_OVERALL',
 'WEIGHTED_STATUS_LAST_YEAR_1_MEAN_OVERALL',
 'EXP_ENCODED_STATUS_LAST_YEAR_1_MEAN_OVERALL',
 'EXP_WEIGHTED_STATUS_LAST_YEAR_1_MEAN_OVERALL',
 'STATUS_MEAN_YEAR_REST_MEAN_OVERALL',
 'STATUS_MAX_YEAR_REST_MEAN_OVERALL',
 'STATUS_LAST_YEAR_REST_MEAN_OVERALL',
 'STATUS_FIRST_YEAR_REST_MEAN_OVERALL',
```

```
'WEIGHTED_STATUS_MEAN_YEAR_REST_MEAN_OVERALL',
'WEIGHTED_STATUS_MAX_YEAR_REST_MEAN_OVERALL',
'WEIGHTED_STATUS_FIRST_YEAR_REST_MEAN_OVERALL',
'WEIGHTED_STATUS_LAST_YEAR_REST_MEAN_OVERALL',
'EXP_ENCODED_STATUS_LAST_YEAR_REST_MEAN_OVERALL',
'EXP_WEIGHTED_STATUS_LAST_YEAR_REST_MEAN_OVERALL',
'CREDIT_DURATION_MEAN_OVERALL',
'FLAG_OVERDUE_RECENT_MEAN_OVERALL',
'MAX_AMT_OVERDUE_DURATION_RATIO_MEAN_OVERALL',
'CURRENT_AMT_OVERDUE_DURATION_RATIO_MEAN_OVERALL',
'AMT_OVERDUE_DURATION_LEFT_RATIO_MEAN_OVERALL',
'CNT_PROLONGED_MAX_OVERDUE_MUL_MEAN_OVERALL',
'CNT_PROLONGED_DURATION_RATIO_MEAN_OVERALL',
'CURRENT_DEBT_TO_CREDIT_RATIO_MEAN_OVERALL',
'CURRENT_CREDIT_DEBT_DIFF_MEAN_OVERALL',
'AMT_ANNUITY_CREDIT_RATIO_MEAN_OVERALL',
'CREDIT_ENDDATE_UPDATE_DIFF_MEAN_OVERALL',
'CURRENCY_currency 1_MEAN_OVERALL',
'CURRENCY_currency 2_MEAN_OVERALL',
'CURRENCY_currency 3_MEAN_OVERALL',
'CURRENCY_currency 4_MEAN_OVERALL',
'CREDIT_ACTIVE_Active_MEAN_OVERALL',
'CREDIT_ACTIVE_Bad debt_MEAN_OVERALL',
'CREDIT_ACTIVE_Closed_MEAN_OVERALL',
'CREDIT_ACTIVE_Sold_MEAN_OVERALL',
'CREDIT_TYPE_Another type of loan_MEAN_OVERALL',
'CREDIT_TYPE_Car loan_MEAN_OVERALL',
'CREDIT_TYPE_Cash loan (non-earmarked)_MEAN_OVERALL',
'CREDIT_TYPE_Consumer credit_MEAN_OVERALL',
'CREDIT_TYPE_Credit card_MEAN_OVERALL',
'CREDIT_TYPE_Interbank credit_MEAN_OVERALL',
'CREDIT_TYPE_Loan for business development_MEAN_OVERALL',
'CREDIT_TYPE_Loan for purchase of shares (margin lending)_MEAN_OVERALL',
'CREDIT_TYPE_Loan for the purchase of equipment_MEAN_OVERALL',
'CREDIT_TYPE_Loan for working capital replenishment_MEAN_OVERALL',
'CREDIT_TYPE_Microloan_MEAN_OVERALL',
'CREDIT_TYPE_Mobile operator loan_MEAN_OVERALL',
'CREDIT_TYPE_Mortgage_MEAN_OVERALL',
'CREDIT_TYPE_Real estate loan_MEAN_OVERALL',
'CREDIT_TYPE_Unknown type of loan_MEAN_OVERALL',
'DAYS_CREDIT_MEAN_CREDITACTIVE_CLOSED',
'DAYS_CREDIT_MIN_CREDITACTIVE_CLOSED',
'DAYS_CREDIT_MAX_CREDITACTIVE_CLOSED',
'DAYS_CREDIT_LAST_CREDITACTIVE_CLOSED',
'CREDIT_DAY_OVERDUE_MEAN_CREDITACTIVE_CLOSED',
'CREDIT_DAY_OVERDUE_MAX_CREDITACTIVE_CLOSED',
'DAYS_CREDIT_ENDDATE_MEAN_CREDITACTIVE_CLOSED',
'DAYS_CREDIT_ENDDATE_MAX_CREDITACTIVE_CLOSED',
'DAYS_ENDDATE_FACT_MEAN_CREDITACTIVE_CLOSED',
'DAYS_ENDDATE_FACT_MIN_CREDITACTIVE_CLOSED',
'AMT_CREDIT_MAX_OVERDUE_MAX_CREDITACTIVE_CLOSED',
'AMT_CREDIT_MAX_OVERDUE_SUM_CREDITACTIVE_CLOSED',
'CNT_CREDIT_PROLONG_MAX_CREDITACTIVE_CLOSED',
'CNT_CREDIT_PROLONG_SUM_CREDITACTIVE_CLOSED',
'AMT_CREDIT_SUM_SUM_CREDITACTIVE_CLOSED',
'AMT_CREDIT_SUM_MAX_CREDITACTIVE_CLOSED',
'AMT_CREDIT_SUM_DEBT_SUM_CREDITACTIVE_CLOSED',
'AMT_CREDIT_SUM_LIMIT_MAX_CREDITACTIVE_CLOSED',
'AMT_CREDIT_SUM_LIMIT_SUM_CREDITACTIVE_CLOSED',
'AMT_CREDIT_SUM_OVERDUE_MAX_CREDITACTIVE_CLOSED',
```

```
'AMT_CREDIT_SUM_OVERDUE_SUM_CREDITACTIVE_CLOSED',
'DAYS_CREDIT_UPDATE_MEAN_CREDITACTIVE_CLOSED',
'DAYS_CREDIT_UPDATE_MIN_CREDITACTIVE_CLOSED',
'AMT_ANNUITY_MEAN_CREDITACTIVE_CLOSED',
'AMT_ANNUITY_SUM_CREDITACTIVE_CLOSED',
'AMT_ANNUITY_MAX_CREDITACTIVE_CLOSED',
'CREDIT_DURATION_MAX_CREDITACTIVE_CLOSED',
'CREDIT_DURATION_MEAN_CREDITACTIVE_CLOSED',
'FLAG_OVERDUE_RECENT_SUM_CREDITACTIVE_CLOSED',
'MAX_AMT_OVERDUE_DURATION_RATIO_MAX_CREDITACTIVE_CLOSED',
'MAX_AMT_OVERDUE_DURATION_RATIO_SUM_CREDITACTIVE_CLOSED',
'CURRENT_AMT_OVERDUE_DURATION_RATIO_MAX_CREDITACTIVE_CLOSED',
'CURRENT_AMT_OVERDUE_DURATION_RATIO_SUM_CREDITACTIVE_CLOSED',
'AMT_OVERDUE_DURATION_LEFT_RATIO_MAX_CREDITACTIVE_CLOSED',
'AMT_OVERDUE_DURATION_LEFT_RATIO_MEAN_CREDITACTIVE_CLOSED',
'CNT_PROLONGED_MAX_OVERDUE_MUL_MEAN_CREDITACTIVE_CLOSED',
'CNT_PROLONGED_MAX_OVERDUE_MUL_MAX_CREDITACTIVE_CLOSED',
'CNT_PROLONGED_DURATION_RATIO_MEAN_CREDITACTIVE_CLOSED',
'CNT_PROLONGED_DURATION_RATIO_MAX_CREDITACTIVE_CLOSED',
'CURRENT_DEBT_TO_CREDIT_RATIO_MEAN_CREDITACTIVE_CLOSED',
'CURRENT_DEBT_TO_CREDIT_RATIO_MIN_CREDITACTIVE_CLOSED',
'CURRENT_CREDIT_DEBT_DIFF_MEAN_CREDITACTIVE_CLOSED',
'CURRENT_CREDIT_DEBT_DIFF_MIN_CREDITACTIVE_CLOSED',
'AMT_ANNUITY_CREDIT_RATIO_MEAN_CREDITACTIVE_CLOSED',
'AMT_ANNUITY_CREDIT_RATIO_MAX_CREDITACTIVE_CLOSED',
'AMT_ANNUITY_CREDIT_RATIO_MIN_CREDITACTIVE_CLOSED',
'CREDIT_ENDDATE_UPDATE_DIFF_MAX_CREDITACTIVE_CLOSED',
'CREDIT_ENDDATE_UPDATE_DIFF_MIN_CREDITACTIVE_CLOSED',
'STATUS_MEAN_MEAN_CREDITACTIVE_CLOSED',
'STATUS_MEAN_MAX_CREDITACTIVE_CLOSED',
'WEIGHTED_STATUS_MEAN_MEAN_CREDITACTIVE_CLOSED',
'WEIGHTED_STATUS_MEAN_MAX_CREDITACTIVE_CLOSED',
'DAYS_CREDIT_MEAN_CREDITACTIVE_ACTIVE',
'DAYS_CREDIT_MIN_CREDITACTIVE_ACTIVE',
'DAYS_CREDIT_MAX_CREDITACTIVE_ACTIVE',
'DAYS_CREDIT_LAST_CREDITACTIVE_ACTIVE',
'CREDIT_DAY_OVERDUE_MEAN_CREDITACTIVE_ACTIVE',
'CREDIT_DAY_OVERDUE_MAX_CREDITACTIVE_ACTIVE',
'DAYS_CREDIT_ENDDATE_MEAN_CREDITACTIVE_ACTIVE',
'DAYS_CREDIT_ENDDATE_MAX_CREDITACTIVE_ACTIVE',
'DAYS_ENDDATE_FACT_MEAN_CREDITACTIVE_ACTIVE',
'DAYS_ENDDATE_FACT_MIN_CREDITACTIVE_ACTIVE',
'AMT_CREDIT_MAX_OVERDUE_MAX_CREDITACTIVE_ACTIVE',
'AMT_CREDIT_MAX_OVERDUE_SUM_CREDITACTIVE_ACTIVE',
'CNT_CREDIT_PROLONG_MAX_CREDITACTIVE_ACTIVE',
'CNT_CREDIT_PROLONG_SUM_CREDITACTIVE_ACTIVE',
'AMT_CREDIT_SUM_SUM_CREDITACTIVE_ACTIVE',
'AMT_CREDIT_SUM_MAX_CREDITACTIVE_ACTIVE',
'AMT_CREDIT_SUM_DEBT_SUM_CREDITACTIVE_ACTIVE',
'AMT_CREDIT_SUM_LIMIT_MAX_CREDITACTIVE_ACTIVE',
'AMT_CREDIT_SUM_LIMIT_SUM_CREDITACTIVE_ACTIVE',
'AMT_CREDIT_SUM_OVERDUE_MAX_CREDITACTIVE_ACTIVE',
'AMT_CREDIT_SUM_OVERDUE_SUM_CREDITACTIVE_ACTIVE',
'DAYS_CREDIT_UPDATE_MEAN_CREDITACTIVE_ACTIVE',
'DAYS_CREDIT_UPDATE_MIN_CREDITACTIVE_ACTIVE',
'AMT_ANNUITY_MEAN_CREDITACTIVE_ACTIVE',
'AMT_ANNUITY_SUM_CREDITACTIVE_ACTIVE',
'AMT_ANNUITY_MAX_CREDITACTIVE_ACTIVE',
'CREDIT_DURATION_MAX_CREDITACTIVE_ACTIVE',
'CREDIT_DURATION_MEAN_CREDITACTIVE_ACTIVE',
```

```
'FLAG_OVERDUE_RECENT_SUM_CREDITACTIVE_ACTIVE',
'MAX_AMT_OVERDUE_DURATION_RATIO_MAX_CREDITACTIVE_ACTIVE',
'MAX_AMT_OVERDUE_DURATION_RATIO_SUM_CREDITACTIVE_ACTIVE',
'CURRENT_AMT_OVERDUE_DURATION_RATIO_MAX_CREDITACTIVE_ACTIVE',
'CURRENT_AMT_OVERDUE_DURATION_RATIO_SUM_CREDITACTIVE_ACTIVE',
'AMT_OVERDUE_DURATION_LEFT_RATIO_MAX_CREDITACTIVE_ACTIVE',
'AMT_OVERDUE_DURATION_LEFT_RATIO_MEAN_CREDITACTIVE_ACTIVE',
'CNT_PROLONGED_MAX_OVERDUE_MUL_MEAN_CREDITACTIVE_ACTIVE',
'CNT_PROLONGED_MAX_OVERDUE_MUL_MAX_CREDITACTIVE_ACTIVE',
'CNT_PROLONGED_DURATION_RATIO_MEAN_CREDITACTIVE_ACTIVE',
'CNT_PROLONGED_DURATION_RATIO_MAX_CREDITACTIVE_ACTIVE',
'CURRENT_DEBT_TO_CREDIT_RATIO_MEAN_CREDITACTIVE_ACTIVE',
'CURRENT_DEBT_TO_CREDIT_RATIO_MIN_CREDITACTIVE_ACTIVE',
'CURRENT_CREDIT_DEBT_DIFF_MEAN_CREDITACTIVE_ACTIVE',
'CURRENT_CREDIT_DEBT_DIFF_MIN_CREDITACTIVE_ACTIVE',
'AMT_ANNUITY_CREDIT_RATIO_MEAN_CREDITACTIVE_ACTIVE',
'AMT_ANNUITY_CREDIT_RATIO_MAX_CREDITACTIVE_ACTIVE',
'AMT_ANNUITY_CREDIT_RATIO_MIN_CREDITACTIVE_ACTIVE',
'CREDIT_ENDDATE_UPDATE_DIFF_MAX_CREDITACTIVE_ACTIVE',
'CREDIT_ENDDATE_UPDATE_DIFF_MIN_CREDITACTIVE_ACTIVE',
'STATUS_MEAN_MEAN_CREDITACTIVE_ACTIVE',
'STATUS_MEAN_MAX_CREDITACTIVE_ACTIVE',
'WEIGHTED_STATUS_MEAN_MEAN_CREDITACTIVE_ACTIVE',
'WEIGHTED_STATUS_MEAN_MAX_CREDITACTIVE_ACTIVE',
'DAYS_CREDIT_MEANCREDIT_ACTIVE_REST',
'DAYS_CREDIT_MINCREDIT_ACTIVE_REST',
'DAYS_CREDIT_MAXCREDIT_ACTIVE_REST',
'DAYS_CREDIT_LASTCREDIT_ACTIVE_REST',
'CREDIT_DAY_OVERDUE_MEANCREDIT_ACTIVE_REST',
'CREDIT_DAY_OVERDUE_MAXCREDIT_ACTIVE_REST',
'DAYS_CREDIT_ENDDATE_MEANCREDIT_ACTIVE_REST',
'DAYS_CREDIT_ENDDATE_MAXCREDIT_ACTIVE_REST',
'DAYS_ENDDATE_FACT_MEANCREDIT_ACTIVE_REST',
'DAYS_ENDDATE_FACT_MINCREDIT_ACTIVE_REST',
'AMT_CREDIT_MAX_OVERDUE_MAXCREDIT_ACTIVE_REST',
'AMT_CREDIT_MAX_OVERDUE_SUMCREDIT_ACTIVE_REST',
'CNT_CREDIT_PROLONG_MAXCREDIT_ACTIVE_REST',
'CNT_CREDIT_PROLONG_SUMCREDIT_ACTIVE_REST',
'AMT_CREDIT_SUM_SUMCREDIT_ACTIVE_REST',
'AMT_CREDIT_SUM_MAXCREDIT_ACTIVE_REST',
'AMT_CREDIT_SUM_DEBT_SUMCREDIT_ACTIVE_REST',
'AMT_CREDIT_SUM_LIMIT_MAXCREDIT_ACTIVE_REST',
'AMT_CREDIT_SUM_LIMIT_SUMCREDIT_ACTIVE_REST',
'AMT_CREDIT_SUM_OVERDUE_MAXCREDIT_ACTIVE_REST',
'AMT_CREDIT_SUM_OVERDUE_SUMCREDIT_ACTIVE_REST',
'DAYS_CREDIT_UPDATE_MEANCREDIT_ACTIVE_REST',
'DAYS_CREDIT_UPDATE_MINCREDIT_ACTIVE_REST',
'AMT_ANNUITY_MEANCREDIT_ACTIVE_REST',
'AMT_ANNUITY_SUMCREDIT_ACTIVE_REST',
'AMT_ANNUITY_MAXCREDIT_ACTIVE_REST',
'CREDIT_DURATION_MAXCREDIT_ACTIVE_REST',
'CREDIT_DURATION_MEANCREDIT_ACTIVE_REST',
'FLAG_OVERDUE_RECENT_SUMCREDIT_ACTIVE_REST',
'MAX_AMT_OVERDUE_DURATION_RATIO_MAXCREDIT_ACTIVE_REST',
'MAX_AMT_OVERDUE_DURATION_RATIO_SUMCREDIT_ACTIVE_REST',
'CURRENT_AMT_OVERDUE_DURATION_RATIO_MAXCREDIT_ACTIVE_REST',
'CURRENT_AMT_OVERDUE_DURATION_RATIO_SUMCREDIT_ACTIVE_REST',
'AMT_OVERDUE_DURATION_LEFT_RATIO_MAXCREDIT_ACTIVE_REST',
'AMT_OVERDUE_DURATION_LEFT_RATIO_MEANCREDIT_ACTIVE_REST',
'CNT_PROLONGED_MAX_OVERDUE_MUL_MEANCREDIT_ACTIVE_REST',
```

```
 'CNT_PROLONGED_MAX_OVERDUE_MUL_MAXCREDIT_ACTIVE_REST',
 'CNT_PROLONGED_DURATION_RATIO_MEANCREDIT_ACTIVE_REST',
 'CNT_PROLONGED_DURATION_RATIO_MAXCREDIT_ACTIVE_REST',
 'CURRENT_DEBT_TO_CREDIT_RATIO_MEANCREDIT_ACTIVE_REST',
 'CURRENT_DEBT_TO_CREDIT_RATIO_MINCREDIT_ACTIVE_REST',
 'CURRENT_CREDIT_DEBT_DIFF_MEANCREDIT_ACTIVE_REST',
 'CURRENT_CREDIT_DEBT_DIFF_MINCREDIT_ACTIVE_REST',
 'AMT_ANNUITY_CREDIT_RATIO_MEANCREDIT_ACTIVE_REST',
 'AMT_ANNUITY_CREDIT_RATIO_MAXCREDIT_ACTIVE_REST',
 'AMT_ANNUITY_CREDIT_RATIO_MINCREDIT_ACTIVE_REST',
 'CREDIT_ENDDATE_UPDATE_DIFF_MAXCREDIT_ACTIVE_REST',
 'CREDIT_ENDDATE_UPDATE_DIFF_MINCREDIT_ACTIVE_REST',
 'STATUS_MEAN_MEANCREDIT_ACTIVE_REST',
 'STATUS_MEAN_MAXCREDIT_ACTIVE_REST',
 'WEIGHTED_STATUS_MEAN_MEANCREDIT_ACTIVE_REST',
 'WEIGHTED_STATUS_MEAN_MAXCREDIT_ACTIVE_REST']
```

In [ ]:

```
# bureau_aggregated = preprocess_bureau_balance_and_bureau(file_directory='./data/',dum
# aggregated_bureau_balance = bureau_aggregated.preprocess_bureau_balance()
```

In [ ]:

```
# bureau_merged_aggregated = bureau_aggregated.preprocess_bureau(aggregated_bureau_bala
```

In [ ]:

```
# bureau_aggregated = bureau_merged_aggregated
```

2.1.2 previous_application.csv

This table contains the static data related to clients and their previous credits with Home Credit Group.

1. First we start by cleaning the erroneous values. From the EDA we saw some DAYS fields with a value equal to 365243.0, they look erroneous, and so we will be replacing them with NaN values.
2. We replace the NaN values for categories with an 'XNA' category.
3. Next, we proceed to feature engineering, where we create some domain based features, such as Credit-Downpayment Ratio, Amount not approved, Credit to Goods ratio, etc.
4. We also try to predict the interest rate, inspired by one of the writeups of winners.
5. To be able to merge it with main table, we need to aggregate the rows of previous_application over SK_ID_CURR. We perform domain based aggregations, over all the previous credits for each customer, such as mean, max, min, etc. Here again we aggregate in three ways. First we perform overall aggregation, next we aggregate for first 2 applications and latest 5 applications. The First and Last are decided by the DAYS_FIRST_DUE of applications. In the end, we merge all these aggregations together.

In [22]:

```
class preprocess_previous_application:
    '''
    Preprocess the previous_application table.
    Contains 5 member functions:
        1. init method
        2. load_dataframe method
```

```
        3. data_cleaning method
        4. preprocessing_feature_engineering method
        5. main method
    '''
    def __init__(self, file_directory='', verbose=True, dump_to_pickle=False):
        '''
        This function is used to initialize the class members

        Inputs:
            self
            file_directory: Path, str, default = ''
                The path where the file exists. Include a '/' at the end of the path in
            verbose: bool, default = True
                Whether to enable verbosity or not
            dump_to_pickle: bool, default = False
                Whether to pickle the final preprocessed table or not

        Returns:
            None
        '''
        self.file_directory = file_directory
        self.verbose = verbose
        self.dump_to_pickle = dump_to_pickle

    def load_dataframe(self):
        '''
        Function to load the previous_application.csv DataFrame.

        Inputs:
            self

        Returns:
            None
        '''
        if self.verbose:
            self.start = datetime.now()
            print('#########################################################')
            print('#        Pre-processing previous_application.csv        #')
            print('#########################################################')
            print("\nLoading the DataFrame, previous_application.csv, into memory...")

        #Loading the DataFrame into memory
        self.previous_application = pd.read_csv(self.file_directory + 'previous_applica
        self.initial_shape = self.previous_application.shape

        if self.verbose:
            print("Loaded previous_application.csv")
            print(f"Time Taken to load = {datetime.now() - self.start}")

    def data_cleaning(self):
        '''
        Function to clean the data. Removes erroneous points, fills categorical NaNs wi

        Inputs:
            self

        Returns:
            None
        '''
        if self.verbose:
```

```
            start = datetime.now()
            print('\nStarting Data Cleaning...')

        #sorting the application from oldest to most recent previous loans for each use
        self.previous_application = self.previous_application.sort_values(by=['SK_ID_CU

        #in the EDA, we found some erroneous values in DAYS columns, so we will replace
        self.previous_application['DAYS_FIRST_DRAWING'][self.previous_application['DAYS

        self.previous_application['DAYS_FIRST_DRAWING'][self.previous_application['DAYS
        self.previous_application['DAYS_FIRST_DUE'][self.previous_application['DAYS_FIR
        self.previous_application['DAYS_LAST_DUE_1ST_VERSION'][self.previous_applicatio
        self.previous_application['DAYS_LAST_DUE'][self.previous_application['DAYS_LAST
        self.previous_application['DAYS_TERMINATION'][self.previous_application['DAYS_T
        #we also see abruply large value for SELLERPLACE_AREA
        self.previous_application['SELLERPLACE_AREA'][self.previous_application['SELLER
        #filling the nan values for the categories
        categorical_columns = self.previous_application.dtypes[self.previous_applicatio
        self.previous_application[categorical_columns] = self.previous_application[cate

        if self.verbose:
            print("Done.")
            print(f"Time taken = {datetime.now() - start}")

    def preprocessing_feature_engineering(self):
        '''
        Function to do preprocessing such as categorical encoding and feature engineeri

        Inputs:
            self

        Returns:
            None
        '''
        if self.verbose:
            start = datetime.now()
            print("\nPerforming Preprocessing and Feature Engineering...")

        #label encoding the categorical variables
        name_contract_dict = {'Approved': 0, 'Refused' : 3, 'Canceled' : 2, 'Unused off
        self.previous_application['NAME_CONTRACT_STATUS'] = self.previous_application['
        yield_group_dict = {'XNA': 0, 'low_action': 1, 'low_normal': 2,'middle': 3, 'hi
        self.previous_application['NAME_YIELD_GROUP'] = self.previous_application['NAME
        appl_per_contract_last_dict = {'Y':1, 'N':0}
        self.previous_application['FLAG_LAST_APPL_PER_CONTRACT'] = self.previous_applic
        remaining_categorical_columns = self.previous_application.dtypes[self.previous_
        for col in remaining_categorical_columns:
            encoding_dict = dict([(j, i) for i, j in enumerate(self.previous_applicatio
            self.previous_application[col] = self.previous_application[col].map(encodin

        #engineering some features based on domain knowledge
        self.previous_application['MISSING_VALUES_TOTAL_PREV'] = self.previous_applicat
        self.previous_application['AMT_DECLINED'] = self.previous_application['AMT_APPL
        self.previous_application['AMT_CREDIT_GOODS_RATIO'] = self.previous_application
        self.previous_application['AMT_CREDIT_GOODS_DIFF'] = self.previous_application[
        self.previous_application['AMT_CREDIT_APPLICATION_RATIO'] = self.previous_appli
        self.previous_application['CREDIT_DOWNPAYMENT_RATIO'] = self.previous_applicati
        self.previous_application['GOOD_DOWNPAYMET_RATIO'] = self.previous_application[
        self.previous_application['INTEREST_DOWNPAYMENT'] = self.previous_application['
        self.previous_application['INTEREST_CREDIT'] = self.previous_application['AMT_C
```

```
        self.previous_application['INTEREST_CREDIT_PRIVILEGED'] = self.previous_applica
        self.previous_application['APPLICATION_AMT_TO_DECISION_RATIO'] = self.previous_
        self.previous_application['AMT_APPLICATION_TO_SELLERPLACE_AREA'] = self.previou
        self.previous_application['ANNUITY'] = self.previous_application['AMT_CREDIT']
        self.previous_application['ANNUITY_GOODS'] = self.previous_application['AMT_GOO
        self.previous_application['DAYS_FIRST_LAST_DUE_DIFF' ] = self.previous_applicat
        self.previous_application['AMT_CREDIT_HOUR_PROCESS_START'] = self.previous_appl
        self.previous_application['AMT_CREDIT_NFLAG_LAST_APPL_DAY'] = self.previous_app
        self.previous_application['AMT_CREDIT_YIELD_GROUP'] = self.previous_application
        #https://www.kaggle.com/c/home-credit-default-risk/discussion/64598
        self.previous_application['AMT_INTEREST'] = self.previous_application['CNT_PAYM
                                                'AMT_ANNUITY'] - self.previous_applicat
        self.previous_application['INTEREST_SHARE'] = self.previous_application['AMT_IN

        self.previous_application['INTEREST_RATE'] = 2 * 12 * self.previous_application
                                                'AMT_CREDIT'] * (self.previous_application[

        if self.verbose:
            print("Done.")
            print(f"Time taken = {datetime.now() - start}")

    def aggregations(self):
        '''
        Function to aggregate the previous applications over SK_ID_CURR

        Inputs:
            self

        Returns:
            aggregated previous_applications
        '''
        if self.verbose:
            print("\nAggregating previous applications over SK_ID_CURR...")

        aggregations_for_previous_application = {
            'MISSING_VALUES_TOTAL_PREV' : ['sum'],
            'NAME_CONTRACT_TYPE' : ['mean','last'],
            'AMT_ANNUITY' : ['mean','sum','max'],
            'AMT_APPLICATION' : ['mean','max','sum'],
            'AMT_CREDIT' : ['mean','max','sum'],
            'AMT_DOWN_PAYMENT' : ['mean','max','sum'],
            'AMT_GOODS_PRICE' : ['mean','max','sum'],
            'WEEKDAY_APPR_PROCESS_START' : ['mean','max','min'],
            'HOUR_APPR_PROCESS_START' : ['mean','max','min'],
            'FLAG_LAST_APPL_PER_CONTRACT' : ['mean','sum'],
            'NFLAG_LAST_APPL_IN_DAY' : ['mean','sum'],
            'RATE_DOWN_PAYMENT' : ['mean','max'],
            'RATE_INTEREST_PRIMARY' : ['mean','max'],
            'RATE_INTEREST_PRIVILEGED' : ['mean','max'],
            'NAME_CASH_LOAN_PURPOSE' : ['mean','last'],
            'NAME_CONTRACT_STATUS' : ['mean','max','last'],
            'DAYS_DECISION' : ['mean','max','min'],
            'NAME_PAYMENT_TYPE' : ['mean', 'last'],
            'CODE_REJECT_REASON' : ['mean','last'],
            'NAME_TYPE_SUITE' : ['mean','last'],
            'NAME_CLIENT_TYPE' : ['mean','last'],
            'NAME_GOODS_CATEGORY' : ['mean','last'],
            'NAME_PORTFOLIO' : ['mean','last'],
            'NAME_PRODUCT_TYPE' : ['mean','last'],
            'CHANNEL_TYPE' : ['mean','last'],
```

```
            'SELLERPLACE_AREA' : ['mean','max','min'],
            'NAME_SELLER_INDUSTRY' : ['mean','last'],
            'CNT_PAYMENT' : ['sum','mean','max'],
            'NAME_YIELD_GROUP' : ['mean','last'],
            'PRODUCT_COMBINATION' : ['mean', 'last'],
            'DAYS_FIRST_DRAWING' : ['mean','max'],
            'DAYS_FIRST_DUE' : ['mean','max'],
            'DAYS_LAST_DUE_1ST_VERSION' : ['mean'],
            'DAYS_LAST_DUE' : ['mean'],
            'DAYS_TERMINATION' : ['mean','max'],
            'NFLAG_INSURED_ON_APPROVAL' : ['sum'],
            'AMT_DECLINED' : ['mean','max','sum'],
            'AMT_CREDIT_GOODS_RATIO' : ['mean', 'max', 'min'],
            'AMT_CREDIT_GOODS_DIFF' : ['sum','mean','max', 'min'],
            'AMT_CREDIT_APPLICATION_RATIO' : ['mean','min'],
            'CREDIT_DOWNPAYMENT_RATIO' : ['mean','max'],
            'GOOD_DOWNPAYMET_RATIO' : ['mean','max'],
            'INTEREST_DOWNPAYMENT' : ['mean','sum','max'],
            'INTEREST_CREDIT' : ['mean','sum','max'],
            'INTEREST_CREDIT_PRIVILEGED' : ['mean','sum','max'],
            'APPLICATION_AMT_TO_DECISION_RATIO' : ['mean','min'],
            'AMT_APPLICATION_TO_SELLERPLACE_AREA' : ['mean','max'],
            'ANNUITY' : ['mean','sum','max'],
            'ANNUITY_GOODS' : ['mean','sum','max'],
            'DAYS_FIRST_LAST_DUE_DIFF' : ['mean','max'],
            'AMT_CREDIT_HOUR_PROCESS_START' : ['mean','sum'],
            'AMT_CREDIT_NFLAG_LAST_APPL_DAY' : ['mean','max'],
            'AMT_CREDIT_YIELD_GROUP' : ['mean','sum','min'],
            'AMT_INTEREST' : ['mean','sum','max','min'],
            'INTEREST_SHARE' : ['mean','max','min'],
            'INTEREST_RATE' : ['mean','max','min']
        }

        #grouping the previous applications over SK_ID_CURR while only taking the lates
        group_last_5 = self.previous_application.groupby('SK_ID_CURR').tail(5).groupby(
        group_last_5.columns = ['_'.join(ele).upper() + '_LAST_5' for ele in group_last_
        #grouping the previous applications over SK_ID_CURR while only take the first 2
        group_first_2 = self.previous_application.groupby('SK_ID_CURR').head(2).groupby
        group_first_2.columns = ['_'.join(ele).upper() + '_FIRST_2' for ele in group_fi
        #grouping the previous applications over SK_ID_CURR while taking all the applic
        group_all = self.previous_application.groupby('SK_ID_CURR').agg(aggregations_fo
        group_all.columns = ['_'.join(ele).upper() + '_ALL' for ele in group_all.column

        #merging all the applications
        previous_application_aggregated = group_last_5.merge(group_first_2, on='SK_ID_C
        preprocessing_previous_application = previous_application_aggregated.merge(grou

        return preprocessing_previous_application

    def main(self):
        '''
        Function to be called for complete preprocessing and aggregation of previous_ap

        Inputs:
            self

        Returns:
            Final pre=processed and aggregated previous_application table.
        '''
        #loading the dataframe
```

```
        self.load_dataframe()

        #cleaning the data
        self.data_cleaning()

        #preproccesing the categorical features and creating new features
        self.preprocessing_feature_engineering()

        #aggregating data over SK_ID_CURR
        previous_application_aggregated = self.aggregations()

        if self.verbose:
            print('Done aggregations.')
            print(f"\nInitial Size of previous_application: {self.initial_shape}")
            print(f'Size of previous_application after Pre-Processing, Feature Engineer
            print(f'\nTotal Time Taken = {datetime.now() - self.start}')

        if self.dump_to_pickle:
            if self.verbose:
                print('\nPickling pre-processed previous_application to previous_applic
            with open(self.file_directory + 'previous_application_preprocessed.pkl', 'w
                pickle.dump(previous_application_aggregated, f)
            if self.verbose:
                print('Done.')
        if self.verbose:
            print('-'*100)

        return previous_application_aggregated
```

In [23]:

```
previous_aggregated = preprocess_previous_application(file_directory='./data/', dump_to
```

```
#########################################################
#        Pre-processing previous_application.csv        #
#########################################################

Loading the DataFrame, previous_application.csv, into memory...
Loaded previous_application.csv
Time Taken to load = 0:00:03.911999

Starting Data Cleaning...
Done.
Time taken = 0:00:02.897001

Performing Preprocessing and Feature Engineering...
Done.
Time taken = 0:00:03.126723

Aggregating previous applications over SK_ID_CURR...
Done aggregations.

Initial Size of previous_application: (1670214, 37)
Size of previous_application after Pre-Processing, Feature Engineering and Aggregation:
(338857, 399)

Total Time Taken = 0:00:21.521721
```

```
Pickling pre-processed previous_application to previous_application_preprocessed.pkl
Done.
-------------------------------------------------------------------------------------------
------------
```

In [ ]:

```
# pre_app = pd.read_csv('./data/previous_application.csv');
```

In [ ]:

```
# pre_app.shape
# pre_app.dtypes
# sm=pre_app.isna().sum(axis=1)
```

2.1.3 installments_payments.csv

This table contains the details about each installment of client's previous credits with Home Credit Group.

1. We start by sorting the data first by SK_ID_CURR and SK_ID_PREV, and then by NUM_INSTALMENT_NUMBER. This brings the latest installments in the end.
2. We create some features, such as the number of days the payment was delayed, the difference in amount of payment required vs paid, etc.
3. Next we aggregate these rows over SK_ID_PREV, such that each client's previous loan gets one row. These aggregations are done in three ways, first overall aggregations, second we aggregate only those installments which were in the last 365 days, and lastly, we aggregate the first 5 installments of every loan. This will help us to capture the starting behaviour, the latest behaviour and the overall behaviour of the client's installments payments.
4. Now to merge this table with main table, we aggregate the data over SK_ID_CURR.

In [26]:

```
class preprocess_installments_payments:
    '''
    Preprocess the installments_payments table.
    Contains 6 member functions:
        1. init method
        2. load_dataframe method
        3. data_preprocessing_and_feature_engineering method
        4. aggregations_sk_id_prev method
        5. aggregations_sk_id_curr method
        6. main method
    '''
    def __init__(self, file_directory='', verbose=True, dump_to_pickle=True):
        '''
        This function is used to initialize the class members

        Inputs:
            self
            file_directory: Path, str, default = ''
                The path where the file exists. Include a '/' at the end of the path in
            verbose: bool, default = True
                Whether to enable verbosity or not
            dump_to_pickle: bool, default = False
                Whether to pickle the final preprocessed table or not
```

```
        Returns:
            None
        '''
        self.file_directory = file_directory
        self.verbose = verbose
        self.dump_to_pickle = dump_to_pickle

    def load_dataframe(self):
        '''
        Function to load the installments_payments.csv DataFrame.

        Inputs:
            self

        Returns:
            None
        '''
        if self.verbose:
            self.start = datetime.now()
            print('##########################################################')
            print('#        Pre-processing installments_payments.csv       #')
            print('##########################################################')
            print("\nLoading the DataFrame, installments_payments.csv, into memory...")

        self.installments_payments = pd.read_csv(self.file_directory + 'installments_pa
        self.initial_shape = self.installments_payments.shape

        if self.verbose:
            print("Loaded previous_application.csv")
            print(f"Time Taken to load = {datetime.now() - self.start}")

    def data_preprocessing_and_feature_engineering(self):
        '''
        Function for pre-processing and feature engineering

        Inputs:
            self

        Returns:
            None
        '''
        if self.verbose:
            start = datetime.now()
            print("\nStarting Data Pre-processing and Feature Engineering...")

        #sorting by SK_ID_PREV and NUM_INSTALMENT_NUMBER
        self.installments_payments = self.installments_payments.sort_values(by=['SK_ID_

        #getting the total NaN values in the table
        self.installments_payments['MISSING_VALS_TOTAL_INSTAL'] = self.installments_pay
        #engineering new features based on some domain based polynomial operations
        self.installments_payments['DAYS_PAYMENT_RATIO'] = self.installments_payments['
        self.installments_payments['DAYS_PAYMENT_DIFF'] = self.installments_payments['D
        self.installments_payments['AMT_PAYMENT_RATIO'] = self.installments_payments['A
        self.installments_payments['AMT_PAYMENT_DIFF'] = self.installments_payments['AM
        self.installments_payments['EXP_DAYS_PAYMENT_RATIO'] = self.installments_paymen
        self.installments_payments['EXP_DAYS_PAYMENT_DIFF'] = self.installments_payment
        self.installments_payments['EXP_AMT_PAYMENT_RATIO'] = self.installments_payment
        self.installments_payments['EXP_AMT_PAYMENT_DIFF'] = self.installments_payments
```

```
        if self.verbose:
            print("Done.")
            print(f"Time Taken = {datetime.now() - start}")

    def aggregations_sk_id_pre(self):
        '''
        Function for aggregations of installments on previous loans over SK_ID_PREV

        Inputs:
            self

        Returns:
            installments_payments table aggregated over previous loans
        '''
        if self.verbose:
            start = datetime.now()
            print("\nPerforming Aggregations over SK_ID_PREV...")

        #aggregating the data over SK_ID_PREV , i.e. for each previous loan
        overall_aggregations = {
            'MISSING_VALS_TOTAL_INSTAL' : ['sum'],
            'NUM_INSTALMENT_VERSION' : ['mean','sum'],
            'NUM_INSTALMENT_NUMBER' : ['max'],
            'DAYS_INSTALMENT' : ['max','min'],
            'DAYS_ENTRY_PAYMENT' : ['max','min'],
            'AMT_INSTALMENT' : ['mean', 'sum', 'max'],
            'AMT_PAYMENT' : ['mean', 'sum', 'max'],
            'DAYS_PAYMENT_RATIO' : ['mean', 'min','max'],
            'DAYS_PAYMENT_DIFF' : ['mean','min','max'],
            'AMT_PAYMENT_RATIO' : ['mean','min','max'],
            'AMT_PAYMENT_DIFF' : ['mean','min','max'],
            'EXP_DAYS_PAYMENT_RATIO' : ['last'],
            'EXP_DAYS_PAYMENT_DIFF' : ['last'],
            'EXP_AMT_PAYMENT_RATIO' : ['last'],
            'EXP_AMT_PAYMENT_DIFF' : ['last']
        }
        limited_period_aggregations = {
            'NUM_INSTALMENT_VERSION' : ['mean','sum'],
            'AMT_INSTALMENT' : ['mean', 'sum', 'max'],
            'AMT_PAYMENT' : ['mean', 'sum', 'max'],
            'DAYS_PAYMENT_RATIO' : ['mean', 'min','max'],
            'DAYS_PAYMENT_DIFF' : ['mean','min','max'],
            'AMT_PAYMENT_RATIO' : ['mean','min','max'],
            'AMT_PAYMENT_DIFF' : ['mean','min','max'],
            'EXP_DAYS_PAYMENT_RATIO' : ['last'],
            'EXP_DAYS_PAYMENT_DIFF' : ['last'],
            'EXP_AMT_PAYMENT_RATIO' : ['last'],
            'EXP_AMT_PAYMENT_DIFF' : ['last']
        }

        #aggregating installments_payments over SK_ID_PREV for last 1 year installments
        group_last_1_year = self.installments_payments[self.installments_payments['DAYS_
        group_last_1_year.columns = ['_'.join(ele).upper() + '_LAST_1_YEAR' for ele in 
        #aggregating installments_payments over SK_ID_PREV for first 5 installments
        group_first_5_installments = self.installments_payments.groupby('SK_ID_PREV', a
        group_first_5_installments.columns = ['_'.join(ele).upper() + '_FIRST_5_INSTALL
        #overall aggregation of installments_payments over SK_ID_PREV
        group_overall = self.installments_payments.groupby(['SK_ID_PREV', 'SK_ID_CURR']
        group_overall.columns = ['_'.join(ele).upper() for ele in group_overall.columns
        group_overall.rename(columns = {'SK_ID_PREV_': 'SK_ID_PREV','SK_ID_CURR_' : 'SK
```

```
        #merging all of the above aggregations together
        installments_payments_agg_prev = group_overall.merge(group_last_1_year, on='SK_
        installments_payments_agg_prev = installments_payments_agg_prev.merge(group_fir

        if self.verbose:
            print("Done.")
            print(f"Time Taken = {datetime.now() - start}")

        return installments_payments_agg_prev
    def aggregations_sk_id_curr(self, installments_payments_agg_prev):
        '''
        Function to aggregate the installments payments on previous loans over SK_ID_CU

        Inputs:
            self
            installments_payments_agg_prev: DataFrame
                installments payments aggregated over SK_ID_PREV

        Returns:
            installments payments aggregated over SK_ID_CURR
        '''
        #aggregating over SK_ID_CURR
        main_features_aggregations = {
            'MISSING_VALS_TOTAL_INSTAL_SUM' : ['sum'],
            'NUM_INSTALMENT_VERSION_MEAN' : ['mean'],
            'NUM_INSTALMENT_VERSION_SUM' : ['mean'],
            'NUM_INSTALMENT_NUMBER_MAX' : ['mean','sum','max'],
            'AMT_INSTALMENT_MEAN' : ['mean','sum','max'],
            'AMT_INSTALMENT_SUM' : ['mean','sum','max'],
            'AMT_INSTALMENT_MAX' : ['mean'],
            'AMT_PAYMENT_MEAN' : ['mean','sum','max'],
            'AMT_PAYMENT_SUM' : ['mean','sum','max'],
            'AMT_PAYMENT_MAX' : ['mean'],
            'DAYS_PAYMENT_RATIO_MEAN' : ['mean','min','max'],
            'DAYS_PAYMENT_RATIO_MIN' : ['mean','min'],
            'DAYS_PAYMENT_RATIO_MAX' : ['mean','max'],
            'DAYS_PAYMENT_DIFF_MEAN' : ['mean','min','max'],
            'DAYS_PAYMENT_DIFF_MIN' : ['mean','min'],
            'DAYS_PAYMENT_DIFF_MAX' : ['mean','max'],
            'AMT_PAYMENT_RATIO_MEAN' : ['mean', 'min','max'],
            'AMT_PAYMENT_RATIO_MIN' : ['mean','min'],
            'AMT_PAYMENT_RATIO_MAX' : ['mean','max'],
            'AMT_PAYMENT_DIFF_MEAN' : ['mean','min','max'],
            'AMT_PAYMENT_DIFF_MIN' : ['mean','min'],
            'AMT_PAYMENT_DIFF_MAX' : ['mean','max'],
            'EXP_DAYS_PAYMENT_RATIO_LAST' : ['mean'],
            'EXP_DAYS_PAYMENT_DIFF_LAST' : ['mean'],
            'EXP_AMT_PAYMENT_RATIO_LAST' : ['mean'],
            'EXP_AMT_PAYMENT_DIFF_LAST' : ['mean']
        }

        grouped_main_features = installments_payments_agg_prev.groupby('SK_ID_CURR').ag
        grouped_main_features.columns = ['_'.join(ele).upper() for ele in grouped_main_

        #grouping remaining ones
        grouped_remaining_features = installments_payments_agg_prev.iloc[:, [1] + list(
                                        groupby('SK_ID_CURR').mean()
        installments_payments_aggregated = grouped_main_features.merge(grouped_remainin
```

```
        return installments_payments_aggregated

    def main(self):
        '''
        Function to be called for complete preprocessing and aggregation of installment

        Inputs:
            self

        Returns:
            Final pre=processed and aggregated installments_payments table.
        '''
        #loading the dataframe
        self.load_dataframe()
        #doing preprocessing and feature engineering
        self.data_preprocessing_and_feature_engineering()
        #First aggregating the data for each SK_ID_PREV
        installments_payments_agg_prev = self.aggregations_sk_id_pre()

        if self.verbose:
            print("\nAggregations over SK_ID_CURR...")
        #aggregating the previous loans for each SK_ID_CURR
        installments_payments_aggregated = self.aggregations_sk_id_curr(installments_pa

        if self.verbose:
            print('\nDone preprocessing installments_payments.')
            print(f"\nInitial Size of installments_payments: {self.initial_shape}")
            print(f'Size of installments_payments after Pre-Processing, Feature Enginee
            print(f'\nTotal Time Taken = {datetime.now() - self.start}')

        if self.dump_to_pickle:
            if self.verbose:
                print('\nPickling pre-processed installments_payments to installments_p
            with open(self.file_directory + 'installments_payments_preprocessed.pkl', '
                pickle.dump(installments_payments_aggregated, f)
            if self.verbose:
                print('Done.')
        if self.verbose:
            print('-'*100)

        return installments_payments_aggregated
```

In [27]:

```
installments_aggregated = preprocess_installments_payments(file_directory='./data/', du
```

```
##########################################################
#        Pre-processing installments_payments.csv       #
##########################################################

Loading the DataFrame, installments_payments.csv, into memory...
Loaded previous_application.csv
Time Taken to load = 0:00:06.875998

Starting Data Pre-processing and Feature Engineering...
Done.
Time Taken = 0:00:10.552000

Performing Aggregations over SK_ID_PREV...
Done.
```

```
Time Taken = 0:00:13.778001

Aggregations over SK_ID_CURR...

Done preprocessing installments_payments.

Initial Size of installments_payments: (13605401, 8)
Size of installments_payments after Pre-Processing, Feature Engineering and Aggregation: 
(339587, 101)

Total Time Taken = 0:00:33.832001

Pickling pre-processed installments_payments to installments_payments_preprocessed.pkl
Done.
----------------------------------------------------------------------------------------
------------
```

2.1.4 POS_CASH_balance.csv

This table contains the Monthly Balance Snapshots of previous Point of Sales and Cash Loans that the applicant had with Home Credit Group. The table contains columns like the status of contract, the number of installments left, etc.

1. Similar to bureau_balance table, this table also has time based features. So we start off by computing the EDAs on CNT_INSTALMENT and CNT_INSTALMENT_FUTURE features.
2. We create some domain based features next.
3. We then aggregate the data over SK_ID_PREV. For this aggregation, we do it in 3 ways. Firstly we aggregate the whole data over SK_ID_PREV. We also aggregate the data for last 2 years separately and rest of the years separately. Finally, we also aggregate the data different Contract types, i.e. Active and Completed.
4. Next, we aggregate the data over SK_ID_CURR, for it to be merged with main table.

In [28]:

```
class preprocess_POS_CASH_balance:
    '''
    Preprocess the POS_CASH_balance table.
    Contains 6 member functions:
        1. init method
        2. load_dataframe method
        3. data_preprocessing_and_feature_engineering method
        4. aggregations_sk_id_prev method
        5. aggregations_sk_id_curr method
        6. main method
    '''

    def __init__(self, file_directory = '', verbose = True, dump_to_pickle = False):
        '''
        This function is used to initialize the class members

        Inputs:
            self
            file_directory: Path, str, default = ''
                The path where the file exists. Include a '/' at the end of the path in
            verbose: bool, default = True
                Whether to enable verbosity or not
            dump_to_pickle: bool, default = False
```

```
                Whether to pickle the final preprocessed table or not

        Returns:
            None
        '''

        self.file_directory = file_directory
        self.verbose = verbose
        self.dump_to_pickle = dump_to_pickle

    def load_dataframe(self):
        '''
        Function to load the POS_CASH_balance.csv DataFrame.

        Inputs:
            self

        Returns:
            None
        '''

        if self.verbose:
            self.start = datetime.now()
            print('##########################################################')
            print('#          Pre-processing POS_CASH_balance.csv          #')
            print('##########################################################')
            print("\nLoading the DataFrame, POS_CASH_balance.csv, into memory...")

        self.pos_cash = pd.read_csv(self.file_directory + 'POS_CASH_balance.csv')
        self.initial_size = self.pos_cash.shape

        if self.verbose:
            print("Loaded POS_CASH_balance.csv")
            print(f"Time Taken to load = {datetime.now() - self.start}")

    def data_preprocessing_and_feature_engineering(self):
        '''
        Function to preprocess the table and create new features.

        Inputs:
            self

        Returns:
            None
        '''

        if self.verbose:
            start = datetime.now()
            print("\nStarting Data Cleaning and Feature Engineering...")

        #making the MONTHS_BALANCE Positive
        self.pos_cash['MONTHS_BALANCE'] = np.abs(self.pos_cash['MONTHS_BALANCE'])
        #sorting the DataFrame according to the month of status from oldest to latest, 
        self.pos_cash = self.pos_cash.sort_values(by=['SK_ID_PREV', 'MONTHS_BALANCE'],

        #computing Exponential Moving Average for some features based on MONTHS_BALANCE
        columns_for_ema = ['CNT_INSTALMENT', 'CNT_INSTALMENT_FUTURE']
        exp_columns = ['EXP_'+ele for ele in columns_for_ema]
        self.pos_cash[exp_columns] = self.pos_cash.groupby('SK_ID_PREV')[columns_for_em
```

```
        #creating new features based on Domain Knowledge
        self.pos_cash['SK_DPD_RATIO'] = self.pos_cash['SK_DPD'] / (self.pos_cash['SK_DP
        self.pos_cash['TOTAL_TERM'] = self.pos_cash['CNT_INSTALMENT'] + self.pos_cash['
        self.pos_cash['EXP_POS_TOTAL_TERM'] = self.pos_cash['EXP_CNT_INSTALMENT'] + sel

        if self.verbose:
            print("Done.")
            print(f"Time Taken = {datetime.now() - start}")

    def aggregations_sk_id_prev(self):
        '''
        Function to aggregated the POS_CASH_balance rows over SK_ID_PREV

        Inputs:
            self

        Returns:
            Aggregated POS_CASH_balance table over SK_ID_PREV
        '''

        if self.verbose:
            start = datetime.now()
            print("\nAggregations over SK_ID_PREV...")

        #aggregating over SK_ID_PREV
        overall_aggregations = {
            'SK_ID_CURR' : ['first'],
            'MONTHS_BALANCE' : ['max'],
            'CNT_INSTALMENT' : ['mean', 'max','min'],
            'CNT_INSTALMENT_FUTURE' : ['mean','max','min'],
            'SK_DPD' : ['max','sum'],
            'SK_DPD_DEF' : ['max','sum'],
            'EXP_CNT_INSTALMENT' : ['last'],
            'EXP_CNT_INSTALMENT_FUTURE' : ['last'],
            'SK_DPD_RATIO' : ['mean','max'],
            'TOTAL_TERM' : ['mean','max','last'],
            'EXP_POS_TOTAL_TERM' : ['mean']
        }
        aggregations_for_year = {
            'CNT_INSTALMENT' : ['mean', 'max','min'],
            'CNT_INSTALMENT_FUTURE' : ['mean','max','min'],
            'SK_DPD' : ['max','sum'],
            'SK_DPD_DEF' : ['max','sum'],
            'EXP_CNT_INSTALMENT' : ['last'],
            'EXP_CNT_INSTALMENT_FUTURE' : ['last'],
            'SK_DPD_RATIO' : ['mean','max'],
            'TOTAL_TERM' : ['mean','max'],
            'EXP_POS_TOTAL_TERM' : ['last']
        }
        aggregations_for_categories = {
            'CNT_INSTALMENT' : ['mean', 'max','min'],
            'CNT_INSTALMENT_FUTURE' : ['mean','max','min'],
            'SK_DPD' : ['max','sum'],
            'SK_DPD_DEF' : ['max','sum'],
            'EXP_CNT_INSTALMENT' : ['last'],
            'EXP_CNT_INSTALMENT_FUTURE' : ['last'],
            'SK_DPD_RATIO' : ['mean','max'],
            'TOTAL_TERM' : ['mean','max'],
            'EXP_POS_TOTAL_TERM' : ['last']
        }
```

```
        #performing overall aggregations over SK_ID_PREV
        pos_cash_aggregated_overall = self.pos_cash.groupby('SK_ID_PREV').agg(overall_a
        pos_cash_aggregated_overall.columns = ['_'.join(ele).upper() for ele in pos_cas
        pos_cash_aggregated_overall.rename(columns = {'SK_ID_CURR_FIRST': 'SK_ID_CURR'}

        #yearwise aggregations
        self.pos_cash['YEAR_BALANCE'] = self.pos_cash['MONTHS_BALANCE'] //12
        #aggregating over SK_ID_PREV for each last 2 years
        pos_cash_aggregated_year = pd.DataFrame()
        for year in range(2):
            group = self.pos_cash[self.pos_cash['YEAR_BALANCE'] == year].groupby('SK_ID
            group.columns = ['_'.join(ele).upper() + '_YEAR_' + str(year) for ele in gr
            if year == 0:
                pos_cash_aggregated_year = group
            else:
                pos_cash_aggregated_year = pos_cash_aggregated_year.merge(group, on = '

        #aggregating over SK_ID_PREV for rest of the years
        pos_cash_aggregated_rest_years = self.pos_cash[self.pos_cash['YEAR_BALANCE'] >=
        pos_cash_aggregated_rest_years.columns = ['_'.join(ele).upper() + '_YEAR_REST'
        #merging all the years aggregations
        pos_cash_aggregated_year = pos_cash_aggregated_year.merge(pos_cash_aggregated_r
        self.pos_cash = self.pos_cash.drop(['YEAR_BALANCE'], axis = 1)

        #aggregating over SK_ID_PREV for each of NAME_CONTRACT_STATUS categories
        contract_type_categories = ['Active', 'Completed']
        pos_cash_aggregated_contract = pd.DataFrame()
        for i, contract_type in enumerate(contract_type_categories):
            group = self.pos_cash[self.pos_cash['NAME_CONTRACT_STATUS'] == contract_typ
            group.columns = ['_'.join(ele).upper() + '_' + contract_type.upper() for el
            if i == 0:
                pos_cash_aggregated_contract = group
            else:
                pos_cash_aggregated_contract = pos_cash_aggregated_contract.merge(group

        pos_cash_aggregated_rest_contract = self.pos_cash[(self.pos_cash['NAME_CONTRACT
                                        (self.pos_cash['NAME_CONTRACT_STATUS'] != 'Comp
        pos_cash_aggregated_rest_contract.columns = ['_'.join(ele).upper() + '_REST' fo
        #merging the categorical aggregations
        pos_cash_aggregated_contract = pos_cash_aggregated_contract.merge(pos_cash_aggr

        #merging all the aggregations
        pos_cash_aggregated = pos_cash_aggregated_overall.merge(pos_cash_aggregated_yea
        pos_cash_aggregated = pos_cash_aggregated.merge(pos_cash_aggregated_contract, o

        #onehot encoding the categorical feature NAME_CONTRACT_TYPE
        name_contract_dummies = pd.get_dummies(self.pos_cash['NAME_CONTRACT_STATUS'], p
        contract_names = name_contract_dummies.columns.tolist()
        #concatenating one-hot encoded categories with main table
        self.pos_cash = pd.concat([self.pos_cash, name_contract_dummies], axis=1)
        #aggregating these over SK_ID_PREV as well
        aggregated_cc_contract = self.pos_cash[['SK_ID_PREV'] + contract_names].groupby

        #merging with the final aggregations
        pos_cash_aggregated = pos_cash_aggregated.merge(aggregated_cc_contract, on = 'S

        if self.verbose:
            print("Done.")
            print(f"Time Taken = {datetime.now() - start}")
```

```
        return pos_cash_aggregated

    def aggregations_sk_id_curr(self, pos_cash_aggregated):
        '''
        Function to aggregated the aggregateed POS_CASH_balance table over SK_ID_CURR

        Inputs:
            self
            pos_cash_aggregated: DataFrame
                aggregated pos_cash table over SK_ID_PREV

        Returns:
            pos_cash_balance table aggregated over SK_ID_CURR
        '''

        #aggregating over SK_ID_CURR
        columns_to_aggregate = pos_cash_aggregated.columns[1:]
        #defining the aggregations to perform
        aggregations_final = {}
        for col in columns_to_aggregate:
            if 'MEAN' in col:
                aggregates = ['mean','sum','max']
            else:
                aggregates = ['mean']
            aggregations_final[col] = aggregates
        pos_cash_aggregated_final = pos_cash_aggregated.groupby('SK_ID_CURR').agg(aggre
        pos_cash_aggregated_final.columns = ['_'.join(ele).upper() for ele in pos_cash_

        return pos_cash_aggregated_final

    def main(self):
        '''
        Function to be called for complete preprocessing and aggregation of POS_CASH_ba

        Inputs:
            self

        Returns:
            Final pre=processed and aggregated POS_CASH_balance table.
        '''

        #loading the dataframe
        self.load_dataframe()
        #performing the data pre-processing and feature engineering
        self.data_preprocessing_and_feature_engineering()
        #performing aggregations over SK_ID_PREV
        pos_cash_aggregated = self.aggregations_sk_id_prev()

        if self.verbose:
            print("\nAggregation over SK_ID_CURR...")
        #doing aggregations over each SK_ID_CURR
        pos_cash_aggregated_final = self.aggregations_sk_id_curr(pos_cash_aggregated)

        if self.verbose:
            print('\nDone preprocessing POS_CASH_balance.')
            print(f"\nInitial Size of POS_CASH_balance: {self.initial_size}")
            print(f'Size of POS_CASH_balance after Pre-Processing, Feature Engineering
            print(f'\nTotal Time Taken = {datetime.now() - self.start}')

        if self.dump_to_pickle:
```

```
            if self.verbose:
                print('\nPickling pre-processed POS_CASH_balance to POS_CASH_balance_pr
            with open(self.file_directory + 'POS_CASH_balance_preprocessed.pkl', 'wb')
                pickle.dump(pos_cash_aggregated_final, f)
            if self.verbose:
                print('Done.')
        if self.verbose:
            print('-'*100)

        return pos_cash_aggregated_final
```

In [29]:

```
pos_aggregated = preprocess_POS_CASH_balance(file_directory='./data/', dump_to_pickle=T
```

```
#########################################################
#          Pre-processing POS_CASH_balance.csv          #
#########################################################

Loading the DataFrame, POS_CASH_balance.csv, into memory...
Loaded POS_CASH_balance.csv
Time Taken to load = 0:00:03.647000

Starting Data Cleaning and Feature Engineering...
Done.
Time Taken = 0:12:36.713000

Aggregations over SK_ID_PREV...
Done.
Time Taken = 0:00:14.785999

Aggregation over SK_ID_CURR...

Done preprocessing POS_CASH_balance.

Initial Size of POS_CASH_balance: (10001358, 8)
Size of POS_CASH_balance after Pre-Processing, Feature Engineering and Aggregation: (337
252, 188)

Total Time Taken = 0:12:58.956000

Pickling pre-processed POS_CASH_balance to POS_CASH_balance_preprocessed.pkl
Done.
----------------------------------------------------------------------------------------
------------
```

In [ ]:

```
# temp = preprocess_POS_CASH_balance(file_directory='./data/', dump_to_pickle=True)
# #loading the dataframe
# temp.load_dataframe()
# #performing the data pre-processing and feature engineering
# temp.data_preprocessing_and_feature_engineering()
```

In [ ]:

```
# #performing aggregations over SK_ID_PREV
# pos_cash_aggregated = temp.aggregations_sk_id_prev()
```

In [ ]:

```
# print("\nAggregation over SK_ID_CURR...")
# #doing aggregations over each SK_ID_CURR
```

```
# pos_cash_aggregated_final = temp.aggregations_sk_id_curr(pos_cash_aggregated)
```

In [ ]:

```
# print('\nPickling pre-processed POS_CASH_balance to POS_CASH_balance_preprocessed.pkl
# with open('./data/POS_CASH_balance_preprocessed.pkl', 'wb') as f:
# pickle.dump(pos_cash_aggregated_final, f)
# if self.verbose:
# print('Done.')
```

2.1.5 credit_card_balance.csv

This table contains information about the previous credit cards that the client had with Home Credit Group.

1. We start off with removing an erroneous value, and then we proceed to feature engineering.
2. We create some domain based features such as total drawings, number of drawings, balance to limit ratio, payment done to minimum payment required difference, etc.
3. This table also contains all these data monthwise, so we calculate the EDAs for some of the features of this table too.
4. For aggregations, we first aggregate over SK_ID_PREV. Here we aggregate on three bases. Firstly, we do overall aggregations. We also do aggregations for last 2 years separately and the rest of the years. Finally we aggregate over SK_ID_PREV for categorical variable NAME_CONTRACT_TYPE.
5. For aggregation over SK_ID_CURR, we saw from the EDA that most of the current clients just had 1 credit card previously, so we do simple mean aggregations over SK_ID_CURR.

In [30]:

```
class preprocess_credit_card_balance:
    '''
    Preprocess the credit_card_balance table.
    Contains 5 member functions:
        1. init method
        2. load_dataframe method
        3. data_preprocessing_and_feature_engineering method
        4. aggregations method
        5. main method
    '''

    def __init__(self, file_directory = '', verbose = True, dump_to_pickle = False):
        '''
        This function is used to initialize the class members

        Inputs:
            self
            file_directory: Path, str, default = ''
                The path where the file exists. Include a '/' at the end of the path in
            verbose: bool, default = True
                Whether to enable verbosity or not
            dump_to_pickle: bool, default = False
                Whether to pickle the final preprocessed table or not

        Returns:
            None
        '''
```

```
        self.file_directory = file_directory
        self.verbose = verbose
        self.dump_to_pickle = dump_to_pickle

    def load_dataframe(self):
        '''
        Function to load the credit_card_balance.csv DataFrame.

        Inputs:
            self

        Returns:
            None
        '''

        if self.verbose:
            self.start = datetime.now()
            print('#########################################################')
            print('#        Pre-processing credit_card_balance.csv         #')
            print('#########################################################')
            print("\nLoading the DataFrame, credit_card_balance.csv, into memory...")

        self.cc_balance = pd.read_csv(self.file_directory + 'credit_card_balance.csv')
        self.initial_size = self.cc_balance.shape

        if self.verbose:
            print("Loaded credit_card_balance.csv")
            print(f"Time Taken to load = {datetime.now() - self.start}")

    def data_preprocessing_and_feature_engineering(self):
        '''
        Function to preprocess the table, by removing erroneous points, and then creati

        Inputs:
            self

        Returns:
            None
        '''

        if self.verbose:
            start = datetime.now()
            print("\nStarting Preprocessing and Feature Engineering...")

        #there is one abruptly large value for AMT_PAYMENT_CURRENT
        self.cc_balance['AMT_PAYMENT_CURRENT'][self.cc_balance['AMT_PAYMENT_CURRENT'] >
        #calculating the total missing values for each previous credit card
        self.cc_balance['MISSING_VALS_TOTAL_CC'] = self.cc_balance.isna().sum(axis = 1)
        #making the MONTHS_BALANCE Positive
        self.cc_balance['MONTHS_BALANCE'] = np.abs(self.cc_balance['MONTHS_BALANCE'])
        #sorting the DataFrame according to the month of status from oldest to latest, 
        self.cc_balance = self.cc_balance.sort_values(by = ['SK_ID_PREV','MONTHS_BALANC

        #Creating new features
        self.cc_balance['AMT_DRAWING_SUM'] = self.cc_balance['AMT_DRAWINGS_ATM_CURRENT'
                                    'AMT_DRAWINGS_OTHER_CURRENT'] + self.cc_balance['AM
        self.cc_balance['BALANCE_LIMIT_RATIO'] = self.cc_balance['AMT_BALANCE'] / (self
        self.cc_balance['CNT_DRAWING_SUM'] = self.cc_balance['CNT_DRAWINGS_ATM_CURRENT'
                                            'CNT_DRAWINGS_OTHER_CURRENT'] + self.cc_bal
```

```
        self.cc_balance['MIN_PAYMENT_RATIO'] = self.cc_balance['AMT_PAYMENT_CURRENT'] /
        self.cc_balance['PAYMENT_MIN_DIFF'] = self.cc_balance['AMT_PAYMENT_CURRENT'] -
        self.cc_balance['MIN_PAYMENT_TOTAL_RATIO'] = self.cc_balance['AMT_PAYMENT_TOTAL
        self.cc_balance['PAYMENT_MIN_DIFF'] = self.cc_balance['AMT_PAYMENT_TOTAL_CURREN
        self.cc_balance['AMT_INTEREST_RECEIVABLE'] = self.cc_balance['AMT_TOTAL_RECEIVA
        self.cc_balance['SK_DPD_RATIO'] = self.cc_balance['SK_DPD'] / (self.cc_balance[

        #calculating the rolling Exponential Weighted Moving Average over months for ce
        rolling_columns = [
            'AMT_BALANCE',
            'AMT_CREDIT_LIMIT_ACTUAL',
            'AMT_RECEIVABLE_PRINCIPAL',
            'AMT_RECIVABLE',
            'AMT_TOTAL_RECEIVABLE',
            'AMT_DRAWING_SUM',
            'BALANCE_LIMIT_RATIO',
            'CNT_DRAWING_SUM',
            'MIN_PAYMENT_RATIO',
            'PAYMENT_MIN_DIFF',
            'MIN_PAYMENT_TOTAL_RATIO',
            'AMT_INTEREST_RECEIVABLE',
            'SK_DPD_RATIO' ]
        exp_weighted_columns = ['EXP_' + ele for ele in rolling_columns]
        self.cc_balance[exp_weighted_columns] = self.cc_balance.groupby(['SK_ID_CURR','

        if self.verbose:
            print("Done.")
            print(f"Time Taken = {datetime.now() - start}")

    def aggregations(self):
        '''
        Function to perform aggregations of rows of credit_card_balance table, first ov
        and then over SK_ID_CURR

        Inputs:
            self

        Returns:
            aggregated credit_card_balance table.
        '''

        if self.verbose:
            print("\nAggregating the DataFrame, first over SK_ID_PREv, then over SK_ID_

        #performing aggregations over SK_ID_PREV
        overall_aggregations = {
            'SK_ID_CURR' : ['first'],
            'MONTHS_BALANCE': ['max'],
            'AMT_BALANCE' : ['sum','mean','max'],
            'AMT_CREDIT_LIMIT_ACTUAL' : ['sum','mean','max'],
            'AMT_DRAWINGS_ATM_CURRENT' : ['sum','max'],
            'AMT_DRAWINGS_CURRENT' : ['sum','max'],
            'AMT_DRAWINGS_OTHER_CURRENT' : ['sum','max'],
            'AMT_DRAWINGS_POS_CURRENT' : ['sum','max'],
            'AMT_INST_MIN_REGULARITY' : ['mean','min','max'],
            'AMT_PAYMENT_CURRENT' : ['mean','min','max'],
            'AMT_PAYMENT_TOTAL_CURRENT' : ['mean','min','max'],
            'AMT_RECEIVABLE_PRINCIPAL' : ['sum','mean','max'],
            'AMT_RECIVABLE' : ['sum','mean','max'],
            'AMT_TOTAL_RECEIVABLE' : ['sum','mean','max'],
```

```
            'CNT_DRAWINGS_ATM_CURRENT' : ['sum','max'],
            'CNT_DRAWINGS_CURRENT' : ['sum','max'],
            'CNT_DRAWINGS_OTHER_CURRENT' : ['sum','max'],
            'CNT_DRAWINGS_POS_CURRENT' : ['sum','max'],
            'CNT_INSTALMENT_MATURE_CUM' : ['sum','max','min'],
            'SK_DPD' : ['sum','max'],
            'SK_DPD_DEF' : ['sum','max'],

            'AMT_DRAWING_SUM' : ['sum','max'],
            'BALANCE_LIMIT_RATIO' : ['mean','max','min'],
            'CNT_DRAWING_SUM' : ['sum','max'],
            'MIN_PAYMENT_RATIO': ['min','mean'],
            'PAYMENT_MIN_DIFF' : ['min','mean'],
            'MIN_PAYMENT_TOTAL_RATIO' : ['min','mean'],
            'AMT_INTEREST_RECEIVABLE' : ['min','mean'],
            'SK_DPD_RATIO' : ['max','mean'],

            'EXP_AMT_BALANCE' : ['last'],
            'EXP_AMT_CREDIT_LIMIT_ACTUAL' : ['last'],
            'EXP_AMT_RECEIVABLE_PRINCIPAL' : ['last'],
            'EXP_AMT_RECIVABLE' : ['last'],
            'EXP_AMT_TOTAL_RECEIVABLE' : ['last'],
            'EXP_AMT_DRAWING_SUM' : ['last'],
            'EXP_BALANCE_LIMIT_RATIO' : ['last'],
            'EXP_CNT_DRAWING_SUM' : ['last'],
            'EXP_MIN_PAYMENT_RATIO' : ['last'],
            'EXP_PAYMENT_MIN_DIFF' : ['last'],
            'EXP_MIN_PAYMENT_TOTAL_RATIO' : ['last'],
            'EXP_AMT_INTEREST_RECEIVABLE' : ['last'],
            'EXP_SK_DPD_RATIO' : ['last'],
            'MISSING_VALS_TOTAL_CC' : ['sum']
        }
        aggregations_for_categories = {
            'SK_DPD' : ['sum','max'],
            'SK_DPD_DEF' : ['sum','max'],
            'BALANCE_LIMIT_RATIO' : ['mean','max','min'],
            'CNT_DRAWING_SUM' : ['sum','max'],
            'MIN_PAYMENT_RATIO': ['min','mean'],
            'PAYMENT_MIN_DIFF' : ['min','mean'],
            'MIN_PAYMENT_TOTAL_RATIO' : ['min','mean'],
            'AMT_INTEREST_RECEIVABLE' : ['min','mean'],
            'SK_DPD_RATIO' : ['max','mean'],
            'EXP_AMT_DRAWING_SUM' : ['last'],
            'EXP_BALANCE_LIMIT_RATIO' : ['last'],
            'EXP_CNT_DRAWING_SUM' : ['last'],
            'EXP_MIN_PAYMENT_RATIO' : ['last'],
            'EXP_PAYMENT_MIN_DIFF' : ['last'],
            'EXP_MIN_PAYMENT_TOTAL_RATIO' : ['last'],
            'EXP_AMT_INTEREST_RECEIVABLE' : ['last'],
            'EXP_SK_DPD_RATIO' : ['last']
        }
        aggregations_for_year = {
            'SK_DPD' : ['sum','max'],
            'SK_DPD_DEF' : ['sum','max'],
            'BALANCE_LIMIT_RATIO' : ['mean','max','min'],
            'CNT_DRAWING_SUM' : ['sum','max'],
            'MIN_PAYMENT_RATIO': ['min','mean'],
            'PAYMENT_MIN_DIFF' : ['min','mean'],
            'MIN_PAYMENT_TOTAL_RATIO' : ['min','mean'],
            'AMT_INTEREST_RECEIVABLE' : ['min','mean'],
```

```
        'SK_DPD_RATIO' : ['max','mean'],
        'EXP_AMT_DRAWING_SUM' : ['last'],
        'EXP_BALANCE_LIMIT_RATIO' : ['last'],
        'EXP_CNT_DRAWING_SUM' : ['last'],
        'EXP_MIN_PAYMENT_RATIO' : ['last'],
        'EXP_PAYMENT_MIN_DIFF' : ['last'],
        'EXP_MIN_PAYMENT_TOTAL_RATIO' : ['last'],
        'EXP_AMT_INTEREST_RECEIVABLE' : ['last'],
        'EXP_SK_DPD_RATIO' : ['last']
    }
    #performing overall aggregations over SK_ID_PREV for all features
    cc_balance_aggregated_overall = self.cc_balance.groupby('SK_ID_PREV').agg(overa
    cc_balance_aggregated_overall.columns = ['_'.join(ele).upper() for ele in cc_ba
    cc_balance_aggregated_overall.rename(columns = {'SK_ID_CURR_FIRST' : 'SK_ID_CUR

    #aggregating over SK_ID_PREV for different categories
    contract_status_categories = ['Active','Completed']
    cc_balance_aggregated_categories = pd.DataFrame()
    for i, contract_type in enumerate(contract_status_categories):
        group = self.cc_balance[self.cc_balance['NAME_CONTRACT_STATUS'] == contract_
        group.columns = ['_'.join(ele).upper() + '_' + contract_type.upper() for el
        if i == 0:
            cc_balance_aggregated_categories = group
        else:
            cc_balance_aggregated_categories = cc_balance_aggregated_categories.mer
    #aggregating over SK_ID_PREV for rest of the categories
    cc_balance_aggregated_categories_rest = self.cc_balance[(self.cc_balance['NAME_
                                                (self.cc_balance.NAME_CONTRACT_STATUS != 'Compl
    cc_balance_aggregated_categories_rest.columns = ['_'.join(ele).upper() + '_REST
    #merging all the categorical aggregations
    cc_balance_aggregated_categories = cc_balance_aggregated_categories.merge(cc_ba

    #aggregating over SK_ID_PREV for different years
    self.cc_balance['YEAR_BALANCE'] = self.cc_balance['MONTHS_BALANCE'] //12
    cc_balance_aggregated_year = pd.DataFrame()
    for year in range(2):
        group = self.cc_balance[self.cc_balance['YEAR_BALANCE'] == year].groupby('S
        group.columns = ['_'.join(ele).upper() + '_YEAR_' + str(year) for ele in gr
        if year == 0:
            cc_balance_aggregated_year = group
        else:
            cc_balance_aggregated_year = cc_balance_aggregated_year.merge(group, on
    #aggregating over SK_ID_PREV for rest of years
    cc_balance_aggregated_year_rest = self.cc_balance[self.cc_balance['YEAR_BALANCE
    cc_balance_aggregated_year_rest.columns = ['_'.join(ele).upper() + '_YEAR_REST'
    #merging all the yearwise aggregations
    cc_balance_aggregated_year = cc_balance_aggregated_year.merge(cc_balance_aggreg
    self.cc_balance = self.cc_balance.drop('YEAR_BALANCE', axis = 1)

    #merging all the aggregations
    cc_aggregated = cc_balance_aggregated_overall.merge(cc_balance_aggregated_categ
    cc_aggregated = cc_aggregated.merge(cc_balance_aggregated_year, on = 'SK_ID_PRE

    #one-hot encoding the categorical column NAME_CONTRACT_STATUS
    name_contract_dummies = pd.get_dummies(self.cc_balance.NAME_CONTRACT_STATUS, pr
    contract_names = name_contract_dummies.columns.tolist()
    #merging the one-hot encoded feature with original table
    self.cc_balance = pd.concat([self.cc_balance, name_contract_dummies], axis=1)
    #aggregating over SK_ID_PREV the one-hot encoded columns
    aggregated_cc_contract = self.cc_balance[['SK_ID_PREV'] + contract_names].group
```

```
        #merging with the aggregated table
        cc_aggregated = cc_aggregated.merge(aggregated_cc_contract, on = 'SK_ID_PREV',

        #now we will aggregate on SK_ID_CURR
        #As seen from EDA, since most of the SK_ID_CURR had only 1 credit card, so for
        cc_aggregated = cc_aggregated.groupby('SK_ID_CURR', as_index = False).mean()

        return cc_aggregated

    def main(self):
        '''
        Function to be called for complete preprocessing and aggregation of credit_card

        Inputs:
            self

        Returns:
            Final pre=processed and aggregated credit_card_balance table.
        '''

        #loading the dataframe
        self.load_dataframe()
        #preprocessing and performing Feature Engineering
        self.data_preprocessing_and_feature_engineering()
        #aggregating over SK_ID_PREV and SK_ID_CURR
        cc_aggregated = self.aggregations()

        if self.verbose:
            print('\nDone preprocessing credit_card_balance.')
            print(f"\nInitial Size of credit_card_balance: {self.initial_size}")
            print(f'Size of credit_card_balance after Pre-Processing, Feature Engineeri
            print(f'\nTotal Time Taken = {datetime.now() - self.start}')

        if self.dump_to_pickle:
            if self.verbose:
                print('\nPickling pre-processed credit_card_balance to credit_card_bala
            with open(self.file_directory + 'credit_card_balance_preprocessed.pkl', 'wb
                pickle.dump(cc_aggregated, f)
            if self.verbose:
                print('Done.')
        if self.verbose:
            print('-'*100)

        return cc_aggregated
```

In [31]:

```
cc_aggregated = preprocess_credit_card_balance(file_directory='./data/', dump_to_pickle
```

```
#########################################################
#         Pre-processing credit_card_balance.csv        #
#########################################################

Loading the DataFrame, credit_card_balance.csv, into memory...
Loaded credit_card_balance.csv
Time Taken to load = 0:00:04.500999

Starting Preprocessing and Feature Engineering...
Done.
```

```
Time Taken = 0:04:38.657000

Aggregating the DataFrame, first over SK_ID_PREv, then over SK_ID_CURR

Done preprocessing credit_card_balance.

Initial Size of credit_card_balance: (3840312, 23)
Size of credit_card_balance after Pre-Processing, Feature Engineering and Aggregation: (103558, 249)

Total Time Taken = 0:04:52.110999

Pickling pre-processed credit_card_balance to credit_card_balance_preprocessed.pkl
Done.
----------------------------------------------------------------------------------------------------
```

2.1.6 application_train and application_test

These tables consists of static data relating to the Borrowers. Each row represents one loan application.

1. First we start with cleaning the data by removing the erroneous datapoints. We also remove the rows in train data with categories such that those categories do not appear in test data. We also convert the Region Rating features to categorical becuase we saw from the EDA that they don't follow an ordinal beviour when it comes to Defaulting Characteristics.
2. Inspired from the winner's writeup for the problem, we also predict the missing values of EXT_SOURCE features by building a regression model on the rest of the numeric features.
3. Next we do feature engineering on the numeric features, and generate features based on Domain Knoweldge, such as INCOME TO ANNUITY ratio, EXT_SOURCE means, etc.
4. We also try to predict the interest rates by using the data from the previous applications features, and predicting using the data from application_train features. We also create a feature based on the Target values from application_train where we compute the mean of targets of 500 nearest neighbors of each row
5. Next we create some features based on the categorical interactions by grouping the data on several categorical combinations and imputing the aggregates for each group as features.
6. We encode the categorical features by response coding, as we didn't want to increase dimensionality by many-folds using OHE.

In [32]:

```
class preprocess_application_train_test:
    '''
    Preprocess the application_train and application_test tables.
    Contains 11 member functions:
        1. init method
        2. load_dataframe method
        3. data_cleaning method
        4. ext_source_values_predictor method
        5. numeric_feature_engineering method
        6. neighbors_EXT_SOURCE_feature method
        7. categorical_interaction_features method
        8. response_fit method
        9. response_transform method
```

```
        10. cnt_payment_prediction method
        11. main method
    '''

    def __init__(self, file_directory = '', verbose = True, dump_to_pickle = False):
        '''
        This function is used to initialize the class members

        Inputs:
            self
            file_directory: Path, str, default = ''
                The path where the file exists. Include a '/' at the end of the path in
            verbose: bool, default = True
                Whether to enable verbosity or not
            dump_to_pickle: bool, default = False
                Whether to pickle the final preprocessed table or not

        Returns:
            None
        '''

        self.verbose = verbose
        self.dump_to_pickle = dump_to_pickle
        self.file_directory = file_directory

    def load_dataframes(self):
        '''
        Function to load the application_train.csv and application_test.csv DataFrames.

        Inputs:
            self

        Returns:
            None
        '''

        if self.verbose:
            self.start = datetime.now()
            print('#######################################################')
            print('#        Pre-processing application_train.csv         #')
            print('#        Pre-processing application_test.csv          #')
            print('#######################################################')
            print("\nLoading the DataFrame, credit_card_balance.csv, into memory...")

        self.application_train = pd.read_csv(self.file_directory + 'application_train.c
        self.application_test = pd.read_csv(self.file_directory + 'application_test.csv
        self.initial_shape = self.application_train.shape

        if self.verbose:
            print("Loaded application_train.csv and application_test.csv")
            print(f"Time Taken to load = {datetime.now() - self.start}")

    def data_cleaning(self):
        '''
        Function to clean the tables, by removing erroneous rows/entries.

        Inputs:
            self

        Returns:
```

```
            None
        '''

        if self.verbose:
            print("\nPerforming Data Cleaning...")

        #there are some FLAG_DOCUMENT features having just one category for almost all
        flag_cols_to_drop = ['FLAG_DOCUMENT_2','FLAG_DOCUMENT_4','FLAG_DOCUMENT_10','FL
                            'FLAG_DOCUMENT_20']
        self.application_train = self.application_train.drop(flag_cols_to_drop, axis =
        self.application_test = self.application_test.drop(flag_cols_to_drop, axis = 1)
        #converting age from days to years
        self.application_train['DAYS_BIRTH'] = self.application_train['DAYS_BIRTH'] * -
        self.application_test['DAYS_BIRTH'] = self.application_test['DAYS_BIRTH'] * -1
        #From the EDA we saw some erroneous values in DAYS_EMPLOYED field
        self.application_train['DAYS_EMPLOYED'][self.application_train['DAYS_EMPLOYED']
        self.application_test['DAYS_EMPLOYED'][self.application_test['DAYS_EMPLOYED'] =
        #OBS Columns have an erroneous value, we'll remove those values
        self.application_train['OBS_30_CNT_SOCIAL_CIRCLE'][self.application_train['OBS_
        self.application_train['OBS_60_CNT_SOCIAL_CIRCLE'][self.application_train['OBS_
        self.application_test['OBS_30_CNT_SOCIAL_CIRCLE'][self.application_test['OBS_30
        self.application_test['OBS_60_CNT_SOCIAL_CIRCLE'][self.application_test['OBS_60
        #there were also 4 rows with 'XNA' as Gender, removing these rows
        self.application_train = self.application_train[self.application_train['CODE_GE
        #filling the categorical columns with 'XNA' value
        categorical_columns = self.application_train.dtypes[self.application_train.dtyp
        self.application_train[categorical_columns] = self.application_train[categorica
        self.application_test[categorical_columns] = self.application_test[categorical_
        #converting columns of REGION_RATING_CLIENT to object type, as we saw some comp
        self.application_train['REGION_RATING_CLIENT'] = self.application_train['REGION
        self.application_train['REGION_RATING_CLIENT_W_CITY'] = self.application_train[
        self.application_test['REGION_RATING_CLIENT'] = self.application_test['REGION_R
        self.application_test['REGION_RATING_CLIENT_W_CITY'] = self.application_test['R
        #counting the total NaN values for each application
        self.application_train['MISSING_VALS_TOTAL_APP'] = self.application_train.isna(
        self.application_test['MISSING_VALS_TOTAL_APP'] = self.application_test.isna().

        if self.verbose:
            print("Done.")

    def ext_source_values_predictor(self):
        '''
        Function to predict the missing values of EXT_SOURCE features

        Inputs:
            self

        Returns:
            None
        '''

        if self.verbose:
            start = datetime.now()
            print("\nPredicting the missing values of EXT_SOURCE columns...")

        #predicting the EXT_SOURCE missing values
        #using only numeric columns for predicting the EXT_SOURCES
        columns_for_modelling = list(set(self.application_test.dtypes[self.application_
                                     - set(['EXT_SOURCE_1','EXT_SOURCE_2','EXT_
        with open('./data/columns_for_ext_values_predictor.pkl', 'wb') as f:
```

```
            pickle.dump(columns_for_modelling, f)

        #we'll train an XGB Regression model for predicting missing EXT_SOURCE values
        #we will predict in the order of least number of missing value columns to max.
        for ext_col in ['EXT_SOURCE_2','EXT_SOURCE_3','EXT_SOURCE_1']:
            #X_model - datapoints which do not have missing values of given column
            #Y_train - values of column trying to predict with non missing values
            #X_train_missing - datapoints in application_train with missing values
            #X_test_missing - datapoints in application_test with missing values
            X_model, X_train_missing, X_test_missing, Y_train = self.application_train[
                                                                self.application_train[
                                                                self.application_test[s
                                                                self.application_train[
            xg = XGBRegressor(n_estimators = 1000, max_depth = 3, learning_rate = 0.1,
            xg.fit(X_model, Y_train)
            #dumping the model to pickle file
            with open(f'./data/nan_{ext_col}_xgbr_model.pkl', 'wb') as f:
                pickle.dump(xg, f)

            self.application_train[ext_col][self.application_train[ext_col].isna()] = x
            self.application_test[ext_col][self.application_test[ext_col].isna()] = xg.

            #adding the predicted column to columns for modelling for next column's pre
            columns_for_modelling = columns_for_modelling + [ext_col]

        if self.verbose:
            print("Done.")
            print(f"Time elapsed = {datetime.now() - start}")

    def numeric_feature_engineering(self, data):
        '''
        Function to perform feature engineering on numeric columns based on domain know

        Inputs:
            self
            data: DataFrame
                The tables of whose features are to be generated

        Returns:
            None
        '''

        #income and credit features
        data['CREDIT_INCOME_RATIO'] = data['AMT_CREDIT'] / (data['AMT_INCOME_TOTAL'] +
        data['CREDIT_ANNUITY_RATIO'] = data['AMT_CREDIT'] / (data['AMT_ANNUITY'] + 0.00
        data['ANNUITY_INCOME_RATIO'] = data['AMT_ANNUITY'] / (data['AMT_INCOME_TOTAL']
        data['INCOME_ANNUITY_DIFF'] = data['AMT_INCOME_TOTAL'] - data['AMT_ANNUITY']
        data['CREDIT_GOODS_RATIO'] = data['AMT_CREDIT'] / (data['AMT_GOODS_PRICE'] + 0.
        data['CREDIT_GOODS_DIFF'] = data['AMT_CREDIT'] - data['AMT_GOODS_PRICE'] + 0.00
        data['GOODS_INCOME_RATIO'] = data['AMT_GOODS_PRICE'] / (data['AMT_INCOME_TOTAL'
        data['INCOME_EXT_RATIO'] = data['AMT_INCOME_TOTAL'] / (data['EXT_SOURCE_3'] + 0
        data['CREDIT_EXT_RATIO'] = data['AMT_CREDIT'] / (data['EXT_SOURCE_3'] + 0.00001
        #age ratios and diffs
        data['AGE_EMPLOYED_DIFF'] = data['DAYS_BIRTH'] - data['DAYS_EMPLOYED']
        data['EMPLOYED_TO_AGE_RATIO'] = data['DAYS_EMPLOYED'] / (data['DAYS_BIRTH'] + 0
        #car ratios
        data['CAR_EMPLOYED_DIFF'] = data['OWN_CAR_AGE'] - data['DAYS_EMPLOYED']
        data['CAR_EMPLOYED_RATIO'] = data['OWN_CAR_AGE'] / (data['DAYS_EMPLOYED']+0.000
        data['CAR_AGE_DIFF'] = data['DAYS_BIRTH'] - data['OWN_CAR_AGE']
        data['CAR_AGE_RATIO'] = data['OWN_CAR_AGE'] / (data['DAYS_BIRTH'] + 0.00001)
```

```
#flag contacts sum
data['FLAG_CONTACTS_SUM'] = data['FLAG_MOBIL'] + data['FLAG_EMP_PHONE'] + data[
                            'FLAG_CONT_MOBILE'] + data['FLAG_PHONE'] + data['FL

data['HOUR_PROCESS_CREDIT_MUL'] = data['AMT_CREDIT'] * data['HOUR_APPR_PROCESS_
#family members
data['CNT_NON_CHILDREN'] = data['CNT_FAM_MEMBERS'] - data['CNT_CHILDREN']
data['CHILDREN_INCOME_RATIO'] = data['CNT_CHILDREN'] / (data['AMT_INCOME_TOTAL'
data['PER_CAPITA_INCOME'] = data['AMT_INCOME_TOTAL'] / (data['CNT_FAM_MEMBERS']
#region ratings
data['REGIONS_RATING_INCOME_MUL'] = (data['REGION_RATING_CLIENT'] + data['REGIO
data['REGION_RATING_MAX'] = [max(ele1, ele2) for ele1, ele2 in zip(data['REGION
data['REGION_RATING_MIN'] = [min(ele1, ele2) for ele1, ele2 in zip(data['REGION
data['REGION_RATING_MEAN'] = (data['REGION_RATING_CLIENT'] + data['REGION_RATIN
data['REGION_RATING_MUL'] = data['REGION_RATING_CLIENT'] * data['REGION_RATING_
#flag regions
data['FLAG_REGIONS'] = data['REG_REGION_NOT_LIVE_REGION'] + data['REG_REGION_NO
                       'REG_CITY_NOT_LIVE_CITY'] + data['REG_CITY_NOT_WORK_CIT
#ext_sources
data['EXT_SOURCE_MEAN'] = (data['EXT_SOURCE_1'] + data['EXT_SOURCE_2'] + data['
data['EXT_SOURCE_MUL'] = data['EXT_SOURCE_1'] * data['EXT_SOURCE_2'] * data['EX
data['EXT_SOURCE_MAX'] = [max(ele1,ele2,ele3) for ele1, ele2, ele3 in zip(data[
data['EXT_SOURCE_MIN'] = [min(ele1,ele2,ele3) for ele1, ele2, ele3 in zip(data[
data['EXT_SOURCE_VAR'] = [np.var([ele1,ele2,ele3]) for ele1, ele2, ele3 in zip(
data['WEIGHTED_EXT_SOURCE'] =  data.EXT_SOURCE_1 * 2 + data.EXT_SOURCE_2 * 3 +
#apartment scores
data['APARTMENTS_SUM_AVG'] = data['APARTMENTS_AVG'] + data['BASEMENTAREA_AVG']
                            'YEARS_BUILD_AVG'] + data['COMMONAREA_AVG'] + data[
                            'FLOORSMAX_AVG'] + data['FLOORSMIN_AVG'] + data['LA
                            'LIVINGAREA_AVG'] + data['NONLIVINGAPARTMENTS_AVG']

data['APARTMENTS_SUM_MODE'] = data['APARTMENTS_MODE'] + data['BASEMENTAREA_MODE
                            'YEARS_BUILD_MODE'] + data['COMMONAREA_MODE'] + dat
                            'FLOORSMAX_MODE'] + data['FLOORSMIN_MODE'] + data['
                            'LIVINGAREA_MODE'] + data['NONLIVINGAPARTMENTS_MODE

data['APARTMENTS_SUM_MEDI'] = data['APARTMENTS_MEDI'] + data['BASEMENTAREA_MEDI
                            'YEARS_BUILD_MEDI'] + data['COMMONAREA_MEDI'] + dat
                            'FLOORSMAX_MEDI'] + data['FLOORSMIN_MEDI'] + data['
                            'LIVINGAREA_MEDI'] + data['NONLIVINGAPARTMENTS_MEDI
data['INCOME_APARTMENT_AVG_MUL'] = data['APARTMENTS_SUM_AVG'] * data['AMT_INCOM
data['INCOME_APARTMENT_MODE_MUL'] = data['APARTMENTS_SUM_MODE'] * data['AMT_INC
data['INCOME_APARTMENT_MEDI_MUL'] = data['APARTMENTS_SUM_MEDI'] * data['AMT_INC
#OBS And DEF
data['OBS_30_60_SUM'] = data['OBS_30_CNT_SOCIAL_CIRCLE'] + data['OBS_60_CNT_SOC
data['DEF_30_60_SUM'] = data['DEF_30_CNT_SOCIAL_CIRCLE'] + data['DEF_60_CNT_SOC
data['OBS_DEF_30_MUL'] = data['OBS_30_CNT_SOCIAL_CIRCLE'] *  data['DEF_30_CNT_S
data['OBS_DEF_60_MUL'] = data['OBS_60_CNT_SOCIAL_CIRCLE'] *  data['DEF_60_CNT_S
data['SUM_OBS_DEF_ALL'] = data['OBS_30_CNT_SOCIAL_CIRCLE'] + data['DEF_30_CNT_S
                            'OBS_60_CNT_SOCIAL_CIRCLE'] + data['DEF_60_CNT_SOCI
data['OBS_30_CREDIT_RATIO'] = data['AMT_CREDIT'] / (data['OBS_30_CNT_SOCIAL_CIR
data['OBS_60_CREDIT_RATIO'] = data['AMT_CREDIT'] / (data['OBS_60_CNT_SOCIAL_CIR
data['DEF_30_CREDIT_RATIO'] = data['AMT_CREDIT'] / (data['DEF_30_CNT_SOCIAL_CIR
data['DEF_60_CREDIT_RATIO'] = data['AMT_CREDIT'] / (data['DEF_60_CNT_SOCIAL_CIR
#Flag Documents combined
data['SUM_FLAGS_DOCUMENTS'] = data['FLAG_DOCUMENT_3'] + data['FLAG_DOCUMENT_5']
                            'FLAG_DOCUMENT_7'] + data['FLAG_DOCUMENT_8'] + data
                            'FLAG_DOCUMENT_11'] + data['FLAG_DOCUMENT_13'] + da
                            'FLAG_DOCUMENT_15'] + data['FLAG_DOCUMENT_16'] + da
                            'FLAG_DOCUMENT_18'] + data['FLAG_DOCUMENT_19'] + da
```

```
        #details change
        data['DAYS_DETAILS_CHANGE_MUL'] = data['DAYS_LAST_PHONE_CHANGE'] * data['DAYS_R
        data['DAYS_DETAILS_CHANGE_SUM'] = data['DAYS_LAST_PHONE_CHANGE'] + data['DAYS_R
        #enquires
        data['AMT_ENQ_SUM'] = data['AMT_REQ_CREDIT_BUREAU_HOUR'] + data['AMT_REQ_CREDIT_
                            'AMT_REQ_CREDIT_BUREAU_MON'] + data['AMT_REQ_CREDIT_BUREAU_
        data['ENQ_CREDIT_RATIO'] = data['AMT_ENQ_SUM'] / (data['AMT_CREDIT'] + 0.00001)

        cnt_payment = self.cnt_payment_prediction(data)
        data['EXPECTED_CNT_PAYMENT'] = cnt_payment
        data['EXPECTED_INTEREST'] = data['AMT_ANNUITY'] *  data['EXPECTED_CNT_PAYMENT']
        data['EXPECTED_INTEREST_SHARE'] = data['EXPECTED_INTEREST'] / (data['AMT_CREDIT
        data['EXPECTED_INTEREST_RATE'] = 2 * 12 * data['EXPECTED_INTEREST'] / (data['AM

        return data

    def neighbors_EXT_SOURCE_feature(self):
        '''
        Function to generate a feature which contains the means of TARGET of 500 neighb

        Inputs:
            self

        Returns:
            None
        '''

        #https://www.kaggle.com/c/home-credit-default-risk/discussion/64821
        #imputing the mean of 500 nearest neighbor's target values for each application
        #neighbors are computed using EXT_SOURCE feature and CREDIT_ANNUITY_RATIO

        knn = KNeighborsClassifier(500, n_jobs = -1)

        train_data_for_neighbors = self.application_train[['EXT_SOURCE_1','EXT_SOURCE_2
        #saving the training data for neighbors
        with open('./data/TARGET_MEAN_500_Neighbors_training_data.pkl', 'wb') as f:
            pickle.dump(train_data_for_neighbors, f)
        train_target = self.application_train.TARGET
        test_data_for_neighbors = self.application_test[['EXT_SOURCE_1','EXT_SOURCE_2',

        knn.fit(train_data_for_neighbors, train_target)
        #pickling the knn model
        with open('./data/KNN_model_TARGET_500_neighbors.pkl', 'wb') as f:
            pickle.dump(knn, f)

        train_500_neighbors = knn.kneighbors(train_data_for_neighbors)[1]
        test_500_neighbors = knn.kneighbors(test_data_for_neighbors)[1]

        #adding the means of targets of 500 neighbors to new column
        self.application_train['TARGET_NEIGHBORS_500_MEAN'] = [self.application_train['
        self.application_test['TARGET_NEIGHBORS_500_MEAN'] = [self.application_train['T

    def categorical_interaction_features(self, train_data, test_data):
        '''
        Function to generate some features based on categorical groupings.

        Inputs:
            self
            train_data, test_data : DataFrames
                train and test dataframes
```

```
        Returns:
            Train and test datasets, with added categorical interaction features.
        '''

        #now we will create features based on categorical interactions
        columns_to_aggregate_on = [
            ['NAME_CONTRACT_TYPE', 'NAME_INCOME_TYPE', 'OCCUPATION_TYPE'],
            ['CODE_GENDER', 'NAME_FAMILY_STATUS', 'NAME_INCOME_TYPE'],
            ['FLAG_OWN_CAR', 'FLAG_OWN_REALTY', 'NAME_INCOME_TYPE'],
            ['NAME_EDUCATION_TYPE','NAME_INCOME_TYPE','OCCUPATION_TYPE'],
            ['OCCUPATION_TYPE','ORGANIZATION_TYPE'],
            ['CODE_GENDER','FLAG_OWN_CAR','FLAG_OWN_REALTY']

        ]
        aggregations = {
            'AMT_ANNUITY' : ['mean','max','min'],
            'ANNUITY_INCOME_RATIO' : ['mean','max','min'],
            'AGE_EMPLOYED_DIFF' : ['mean','min'],
            'AMT_INCOME_TOTAL' : ['mean','max','min'],
            'APARTMENTS_SUM_AVG' : ['mean','max','min'],
            'APARTMENTS_SUM_MEDI' : ['mean','max','min'],
            'EXT_SOURCE_MEAN' : ['mean','max','min'],
            'EXT_SOURCE_1' : ['mean','max','min'],
            'EXT_SOURCE_2' : ['mean','max','min'],
            'EXT_SOURCE_3' : ['mean','max','min']
        }

        #extracting values
        for group in columns_to_aggregate_on:
            #grouping based on categories
            grouped_interactions = train_data.groupby(group).agg(aggregations)
            grouped_interactions.columns = ['_'.join(ele).upper() + '_AGG_' + '_'.join(
            #saving the grouped interactions to pickle file
            group_name = '_'.join(group)
            with open(f'./data/Application_train_grouped_interactions_{group_name}.pkl'
                pickle.dump(grouped_interactions, f)
            #merging with the original data
            train_data = train_data.join(grouped_interactions, on = group)
            test_data = test_data.join(grouped_interactions, on = group)

        return train_data, test_data

    def response_fit(self, data, column):
        '''
        Response Encoding Fit Function
        Function to create a vocabulary with the probability of occurrence of each cate
        for a given class label.

        Inputs:
            self
            data: DataFrame
                training Dataset
            column: str
                the categorical column for which vocab is to be generated

        Returns:
            Dictionary of probability of occurrence of each category in a particular cl
        '''
```

```
        dict_occurrences = {1: {}, 0: {}}
        for label in [0,1]:
            dict_occurrences[label] = dict((data[column][data.TARGET == label].value_co

        return dict_occurrences

    def response_transform(self, data, column, dict_mapping):
        '''
        Response Encoding Transform Function
        Function to transform the categorical feature into two features, which contain
        of occurrence of that category for each class label.

        Inputs:
            self
            data: DataFrame
                DataFrame whose categorical features are to be encoded
            column: str
                categorical column whose encoding is to be done
            dict_mapping: dict
                Dictionary obtained from Response Fit function for that particular colu

        Returns:
            None
        '''

        data[column + '_0'] = data[column].map(dict_mapping[0])
        data[column + '_1'] = data[column].map(dict_mapping[1])

    def cnt_payment_prediction(self, data_to_predict):
        '''
        Function to predict the Count_payments on Current Loans using data from previou

        Inputs:
            self
            data_to_predict: DataFrame
                the values using which the model would predict the Count_payments on cu

        Returns:
            Predicted Count_payments of the current applications.
        '''

        #https://www.kaggle.com/c/home-credit-default-risk/discussion/64598
        previous_application = pd.read_csv('./data/previous_application.csv')
        train_data = previous_application[['AMT_CREDIT', 'AMT_ANNUITY', 'CNT_PAYMENT']]
        train_data['CREDIT_ANNUITY_RATIO'] = train_data['AMT_CREDIT'] / (train_data['AM
        #value to predict is our CNT_PAYMENT
        train_value = train_data.pop('CNT_PAYMENT')

        #test data would be our application_train data
        test_data = data_to_predict[['AMT_CREDIT','AMT_ANNUITY']].fillna(0)
        test_data['CREDIT_ANNUITY_RATIO'] = test_data['AMT_CREDIT'] / (test_data['AMT_A

        lgbmr = LGBMRegressor(max_depth = 9, n_estimators = 5000, n_jobs = -1, learning
                              random_state = 125)
        lgbmr.fit(train_data, train_value)
        #dumping the model to pickle file
        with open('./data/cnt_payment_predictor_lgbmr.pkl', 'wb') as f:
            pickle.dump(lgbmr, f)
        #predicting the CNT_PAYMENT for test_data
        cnt_payment = lgbmr.predict(test_data)
```

```
        return cnt_payment

    def main(self):
        '''
        Function to be called for complete preprocessing of application_train and appli

        Inputs:
            self

        Returns:
            Final pre=processed application_train and application_test tables.
        '''

        #loading the DataFrames first
        self.load_dataframes()
        #first doing Data Cleaning
        self.data_cleaning()
        #predicting the missing values of EXT_SOURCE columns
        self.ext_source_values_predictor()

        #doing the feature engineering
        if self.verbose:
            start = datetime.now()
            print("\nStarting Feature Engineering...")
            print("\nCreating Domain Based Features on Numeric Data")
        #Creating Numeric features based on domain knowledge
        self.application_train = self.numeric_feature_engineering(self.application_trai
        self.application_test = self.numeric_feature_engineering(self.application_test)
        #500 Neighbors Target mean
        self.neighbors_EXT_SOURCE_feature()
        if self.verbose:
            print("Done.")
            print(f"Time Taken = {datetime.now() - start}")

        if self.verbose:
            start = datetime.now()
            print("Creating features based on Categorical Interactions on some Numeric
        #creating features based on categorical interactions
        self.application_train, self.application_test = self.categorical_interaction_fe
        if self.verbose:
            print("Done.")
            print(f"Time taken = {datetime.now() - start}")

        #using response coding on categorical features, to keep the dimensionality in c
        #categorical columns to perform response coding on
        categorical_columns_application = self.application_train.dtypes[self.applicatio
        for col in categorical_columns_application:
            #extracting the dictionary with values corresponding to TARGET variable 0 a
            mapping_dictionary = self.response_fit(self.application_train, col)
            #saving the mapping dictionary to pickle file
            with open(f'./data/Response_coding_dict_{col}.pkl', 'wb') as f:
                pickle.dump(mapping_dictionary, f)
            #mapping this dictionary with our DataFrame
            self.response_transform(self.application_train, col, mapping_dictionary)
            self.response_transform(self.application_test, col, mapping_dictionary)
            #removing the original categorical columns
            _ = self.application_train.pop(col)
            _ = self.application_test.pop(col)
```

```
        if self.verbose:
            print('Done preprocessing appplication_train and application_test.')
            print(f"\nInitial Size of application_train: {self.initial_shape}")
            print(f'Size of application_train after Pre-Processing and Feature Engineer
            print(f'\nTotal Time Taken = {datetime.now() - self.start}')

        if self.dump_to_pickle:
            if self.verbose:
                print('\nPickling pre-processed application_train and application_test
            with open(self.file_directory + 'application_train_preprocessed.pkl', 'wb')
                pickle.dump(self.application_train, f)
            with open(self.file_directory + 'application_test_preprocessed.pkl', 'wb')
                pickle.dump(self.application_test, f)
            if self.verbose:
                print('Done.')
        if self.verbose:
            print('-'*100)

        return self.application_train, self.application_test
```

In [36]:

```
application_train, application_test = preprocess_application_train_test(file_directory=
```

```
#######################################################
#       Pre-processing application_train.csv          #
#       Pre-processing application_test.csv           #
#######################################################

Loading the DataFrame, credit_card_balance.csv, into memory...
Loaded application_train.csv and application_test.csv
Time Taken to load = 0:00:02.400999

Performing Data Cleaning...
Done.

Predicting the missing values of EXT_SOURCE columns...
Done.
Time elapsed = 0:03:22.624177

Starting Feature Engineering...

Creating Domain Based Features on Numeric Data
Done.
Time Taken = 0:02:20.478482
Creating features based on Categorical Interactions on some Numeric Features
Done.
Time taken = 0:00:02.156001
Done preprocessing appplication_train and application_test.

Initial Size of application_train: (307511, 122)
Size of application_train after Pre-Processing and Feature Engineering: (307507, 370)

Total Time Taken = 0:05:50.407627

Pickling pre-processed application_train and application_test to application_train_prepr
ocessed.pkl and application_test_preprocessed, respectively.
Done.
----------------------------------------------------------------------------------------
------------
```

2.2 Merging all tables

Now we will merge all the preprocessed tables with the application_train and application_test tables. The merges will be Left Outer Joins, such that all the current applications are preserved, as we have to model on them.

In [37]:

```
def merge_all_tables(application_train, application_test, bureau_aggregated, previous_a
                     installments_aggregated, pos_aggregated, cc_aggregated):
    '''
    Function to merge all the tables together with the application_train and applicatio
    on SK_ID_CURR.

    Inputs:
        All the previously pre-processed Tables.

    Returns:
        Single merged tables, one for training data and one for test data
    '''

    #merging application_train and application_test with Aggregated bureau table
    app_train_merged = application_train.merge(bureau_aggregated, on = 'SK_ID_CURR', ho
    app_test_merged = application_test.merge(bureau_aggregated, on = 'SK_ID_CURR', how
    #merging with aggregated previous_applications
    app_train_merged = app_train_merged.merge(previous_aggregated, on = 'SK_ID_CURR', h
    app_test_merged = app_test_merged.merge(previous_aggregated, on = 'SK_ID_CURR', how
    #merging with aggregated installments tables
    app_train_merged = app_train_merged.merge(installments_aggregated, on = 'SK_ID_CURR
    app_test_merged = app_test_merged.merge(installments_aggregated, on = 'SK_ID_CURR',
    #merging with aggregated POS_Cash balance table
    app_train_merged = app_train_merged.merge(pos_aggregated, on = 'SK_ID_CURR', how =
    app_test_merged = app_test_merged.merge(pos_aggregated, on = 'SK_ID_CURR', how = 'l
    #merging with aggregated credit card table
    app_train_merged = app_train_merged.merge(cc_aggregated, on = 'SK_ID_CURR', how = '
    app_test_merged = app_test_merged.merge(cc_aggregated, on = 'SK_ID_CURR', how = 'le

    return reduce_mem_usage(app_train_merged), reduce_mem_usage(app_test_merged)
```

In [38]:

```
train_data, test_data = merge_all_tables(application_train, application_test,
                                         bureau_aggregated, previous_aggregated,
                                         installments_aggregated, pos_aggregated,
                                         cc_aggregated)
```

```
--------------------------------------------------------------------------------------------
Memory usage of dataframe is 3634.096855163574 MB
Memory usage of after optimization is 1288.0046310424805 MB
Decreased by 64.55777921239509
--------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------
Memory usage of dataframe is 575.681396484375 MB
Memory usage of after optimization is 209.744384765625 MB
Decreased by 63.565891472868216
--------------------------------------------------------------------------------------------
```

3. Feature Engineering more

Features based on interaction among different tables

We will create some more features based on interactions between different tables. For example, we will calculate the Annuity to income ratio for previous applications, similarly we will calculate Credit to income ratios, and several such features.

In [39]:

```
def create_new_features(data):
    '''
    Function to create few more features after the merging of features, by using the
    interactions between various tables.

    Inputs:
        data: DataFrame

    Returns:
        None
    '''
    #previous applications columns
    prev_annuity_columns = [ele for ele in previous_aggregated.columns if 'AMT_ANNUITY'
    for col in prev_annuity_columns:
        data['PREV_' + col + '_INCOME_RATIO'] = data[col] / (data['AMT_INCOME_TOTAL'] +
    prev_goods_columns = [ele for ele in previous_aggregated.columns if 'AMT_GOODS' in
    for col in prev_goods_columns:
        data['PREV_' + col + '_INCOME_RATIO'] = data[col] / (data['AMT_INCOME_TOTAL'] +

    #credit_card_balance columns
    cc_amt_principal_cols = [ele for ele in cc_aggregated.columns if 'AMT_RECEIVABLE_PR
    for col in cc_amt_principal_cols:
        data['CC_' + col + '_INCOME_RATIO'] = data[col] / (data['AMT_INCOME_TOTAL'] + 0
    cc_amt_recivable_cols = [ele for ele in cc_aggregated.columns if 'AMT_RECIVABLE' in
    for col in cc_amt_recivable_cols:
        data['CC_' + col + '_INCOME_RATIO'] = data[col] / (data['AMT_INCOME_TOTAL'] + 0
    cc_amt_total_receivable_cols = [ele for ele in cc_aggregated.columns if 'TOTAL_RECE
    for col in cc_amt_total_receivable_cols:
        data['CC_' + col + '_INCOME_RATIO'] = data[col] / (data['AMT_INCOME_TOTAL'] + 0

    #installments_payments columns
    installments_payment_cols = [ele for ele in installments_aggregated.columns if 'AMT_
    for col in installments_payment_cols:
        data['INSTALLMENTS_' + col + '_INCOME_RATIO'] = data[col] / (data['AMT_INCOME_T
    #https://www.kaggle.com/c/home-credit-default-risk/discussion/64821
    installments_max_installment = ['AMT_INSTALMENT_MEAN_MAX', 'AMT_INSTALMENT_SUM_MAX'
    for col in installments_max_installment:
        data['INSTALLMENTS_ANNUITY_' + col + '_RATIO'] = data['AMT_ANNUITY'] / (data[co

    #POS_CASH_balance features have been created in its own dataframe itself

    #bureau and bureau_balance columns
    bureau_days_credit_cols = [ele for ele in bureau_aggregated.columns if 'DAYS_CREDIT
    for col in bureau_days_credit_cols:
        data['BUREAU_' + col + '_EMPLOYED_DIFF'] = data[col] - data['DAYS_EMPLOYED']
        data['BUREAU_' + col + '_REGISTRATION_DIFF'] = data[col] - data['DAYS_REGISTRAT
    bureau_overdue_cols = [ele for ele in bureau_aggregated.columns if 'AMT_CREDIT' in
    for col in bureau_overdue_cols:
        data['BUREAU_' + col + '_INCOME_RATIO'] = data[col] / (data['AMT_INCOME_TOTAL']
```

```
    bureau_amt_annuity_cols = [ele for ele in bureau_aggregated.columns if 'AMT_ANNUITY
    for col in bureau_amt_annuity_cols:
        data['BUREAU_' + col + '_INCOME_RATIO'] = data[col] / (data['AMT_INCOME_TOTAL']
```

In [40]:

```
create_new_features(train_data)
create_new_features(test_data)

print("After Pre-processing, aggregation, merging and Feature Engineering,")
print(f"Final Shape of Training Data = {train_data.shape}")
print(f"Final Shape of Test Data = {test_data.shape}")
```

```
After Pre-processing, aggregation, merging and Feature Engineering,
Final Shape of Training Data = (307507, 1634)
Final Shape of Test Data = (48744, 1633)
```

In [44]:

```
#freeing up the memory
del application_train, application_test, bureau_aggregated, previous_aggregated, instal
```

In [41]:

```
def final_pickle_dump(train_data, test_data, train_file_name, test_file_name, file_dire
    '''
    Function to dump the preprocessed files to pickle.

    Inputs:
        train_data: DataFrame
            Training Data
        test_data: DataFrame
            Test Data
        train_file_name: str
            Name of pickle file for training data
        test_file_name: str
            Name of pickle file for test data
        file_directory: str, default = ''
            Path of directory to save pickle file into
        verbose: bool, default = True
            Whether to keep verbosity or not

    Returns:
        None
    '''
    if verbose:
        print("Dumping the final preprocessed data to pickle files.")
        start = datetime.now()
    with open(file_directory + train_file_name + '.pkl','wb') as f:
        pickle.dump(train_data, f)
    with open(file_directory + test_file_name + '.pkl','wb') as f:
        pickle.dump(test_data,f)

    if verbose:
        print("Done.")
        print(f"Time elapsed = {datetime.now() - start}")

final_pickle_dump(train_data, test_data, 'train_data_final', 'test_data_final',file_dir
```

```
Dumping the final preprocessed data to pickle files.
Done.
```

```
Time elapsed = 0:00:01.396050
```

In [42]:

```
#removing the SK_ID_CURR from training and test data
train_data = train_data.drop(['SK_ID_CURR'], axis = 1)
skid_test = test_data.pop('SK_ID_CURR')
#extracting the class labels for training data
target_train = train_data.pop('TARGET')
```

4. Feature selection

In this section, we will try to reduce the number of features, in such a way that it doesn't have a negative impact on the performance of the model.

4.1 Looking for empty features

Here, empty features refer to those features which have just one unique value. These features are useless for the classifiers, as they do not contain any information.

In [43]:

```
empty_columns = []
for col in train_data.columns:
    if len(train_data[col].unique()) <=1:
        empty_columns.append(col)

print(f"There are {len(empty_columns)} columns with just 1 unique value")
print("Removing these from dataset")
train_data = train_data.drop(empty_columns, axis = 1)
test_data = test_data.drop(empty_columns, axis = 1)
```

```
There are 24 columns with just 1 unique value
Removing these from dataset
```

In [53]:

```
cl = train_data.columns
cl
```

Out[53]:

```
Index(['CNT_CHILDREN', 'AMT_INCOME_TOTAL', 'AMT_CREDIT', 'AMT_ANNUITY',
       'AMT_GOODS_PRICE', 'REGION_POPULATION_RELATIVE', 'DAYS_BIRTH',
       'DAYS_EMPLOYED', 'DAYS_REGISTRATION', 'DAYS_ID_PUBLISH',
       ...
       'BUREAU_AMT_CREDIT_SUM_OVERDUE_SUM_CREDITACTIVE_CLOSED_INCOME_RATIO',
       'BUREAU_AMT_CREDIT_MAX_OVERDUE_MAX_CREDITACTIVE_ACTIVE_INCOME_RATIO',
       'BUREAU_AMT_CREDIT_MAX_OVERDUE_SUM_CREDITACTIVE_ACTIVE_INCOME_RATIO',
       'BUREAU_AMT_CREDIT_SUM_OVERDUE_MAX_CREDITACTIVE_ACTIVE_INCOME_RATIO',
       'BUREAU_AMT_CREDIT_SUM_OVERDUE_SUM_CREDITACTIVE_ACTIVE_INCOME_RATIO',
       'BUREAU_AMT_CREDIT_MAX_OVERDUE_MAXCREDIT_ACTIVE_REST_INCOME_RATIO',
       'BUREAU_AMT_CREDIT_MAX_OVERDUE_SUMCREDIT_ACTIVE_REST_INCOME_RATIO',
       'BUREAU_AMT_CREDIT_SUM_OVERDUE_MAXCREDIT_ACTIVE_REST_INCOME_RATIO',
       'BUREAU_AMT_CREDIT_SUM_OVERDUE_SUMCREDIT_ACTIVE_REST_INCOME_RATIO',
       'BUREAU_AMT_ANNUITY_MEAN_OVERALL_INCOME_RATIO'],
      dtype='object', length=1608)
```

4.2 Recursive feature selection using LightGBM

In this section, we will further try to reduce the feature set, using a Classification Model, using the feature importance attribute . In this method, we will recursively run the Classification model on training dataset, and will check the Cross Validation AUC. If the Cross-Validation AUC goes below a certain threshold, we will stop adding the features.

The steps would be:

1. Run the classifier on whole training set, and calculate 3 fold cross-validation AUC.
2. Select the features which have non-zero feature importance as per the model.
3. Rerun the Classifier with the features which had zero feature importance. This is done because there might be cases where the classifier would have assigned 0-feature imporatnce to some features but that could be due to just that iteration and randomness. So we rerun the classifier on those features, to see if they alone can give good metric score.
4. Stop adding features if the Cross Validation score for low importance features goes below a threshold.

In [44]:

```
class recursive_feature_selector:
    '''
    Class to recursively select top features.
    Contains 2 methods:
        1. init method
        2. main method
    '''
    def __init__(self, train_data, test_data, target_train, num_folds=3, verbose=True,
        '''
        Function to initialize the class variables.

        Inputs:
            self
            train_data: DataFrame
                Training Data
            test_data: DataFrame
                Test Data
            target_train: Series
                Class Labels for training Data
            num_folds: int, default = 3
                Number of folds for K-Fold CV
            verbose: bool, default = True
                Whether to keep verbosity or not
            random_state: int, default = 5358
                The random state for the classifier for recursive feature selection

        Returns:
            None
        '''
        self.train_data = train_data
        self.test_data = test_data
        self.target_train = target_train
        self.num_folds = num_folds
        self.verbose = verbose
        self.random_state = random_state

    def main(self):
        '''
        Function to select features recursively
```

```
        Inputs:
            self

        Returns:
            Training and testing data with reduced number of features
        '''
        if self.verbose:
            print("Starting Feature Selection...")
            start = datetime.now()

        #set of importance features
        self.important_columns = set()
        score =1
        i = 1

        while score > 0.72:
            if self.verbose:
                print(f'Iteration {i}:')

            #removing the features which have been selected from the modelling data
            selection_data = self.train_data.drop(list(self.important_columns), axis= 1
            #defining the CV strategy
            fold = StratifiedKFold(n_splits = self.num_folds, shuffle = True, random_st
            #reinitializing the score
            score = 0
            model_feature_importance = np.zeros_like(selection_data.columns)

            #doing K-Fold cross validation
            for fold_num, (train_indices, val_indices) in enumerate(fold.split(selectio
                if self.verbose:
                    print(f"\t\tFitting fold {fold_num}")

                #defining the train and validation data
                x_train = selection_data.iloc[train_indices]
                x_val = selection_data.iloc[val_indices]
                y_train = self.target_train.iloc[train_indices]
                y_val = self.target_train.iloc[val_indices]

                #instantiating the LightGBM classifier
                lg = LGBMClassifier(n_jobs = -1, random_state = self.random_state)
                lg.fit(x_train, y_train)

                #appending the feature importance of each feature averaged over differe
                model_feature_importance += lg.feature_importances_ / self.num_folds
                #average k-fold ROC-AUC score
                score += roc_auc_score(y_val, lg.predict_proba(x_val)[:,1]) / self.num_

            #getting the non-zero feature importance columns
            imp_cols_indices = np.where(np.abs(model_feature_importance) > 0)
            #names of non-zero feature importance columns
            cols_imp = self.train_data.columns[imp_cols_indices]

            if score > 0.7:
                self.important_columns.update(cols_imp)
                if self.verbose:
                    print(f"\tNo. of important columns kept = {len(self.important_colum
            if self.verbose:
                print(f"\tCross Validation score = {score}")
            i += 1
```

```
        self.important_columns = list(self.important_columns)

        if self.verbose:
            print("\nDone Selecting Features.")
            print(f"Total columns removed = {self.train_data.shape[1] - len(self.import
            print(f"\nInitial Shape of train_data = {self.train_data.shape}")
        self.train_data = self.train_data[self.important_columns]
        self.test_data = self.test_data[self.important_columns]
        if self.verbose:
            print(f"Final Shape of train_data = {self.train_data.shape}")
            print(f"\nTotal Time Taken = {datetime.now() - start}")

        #saving the final columns into a pickle file
        with open('./data/final_cols.pkl', 'wb') as f:
            pickle.dump(train_data.columns.tolist(), f)

        gc.collect()

        return self.train_data, self.test_data
```

In [46]:

```
#instantiating the class recursive_feature_selector
feature_selector = recursive_feature_selector(train_data, test_data, target_train)
train_data, test_data = feature_selector.main()
important_columns = feature_selector.important_columns
```

```
Starting Feature Selection...
Iteration 1:
                Fitting fold 1
                Fitting fold 2
                Fitting fold 3
        No. of important columns kept = 1155
        Cross Validation score = 0.7973384417698235
Iteration 2:
                Fitting fold 1
                Fitting fold 2
                Fitting fold 3
        No. of important columns kept = 1242
        Cross Validation score = 0.715891922683983

Done Selecting Features.
Total columns removed = 366

Initial Shape of train_data = (307507, 1608)
Final Shape of train_data = (307507, 1242)

Total Time Taken = 0:02:31.495749
```

4.3 Saving Processed Data

In [47]:

```
with open('./data/pre_modelling_train.pkl','wb') as f:
        pickle.dump(train_data,f)
```

In [48]:

```
with open('./data/pre_modelling_test.pkl','wb') as f:
        pickle.dump(test_data,f)
```

In [49]:
```
with open('./data/pre_modelling_target_train.pkl','wb') as f:
        pickle.dump(target_train,f)
```

In [50]:
```
with open('./data/pre_modelling_skid_test.pkl','wb') as f:
        pickle.dump(skid_test,f)
```

In [51]:
```
with open('./data/pre_modelling_important_columns.pkl','wb') as f:
        pickle.dump(skid_test,f)
```

In [ ]:
```
# The above four files are saved for training models.
```

In [8]:
```
# train_data = pickle.load( open( "./data/pre_modelling_train.pkl", "rb" ) )
# test_data = pickle.load( open( "./data/pre_modelling_test.pkl", "rb" ) )
# target_train = pickle.load( open( "./data/pre_modelling_target_train.pkl", "rb" ) )
# skid_test = pickle.load( open( "./data/pre_modelling_skid_test.pkl", "rb" ) )
```