In [ ]:
```
######### PATR 0: Import important functions ##################################
# Pandas and numpy for data manipulation
import pandas as pd
import numpy as np

import functools
from scipy.stats import kurtosis, skew

#kurtosis_pearson = functools.partial(kurtosis, fisher=False)
#skew_p = functools.partial(skew)
#std_p = functools.partial(std)
# matplotlib and seaborn for plotting
import matplotlib.pyplot as plt
import seaborn as sns

# Suppress warnings from pandas
import warnings
warnings.filterwarnings('ignore')
plt.style.use('fivethirtyeight')


import matplotlib.pyplot as plt
import lightgbm as lgb

from sklearn.model_selection import KFold
from sklearn.metrics import roc_auc_score
from sklearn.preprocessing import LabelEncoder

# Memory management
import gc

def split_train_test(data, test_ratio):
    # data = housing
    np.random.seed(42)
    shuffled_indices = np.random.permutation(len(data))
    test_set_size = int(len(data)*test_ratio)
    test_indices = shuffled_indices[:test_set_size]
    train_indices = shuffled_indices[test_set_size:]
    return data.iloc[train_indices], data.iloc[test_indices]

def remove_missing_columns(train, test, threshold = 99):
```

```
    # Calculate missing stats for train and test (remember to calculate a percent!)
    train_miss = pd.DataFrame(train.isnull().sum())
    train_miss['percent'] = 100 * train_miss[0] / len(train)

    test_miss = pd.DataFrame(test.isnull().sum())
    test_miss['percent'] = 100 * test_miss[0] / len(test)

    # list of missing columns for train and test
    missing_train_columns = list(train_miss.index[train_miss['percent'] > threshold])
    missing_test_columns = list(test_miss.index[test_miss['percent'] > threshold])

    # Combine the two lists together
    missing_columns = list(set(missing_train_columns + missing_test_columns))

    # Print information
    print('There are %d columns with greater than %d%% missing values.' % (len(missing_columns), threshold))

    # Drop the missing columns and return
    train = train.drop(columns = missing_columns)
    test = test.drop(columns = missing_columns)

    return train, test
```

In [ ]:
```
######### PATR 1: Features Engineering ##################################

# -*- coding: utf-8 -*-
"""
Created on Fri Sep  6 01:35:32 2019

@author: Phong
"""

####. 1_Function to Aggregate Numeric Data
def agg_numeric(df, parent_var, df_name):
    """
    Groups and aggregates the numeric values in a child dataframe
    by the parent variable.

    Parameters
    --------
        df (dataframe):
            the child dataframe to calculate the statistics on
        parent_var (string):
            the parent variable used for grouping and aggregating
        df_name (string):
            the variable used to rename the columns

    Return
    --------
        agg (dataframe):
            a dataframe with the statistics aggregated by the `parent_var` for
            all numeric columns. Each observation of the parent variable will have
            one row in the dataframe with the parent variable as the index.
            The columns are also renamed using the `df_name`. Columns with all duplicate
            values are removed.

    """

    # Remove id variables other than grouping variable
    for col in df:
        if col != parent_var and 'SK_ID' in col:
            df = df.drop(columns = col)

    # Only want the numeric variables
```

```
    parent_ids = df[parent_var].copy()
    numeric_df = df.select_dtypes('number').copy()
    numeric_df[parent_var] = parent_ids

    # Group by the specified variable and calculate the statistics
    agg = numeric_df.groupby(parent_var).agg(['count', 'mean', 'max', 'min', 'sum','var','std', 'skew'])#]) # With 'std':

    # Need to create new column names
    columns = []

    # Iterate through the variables names
    for var in agg.columns.levels[0]:
        if var != parent_var:
            # Iterate through the stat names
            for stat in agg.columns.levels[1]:
                # Make a new column name for the variable and stat
                columns.append('%s_%s_%s' % (df_name, var, stat))

    agg.columns = columns

    # Remove the columns with all redundant values
    _, idx = np.unique(agg, axis = 1, return_index=True)
    agg = agg.iloc[:, idx]

    return agg


#### 2. Function to calculate categorical counts
# normed count, which is the count for a category divided by the total counts for all categories in a categorical variabl
# counts: the occurrences  of each category in a categorical variable

def agg_categorical(df, parent_var, df_name):
    """
    Aggregates the categorical features in a child dataframe
    for each observation of the parent variable.

    Parameters
    --------
    df : dataframe
        The dataframe to calculate the value counts for.

    parent_var : string
```

```
        The variable by which to group and aggregate the dataframe. For each unique
        value of this variable, the final dataframe will have one row

    df_name : string
        Variable added to the front of column names to keep track of columns


    Return
    --------
    categorical : dataframe
        A dataframe with aggregated statistics for each observation of the parent_var
        The columns are also renamed and columns with duplicate values are removed.

    """

    # Select the categorical columns
    categorical = pd.get_dummies(df.select_dtypes('category'))

    # Make sure to put the identifying id on the column
    categorical[parent_var] = df[parent_var]

    # Groupby the group var and calculate the sum and mean
    categorical = categorical.groupby(parent_var).agg(['sum', 'count','mean'])

    column_names = []

    # Iterate through the columns in level 0
    for var in categorical.columns.levels[0]:
        # Iterate through the stats in level 1
        for stat in ['sum', 'count', 'mean']:
            # Make a new column name
            column_names.append('%s_%s_%s' % (df_name, var, stat))


    categorical.columns = column_names

    # Remove duplicate columns by values
    _, idx = np.unique(categorical, axis = 1, return_index = True)
    categorical = categorical.iloc[:, idx]

    return categorical
```

```
    # add1 = agg_categorical(bureau1 , parent_var = 'SK_ID_CURR', df_name = 'app')

# Function to Aggregate Stats at the Client Level

def aggregate_client(df, group_vars, df_names):
    """Aggregate a dataframe with data at the loan level
    at the client level

    Args:
        df (dataframe): data at the loan level
        group_vars (list of two strings): grouping variables for the loan
        and then the client (example ['SK_ID_PREV', 'SK_ID_CURR'])
        names (list of two strings): names to call the resulting columns
        (example ['cash', 'client'])

    Returns:
        df_client (dataframe): aggregated numeric stats at the client level.
        Each client will have a single row with all the numeric data aggregated
    """
    # Aggregate the numeric columns
    df_agg = agg_numeric(df, parent_var = group_vars[0], df_name=df_names[0])

    # Handle categorical variables
    if any(df.dtypes == 'category'):
        df_counts = agg_categorical(df, parent_var = group_vars[0], df_name = df_names[0])

        # Merge 2 dfs:
        df_by_loan1 = df_counts.merge(df_agg, on = group_vars[0], how = 'outer')
        gc.enable()
        del df_agg, df_counts
        gc.collect()

        # # Merge to get the client id in dataframe

        df_by_loan1 = df_by_loan1.merge(df[[group_vars[0], group_vars[1]]], on = group_vars[0], how = 'left')

        # remove the loan id

        df_by_loan1 = df_by_loan1.drop(columns= [group_vars[0]])
        # Aggregate numeric stats by column
        df_by_client = agg_numeric(df_by_loan1, parent_var = group_vars[1], df_name = df_names[1])
    # No categorical variables
```

```
    else:
        df_by_loan1 = df_agg.merge(df[[group_vars[0], group_vars[1]]], on = group_vars[0], how ='left')
        gc.enable()
        del df_agg
        gc.collect()

        # Remove the loan id
        df_by_loan1 = df_by_loan1.drop(columns = [group_vars[0]])
        # Aggregate numeric stats by column
        df_by_client = agg_numeric(df_by_loan1, parent_var = group_vars[1], df_name = df_names[1])

    # Memory management
    gc.enable()
    del df, df_by_loan1
    gc.collect()

    return df_by_client



# Function to Convert Data Types
# This will help reduce memory usage by using more efficient types for the variables: or example category is often a bett
import sys

def return_size(df):
    """Return size of dataframe in gigabytes"""
    return round(sys.getsizeof(df) / 1e9, 2)

def convert_types(df, print_info = False):

    original_memory = df.memory_usage().sum()

    # Iterate through each column
    for c in df:

        # Convert ids and booleans to integers
        if ('SK_ID' in c):
            df[c] = df[c].fillna(0).astype(np.int32)

        # Convert objects to category
        elif (df[c].dtype == 'object') and (df[c].nunique() < df.shape[0]):
            df[c] = df[c].astype('category')
```

```
        # Booleans mapped to integers
        elif list(df[c].unique()) == [1, 0]:
            df[c] = df[c].astype(bool)

        # Float64 to float32
        elif df[c].dtype == float:
            df[c] = df[c].astype(np.float32)

        # Int64 to int32
        elif df[c].dtype == int:
            df[c] = df[c].astype(np.int32)

    new_memory = df.memory_usage().sum()

    if print_info:
        print(f'Original Memory Usage: {round(original_memory / 1e9, 2)} gb.')
        print(f'New Memory Usage: {round(new_memory / 1e9, 2)} gb.')

    return df




##############################################################################################################################
######################## START FEATURES ENGINEERING HERE #####################################################################
##############################################################################################################################


##################################### 0. Application train Data set ##########################################################


app_train = pd.read_csv('application_train.csv')
############### Create Domain Knowledge features
# CREDIT_INCOME_PERCENT: the percentage of the credit amount relative to a client's income
# ANNUITY_INCOME_PERCENT: the percentage of the loan annuity relative to a client's income
# CREDIT_TERM: the length of the payment in months (since the annuity is the monthly amount due
# DAYS_EMPLOYED_PERCENT: the percentage of the days employed relative to the client's age

app_train['CREDIT_INCOME_PERCENT'] = app_train['AMT_CREDIT']/app_train['AMT_INCOME_TOTAL']
app_train['ANNUITY_INCOME_PERCENT'] = app_train['AMT_ANNUITY']/app_train['AMT_INCOME_TOTAL']
```

```
app_train['CREDIT_TERM'] = app_train['AMT_ANNUITY']/app_train['AMT_CREDIT']
app_train['DAYS_EMPLOYED_PERCENT'] = app_train['DAYS_EMPLOYED']/app_train['DAYS_BIRTH']
app_train['INCOME_PER_PERSON'] = app_train['AMT_INCOME_TOTAL'] / app_train['CNT_FAM_MEMBERS']
app_train['PAYMENT_RATE'] = app_train['AMT_ANNUITY'] / app_train['AMT_CREDIT']

train_set, test_set = split_train_test(train_set, 0.2)

train_set.to_csv('train_set.csv', index = False)
test_set.to_csv('test_set.csv', index = False)



#########################################################################################################################
######################## STAGE 1: Features from application_train, bureau and bureau_balance ###########################
########### 1_ Bureau Data Set ##########################################################################################
train = pd.read_csv('train_set.csv')
test = pd.read_csv('test_set.csv')
bureau = pd.read_csv('bureau.csv')#.head(50000)
bureau = convert_types(bureau, print_info = True)
bureau.info()

previous_loan_counts = bureau.groupby('SK_ID_CURR', as_index = False)['SK_ID_BUREAU'].count().rename(columns ={'SK_ID_BUR
previous_loan_counts.head()
train = train.merge(previous_loan_counts, on = 'SK_ID_CURR', how = 'left')
train['previous_loan_counts'] = train['previous_loan_counts'].fillna(0)
train.info()

test = test.merge(previous_loan_counts, on = 'SK_ID_CURR', how = 'left')
test['previous_loan_counts'] = test['previous_loan_counts'].fillna(0)
test.info()

bureau['CREDIT_DAY_OVERDUE_TIME_DAYS_CREDIT'] = bureau['CREDIT_DAY_OVERDUE'] * bureau['DAYS_CREDIT']
bureau_by_client = aggregate_client(bureau, group_vars =['SK_ID_BUREAU', 'SK_ID_CURR'], df_names = ['bureau', 'client'])

list(bureau_by_client.columns)

train=train.merge(bureau_by_client, on = 'SK_ID_CURR', how = 'left' )

test = test.merge(bureau_by_client, on = 'SK_ID_CURR', how = 'left' )

gc.enable()
del bureau , bureau_by_client
```

```
gc.collect()
train, test = remove_missing_columns(train, test)
train.info()

########### 2_ Bureau_balance Data Set ###########################################################################
bureau_balance = pd.read_csv('bureau_balance.csv')
bureau_balance.head()
bureau = pd.read_csv('bureau.csv')[['SK_ID_BUREAU', 'SK_ID_CURR']]
bureau_balance = bureau_balance.merge(bureau, on ='SK_ID_BUREAU', how = 'left')

bureau_balance = convert_types(bureau_balance, print_info = True)
bureau_balance.info()

bureau_balance_by_client = aggregate_client(bureau_balance, group_vars =['SK_ID_BUREAU', 'SK_ID_CURR'], df_names = ['bure

bureau_balance_by_client.head()

train=train.merge(bureau_balance_by_client, on = 'SK_ID_CURR', how = 'left')

test = test.merge(bureau_balance_by_client, on = 'SK_ID_CURR', how = 'left')

gc.enable()
del bureau_balance_by_client, bureau_balance, bureau
gc.collect()
train, test = remove_missing_columns(train, test)
train.info()

train.to_csv('train_after_stage1.csv', index = False)
test.to_csv('test_after_stage1.csv', index = False)

train.info()
'TARGET' in list(train.columns)
'TARGET' in list(test.columns)
set(list(train.columns)) - set(list(test.columns))
test.info()

##################################### STAGE 2 ####################################################################
################ All data except for ata from Installment Payments ###############################################
########### 3_ previous_application Data Set #####################################################################

previous=pd.read_csv('previous_application.csv')
previous = convert_types(previous, print_info=True)
```

```
previous.head()

previous_agg = agg_numeric(previous, 'SK_ID_CURR', 'previous')
previous_agg.shape # 37 columns -> 70 columns


previous_counts = agg_categorical(previous, 'SK_ID_CURR', 'previous')
previous_counts.shape # 37 columns -> 285 columns
list(previous_counts.columns)

# train = pd.read_csv('train_after_stage1.csv')
train = convert_types(train)

# test =pd.read_csv('test_after_stage1.csv')

test.info()
test = convert_types(test)

# Merge new features into train and test
train = train.merge(previous_counts, on ='SK_ID_CURR', how = 'left')
train = train.merge(previous_agg, on = 'SK_ID_CURR', how = 'left')

test = test.merge(previous_counts, on ='SK_ID_CURR', how = 'left')
test = test.merge(previous_agg, on = 'SK_ID_CURR', how = 'left')

# Remove variables to free memory
gc.enable()
del previous, previous_agg, previous_counts
gc.collect()

train, test = remove_missing_columns(train, test)

########### 4_ Monthly Cash Data Set##########################################################################################
cash = pd.read_csv('POS_CASH_balance.csv')
cash = convert_types(cash, print_info = True)
cash.head()
cash.info()

cash_by_client = aggregate_client(cash, group_vars =['SK_ID_PREV', 'SK_ID_CURR'], df_names =['cash', 'client'])
cash_by_client.info()
cash_by_client.head()
```

```
print('Cash by client Shape: ', cash_by_client.shape)
train = train.merge(cash_by_client, on ='SK_ID_CURR', how ='left')
test = test.merge(cash_by_client, on = 'SK_ID_CURR', how ='left')

gc.enable()
del cash, cash_by_client
gc.collect()

train, test = remove_missing_columns(train, test)

########### 5_ Monthly Credit Data Set ####################################################################################

credit = pd.read_csv('credit_card_balance.csv')
credit = convert_types(credit, print_info = True)
credit.info()
credit.head()

credit_by_client = aggregate_client(credit, group_vars=['SK_ID_PREV', 'SK_ID_CURR'], df_names=['credit','client'])
credit_by_client.head()

train = train.merge(credit_by_client, on = 'SK_ID_CURR', how ='left')
test = test.merge(credit_by_client, on = 'SK_ID_CURR', how = 'left')

gc.enable()
del credit, credit_by_client
gc.collect()
train, test = remove_missing_columns(train, test)
##########################################################################################################################
########################################## STAGE 3: include data from Installment Payments ################################
########### 6_Installment Payments Data Set ##############################################################################

installments =pd.read_csv('installments_payments.csv')
installments = convert_types(installments, print_info = True)
installments.info()
installments.head()


installments_by_client = aggregate_client(installments, group_vars =['SK_ID_PREV', 'SK_ID_CURR'], df_names = ['installmen

installments_by_client.head()

train=train.merge(installments_by_client, on = 'SK_ID_CURR', how = 'left' )
```

```
test = test.merge(installments_by_client, on = 'SK_ID_CURR', how = 'left' )

gc.enable()
del installments, installments_by_client
gc.collect()
train, test = remove_missing_columns(train, test)
train.info()
test.info()

test['TARGET'] = test_labels

print(f'Final training size: {return_size(train)}')
print(f'Final testing size: {return_size(test)}')

train.to_csv('train_after_stage3.csv', index = False)
test.to_csv('test_after_stage3.csv', index = False)
set(list(train.columns)) - set(list(test.columns))


################################# STAGE 4 ########################################################################
############################### Create more polinomial features for data after stage 3 ####################################
################################## ##########################################################################
train = pd.read_csv('train_after_stage3.csv')
app_train = train # convert_types(train)

test =pd.read_csv('test_after_stage3.csv')

app_test = test#convert_types(test)


######################## Create the Polynomial Features from most important 8 feactors:



poly_features = app_train[['EXT_SOURCE_1', 'EXT_SOURCE_2', 'EXT_SOURCE_3', 'DAYS_BIRTH',
                           'AMT_CREDIT','AMT_ANNUITY', 'AMT_ANNUITY', 'AMT_GOODS_PRICE', 'DAYS_EMPLOYED']]
poly_features_test = app_test[['EXT_SOURCE_1', 'EXT_SOURCE_2', 'EXT_SOURCE_3', 'DAYS_BIRTH',
                           'AMT_CREDIT','AMT_ANNUITY', 'AMT_ANNUITY', 'AMT_GOODS_PRICE', 'DAYS_EMPLOYED']]


# Imputer for handling the missing values
```

```
from sklearn.preprocessing import Imputer
imputer = Imputer(strategy = 'median')
poly_target = app_train['TARGET']

poly_features = imputer.fit_transform(poly_features)
poly_features_test = imputer.transform(poly_features_test)

# Create the polynomial object with specified degree: add 212 features
from sklearn.preprocessing import PolynomialFeatures
poly_transformer = PolynomialFeatures(degree = 3)
poly_transformer.fit(poly_features)
poly_features = poly_transformer.transform(poly_features)
poly_features_test = poly_transformer.transform(poly_features_test)
print('Polynomial Features shape: ', poly_features.shape, poly_features_test.shape)
column = poly_transformer.get_feature_names(input_features =['EXT_SOURCE_1', 'EXT_SOURCE_2', 'EXT_SOURCE_3', 'DAYS_BIRTH'
                           'AMT_CREDIT','AMT_ANNUITY', 'AMT_ANNUITY', 'AMT_GOODS_PRICE', 'DAYS_EMPLOYED'])

# Create a dataframe of the features
poly_features = pd.DataFrame(poly_features, columns = column)
poly_features['SK_ID_CURR'] = app_train['SK_ID_CURR']
poly_features_test = pd.DataFrame(poly_features_test, columns = column)
poly_features_test['SK_ID_CURR'] = app_test['SK_ID_CURR']
poly_features_test.shape
poly_features.shape
# Merger to app_train and app_test:
app_train = app_train.merge(poly_features, on = 'SK_ID_CURR', how = 'left')
app_train.shape
app_test = app_test.merge(poly_features_test, on = 'SK_ID_CURR', how ='left')
app_test.shape



app_train.to_csv('train_after_stage4.csv', index = False)
app_test.to_csv('test_after_stage4.csv', index = False)

########################################################################################################################
############### ROUND 2: Create More Knowledge Features and Time Features ###############################################
########################################################################################################################

############# 0.Application_train Data Set #############################################################################

import pandas as pd
```

```
#df = pd.read_csv('application_train.csv')
df.head()
df = train
df['DAYS_EMPLOYED'].replace(365243, np.nan, inplace= True)

# Time features;
df['train_NEW_EMPLOY_TO_BIRTH_RATIO'] = df['DAYS_EMPLOYED'] / df['DAYS_BIRTH']
df['train_DAYS_EMPLOYED - DAYS_BIRTH'] = df['DAYS_EMPLOYED'] - df['DAYS_BIRTH']
df['train_NEW_CAR_TO_BIRTH_RATIO'] = df['OWN_CAR_AGE'] / df['DAYS_BIRTH']
df['train_NEW_CAR_TO_EMPLOY_RATIO'] = df['OWN_CAR_AGE'] / df['DAYS_EMPLOYED']
df['train_NEW_PHONE_TO_BIRTH_RATIO'] = df['DAYS_LAST_PHONE_CHANGE'] / df['DAYS_BIRTH']
df['train_NEW_PHONE_TO_EMPLOY_RATIO'] = df['DAYS_LAST_PHONE_CHANGE'] / df['DAYS_EMPLOYED']
df['train_EXT_SOURCE_1 / DAYS_BIRTH'] = df['EXT_SOURCE_1'] / df['DAYS_BIRTH']
df['train_EXT_SOURCE_2 / DAYS_BIRTH'] = df['EXT_SOURCE_2'] / df['DAYS_BIRTH']
df['train_EXT_SOURCE_3 / DAYS_BIRTH'] = df['EXT_SOURCE_3'] / df['DAYS_BIRTH']
# Knowledge features:
df['train_NEW_CREDIT_TO_ANNUITY_RATIO'] = df['AMT_CREDIT'] / df['AMT_ANNUITY']
df['train_NEW_CREDIT_TO_GOODS_RATIO'] = df['AMT_CREDIT'] / df['AMT_GOODS_PRICE']
df['train_NEW_INC_PER_CHLD'] = df['AMT_INCOME_TOTAL'] / (1 + df['CNT_CHILDREN'])
df['train_NEW_ANNUITY_TO_INCOME_RATIO'] = df['AMT_ANNUITY'] / (1 + df['AMT_INCOME_TOTAL'])
df['train_NEW_SOURCES_PROD'] = df['EXT_SOURCE_1'] * df['EXT_SOURCE_2'] * df['EXT_SOURCE_3']
df['train_NEW_EXT_SOURCES_MEAN'] = df[['EXT_SOURCE_1', 'EXT_SOURCE_2', 'EXT_SOURCE_3']].mean(axis=1)
df['train_NEW_SCORES_STD'] = df[['EXT_SOURCE_1', 'EXT_SOURCE_2', 'EXT_SOURCE_3']].std(axis=1)
df['train_NEW_SCORES_STD'] = df['NEW_SCORES_STD'].fillna(df['NEW_SCORES_STD'].mean())
df['train_NEW_CREDIT_TO_INCOME_RATIO'] = df['AMT_CREDIT'] / df['AMT_INCOME_TOTAL']
dropcolum=['FLAG_DOCUMENT_2','FLAG_DOCUMENT_4',
    'FLAG_DOCUMENT_5','FLAG_DOCUMENT_6','FLAG_DOCUMENT_7',
    'FLAG_DOCUMENT_8','FLAG_DOCUMENT_9','FLAG_DOCUMENT_10',
    'FLAG_DOCUMENT_11','FLAG_DOCUMENT_12','FLAG_DOCUMENT_13',
    'FLAG_DOCUMENT_14','FLAG_DOCUMENT_15','FLAG_DOCUMENT_16',
    'FLAG_DOCUMENT_17','FLAG_DOCUMENT_18','FLAG_DOCUMENT_19',
    'FLAG_DOCUMENT_20','FLAG_DOCUMENT_21']
df= df.drop(dropcolum,axis=1)

train = df

######## 1. bureau ##########################################################################
# bureau: information about client's previous loans with other financial institutions reported to Home Credit. Each previ
# bureau_balance: monthly information about the previous loans.
# Each month has its own row.
# https://www.kaggle.com/shanth84/home-credit-bureau-data-feature-engineering
```

```
# UNDERSTANDING OF VARIABLES¶
# CREDIT_ACTIVE - Current status of a Loan - Closed/ Active (2 values)

# CREDIT_CURRENCY - Currency in which the transaction was executed - Currency1, Currency2, Currency3, Currency4 ( 4 value

# CREDIT_DAY_OVERDUE - Number of overdue days

# CREDIT_TYPE - Consumer Credit, Credit card, Mortgage, Car loan, Microloan, Loan for working capital replemishment, Loan

# DAYS_CREDIT - Number of days ELAPSED since customer applied for CB credit with respect to current application Interpret

# DAYS_CREDIT_ENDDATE - Number of days the customer CREDIT is valid at the time of application CREDIT_DAY_OVERDUE - Numbe

# AMT_CREDIT_SUM - Total available credit for a customer AMT_CREDIT_SUM_DEBT - Total amount yet to be repayed
# AMT_CREDIT_SUM_LIMIT - Current Credit that has been utilized
# AMT_CREDIT_SUM_OVERDUE - Current credit payment that is overdue
# CNT_CREDIT_PROLONG - How many times was the Credit date prolonged
bureau = pd.read_csv('bureau.csv')
bureau = convert_types(bureau, print_info = True)
bureau.info()
# FEATURE 1 - NUMBER OF PAST LOANS PER CUSTOMER
previous_loan_counts = bureau.groupby('SK_ID_CURR', as_index = False)['SK_ID_BUREAU'].count().rename(columns ={'SK_ID_BUR
previous_loan_counts.head()
train = train.merge(previous_loan_counts, on = 'SK_ID_CURR', how = 'left')
train['previous_loan_counts'] = train['previous_loan_counts'].fillna(0)
train.info()

# bureau_time_features =bureau[['SK_ID_CURR','DAYS_CREDIT_UPDATE', 'DAYS_ENDDATE_FACT', 'DAYS_CREDIT_ENDDATE',
#                         'CREDIT_DAY_OVERDUE', 'DAYS_CREDIT']]
# New time features
bureau['bureau_CREDIT_DAY_OVERDUE_TIME_DAYS_CREDIT'] = bureau['CREDIT_DAY_OVERDUE'] * bureau['DAYS_CREDIT']
bureau['bureau_AMT_CREDIT_SUM - AMT_CREDIT_SUM_DEBT'] = bureau['AMT_CREDIT_SUM'] - bureau['AMT_CREDIT_SUM_DEBT']
bureau['bureau_AMT_CREDIT_SUM - AMT_CREDIT_SUM_LIMIT'] = bureau['AMT_CREDIT_SUM'] - bureau['AMT_CREDIT_SUM_LIMIT']
bureau['bureau_AMT_CREDIT_SUM - AMT_CREDIT_SUM_OVERDUE'] = bureau['AMT_CREDIT_SUM'] - bureau['AMT_CREDIT_SUM_OVERDUE']

bureau['bureau_DAYS_CREDIT - CREDIT_DAY_OVERDUE'] = bureau['DAYS_CREDIT'] - bureau['CREDIT_DAY_OVERDUE']
bureau['bureau_DAYS_CREDIT - DAYS_CREDIT_ENDDATE'] = bureau['DAYS_CREDIT'] - bureau['DAYS_CREDIT_ENDDATE']
bureau['bureau_DAYS_CREDIT - DAYS_ENDDATE_FACT'] = bureau['DAYS_CREDIT'] - bureau['DAYS_ENDDATE_FACT']
bureau['bureau_DAYS_CREDIT_ENDDATE - DAYS_ENDDATE_FACT'] = bureau['DAYS_CREDIT_ENDDATE'] - bureau['DAYS_ENDDATE_FACT']
bureau['bureau_DAYS_CREDIT_UPDATE - DAYS_CREDIT_ENDDATE'] = bureau['DAYS_CREDIT_UPDATE'] - bureau['DAYS_CREDIT_ENDDATE']

# Conduct Aggregation
```

```
bureau_by_client = aggregate_client(bureau, group_vars =['SK_ID_BUREAU', 'SK_ID_CURR'], df_names = ['bureau', 'client'])

list(bureau_by_client.columns)

train=train.merge(bureau_by_client, on = 'SK_ID_CURR', how = 'left' )
test=test.merge(bureau_by_client, on = 'SK_ID_CURR', how = 'left' )

gc.enable()
del bureau , bureau_by_client
gc.collect()
train = remove_missing_columns(train)
train.info()
train.to_csv('train_after_stage5_1.csv', index = False)
test.to_csv('test_after_stage5_1.csv', index = False)
#test_labels = test['TARGET']
#test_labels.to_csv('test_labels.csv', index = False)
#test1 = test.drop(columns =['TARGET'])
########### 2. Bureau_balance Data Set ##########################################################################
# bureau: information about client's previous loans with other financial institutions reported to Home Credit. Each previ
bureau_balance = pd.read_csv('bureau_balance.csv')
bureau_balance.head()
bureau = pd.read_csv('bureau.csv')[['SK_ID_BUREAU', 'SK_ID_CURR']]
bureau_balance = bureau_balance.merge(bureau, on ='SK_ID_BUREAU', how = 'left')

bureau_balance = convert_types(bureau_balance, print_info = True)
bureau_balance.info()

bureau_balance_by_client = aggregate_client(bureau_balance, group_vars =['SK_ID_BUREAU', 'SK_ID_CURR'], df_names = ['bure

bureau_balance_by_client.head()

train=train.merge(bureau_balance_by_client, on = 'SK_ID_CURR', how = 'left')
test=test.merge(bureau_balance_by_client, on = 'SK_ID_CURR', how = 'left')

gc.enable()
del bureau_balance_by_client, bureau_balance, bureau
gc.collect()
train, test = remove_missing_columns(train, test)
train.info()

train.to_csv('train_after_stage2_0.csv', index = False) # 970 features
test.to_csv('test_after_stage2_0.csv', index = False)
```

```
########### 3.previous_application Data Set ###################################################################

previous=pd.read_csv('previous_application.csv')
previous = convert_types(previous, print_info=True)
previous.head()

previous['prev AMT_APPLICATION / AMT_CREDIT'] = previous['AMT_APPLICATION'] / previous['AMT_CREDIT']
previous['prev AMT_APPLICATION - AMT_CREDIT'] = previous['AMT_APPLICATION'] - previous['AMT_CREDIT']
previous['prev AMT_APPLICATION - AMT_GOODS_PRICE'] = previous['AMT_APPLICATION'] - previous['AMT_GOODS_PRICE']
previous['prev AMT_GOODS_PRICE - AMT_CREDIT'] = previous['AMT_GOODS_PRICE'] - previous['AMT_CREDIT']
previous['prev DAYS_FIRST_DRAWING - DAYS_FIRST_DUE'] = previous['DAYS_FIRST_DRAWING'] - previous['DAYS_FIRST_DUE']
previous['prev DAYS_TERMINATION less -500'] = (previous['DAYS_TERMINATION'] < -500).astype(int)

previous_agg = agg_numeric(previous, 'SK_ID_CURR', 'previous')
previous_agg.shape # 37 columns -> 70 columns


previous_counts = agg_categorical(previous, 'SK_ID_CURR', 'previous')
previous_counts.shape # 37 columns -> 285 columns
list(previous_counts.columns)

# train = pd.read_csv('train_after_stage1.csv')
train = convert_types(train)

# Merge new features into train and test
#train = train.merge(previous_counts, on ='SK_ID_CURR', how = 'left')
train = train.merge(previous_agg, on = 'SK_ID_CURR', how = 'left')
test = test.merge(previous_agg, on = 'SK_ID_CURR', how = 'left')

# Remove variables to free memory
gc.enable()
del previous, previous_agg, previous_counts
gc.collect()

train, test = remove_missing_columns(train, test)


########### 4_ Monthly Cash Data Set ###########################################################################

cash = pd.read_csv('POS_CASH_balance.csv')
```

```
cash = convert_types(cash, print_info = True)
cash.head()
cash.info()

# Replace some outliers
cash.loc[cash['CNT_INSTALMENT_FUTURE'] > 60, 'CNT_INSTALMENT_FUTURE'] = np.nan

# Some new features
cash['pos CNT_INSTALMENT more CNT_INSTALMENT_FUTURE'] = (cash['CNT_INSTALMENT'] > cash['CNT_INSTALMENT_FUTURE']).astype(i



cash_by_client = aggregate_client(cash, group_vars =['SK_ID_PREV', 'SK_ID_CURR'], df_names =['cash', 'client'])
cash_by_client.info()
cash_by_client.head()

print('Cash by client Shape: ', cash_by_client.shape)
train = train.merge(cash_by_client, on ='SK_ID_CURR', how ='left')

gc.enable()
del cash, cash_by_client
gc.collect()

train, test= remove_missing_columns(train, test)

########### 5_ Monthly Credit Data Set ##################################################################################


credit = pd.read_csv('credit_card_balance.csv')
credit = convert_types(credit, print_info = True)
credit.info()
credit.head()

 # Replace some outliers
credit.loc[credit['AMT_PAYMENT_CURRENT'] > 4000000, 'AMT_PAYMENT_CURRENT'] = np.nan
credit.loc[credit['AMT_CREDIT_LIMIT_ACTUAL'] > 1000000, 'AMT_CREDIT_LIMIT_ACTUAL'] = np.nan

# Some new features
credit['credit_card_ missing'] = credit.isnull().sum(axis = 1).values
credit['credit_card_SK_DPD - MONTHS_BALANCE'] = credit['SK_DPD'] - credit['MONTHS_BALANCE']
credit['credit_card_SK_DPD_DEF - MONTHS_BALANCE'] = credit['SK_DPD_DEF'] - credit['MONTHS_BALANCE']
```

```
credit['credit_card_SK_DPD - SK_DPD_DEF'] = credit['SK_DPD'] - credit['SK_DPD_DEF']

credit['credit_card_AMT_TOTAL_RECEIVABLE - AMT_RECIVABLE'] = credit['AMT_TOTAL_RECEIVABLE'] - credit['AMT_RECIVABLE']
credit['credit_card_AMT_TOTAL_RECEIVABLE - AMT_RECEIVABLE_PRINCIPAL'] = credit['AMT_TOTAL_RECEIVABLE'] - credit['AMT_RECE
credit['credit_card_AMT_RECIVABLE - AMT_RECEIVABLE_PRINCIPAL'] = credit['AMT_RECIVABLE'] - credit['AMT_RECEIVABLE_PRINCIP

credit['credit_card_AMT_BALANCE - AMT_RECIVABLE'] = credit['AMT_BALANCE'] - credit['AMT_RECIVABLE']
credit['credit_card_AMT_BALANCE - AMT_RECEIVABLE_PRINCIPAL'] = credit['AMT_BALANCE'] - credit['AMT_RECEIVABLE_PRINCIPAL']
credit['credit_card_AMT_BALANCE - AMT_TOTAL_RECEIVABLE'] = credit['AMT_BALANCE'] - credit['AMT_TOTAL_RECEIVABLE']
credit['credit_card_AMT_DRAWINGS_CURRENT - AMT_DRAWINGS_ATM_CURRENT'] = credit['AMT_DRAWINGS_CURRENT'] - credit['AMT_DRAW
credit['credit_card_AMT_DRAWINGS_CURRENT - AMT_DRAWINGS_OTHER_CURRENT'] = credit['AMT_DRAWINGS_CURRENT'] - credit['AMT_DR
credit['credit_card_AMT_DRAWINGS_CURRENT - AMT_DRAWINGS_POS_CURRENT'] = credit['AMT_DRAWINGS_CURRENT'] - credit['AMT_DRAW

credit_by_client = aggregate_client(credit, group_vars=['SK_ID_PREV', 'SK_ID_CURR'], df_names=['credit','client'])
credit_by_client.head()

train = train.merge(credit_by_client, on = 'SK_ID_CURR', how ='left')
test = test.merge(credit_by_client, on = 'SK_ID_CURR', how ='left')

gc.enable()
del credit, credit_by_client
gc.collect()
train, test = remove_missing_columns(train, test)

train.to_csv('train_after_stage2_1.csv', index = False) # 2600 features
test.to_csv('train_after_stage2_1.csv', index = False)

########### 6_ Installment Payments Data Set ############################################################################

installments =pd.read_csv('installments_payments.csv')
installments = convert_types(installments, print_info = True)
installments.info()
installments.head()
# Replace some outliers
# Replace some outliers
installments.loc[installments['NUM_INSTALMENT_VERSION'] > 70, 'NUM_INSTALMENT_VERSION'] = np.nan
installments.loc[installments['DAYS_ENTRY_PAYMENT'] < -4000, 'DAYS_ENTRY_PAYMENT'] = np.nan


# Percentage and difference paid in each installment (amount paid and installment value)
installments['ins_PAYMENT_PERC'] = installments['AMT_PAYMENT'] / installments['AMT_INSTALMENT']
installments['ins_PAYMENT_DIFF'] = installments['AMT_INSTALMENT'] - installments['AMT_PAYMENT']
```

```
# Days past due and days before due (no negative values)
installments['ins_DPD'] = installments['DAYS_ENTRY_PAYMENT'] - installments['DAYS_INSTALMENT']
installments['ins_DBD'] = installments['DAYS_INSTALMENT'] - installments['DAYS_ENTRY_PAYMENT']
installments['ins_DPD'] = installments['DPD'].apply(lambda x: x if x > 0 else 0)
installments['ins_DBD'] = installments['DBD'].apply(lambda x: x if x > 0 else 0)
# Others
installments['ins_DAYS_ENTRY_PAYMENT - DAYS_INSTALMENT'] = installments['DAYS_ENTRY_PAYMENT'] - installments['DAYS_INSTAL
installments['ins_NUM_INSTALMENT_NUMBER_100'] = (installments['NUM_INSTALMENT_NUMBER'] == 100).astype(int)
installments['ins_DAYS_INSTALMENT more NUM_INSTALMENT_NUMBER'] = (installments['DAYS_INSTALMENT'] > installments['NUM_INS


installments_by_client = aggregate_client(installments, group_vars =['SK_ID_PREV', 'SK_ID_CURR'], df_names = ['installmen

installments_by_client.head()

train=train.merge(installments_by_client, on = 'SK_ID_CURR', how = 'left' )
test=test.merge(installments_by_client, on = 'SK_ID_CURR', how = 'left' )

gc.enable()
del installments, installments_by_client
gc.collect()
train, test = remove_missing_columns(train, test)
train.info()

print(f'Final training size: {return_size(train)}')


train.to_csv('train_after_stage3_1.csv', index = False) # 3200 features
test.to_csv('test_after_stage3_1.csv', index = False)
```

In [ ]:
```
######### PATR 2: Import models that will be used ######################################################################
############## Model function for testing  ('logistic_model', 'random_forest_model', 'xg_boost_model', 'lgb_model') using
from sklearn.preprocessing import MinMaxScaler, Imputer
from sklearn.linear_model import LogisticRegression
from xgboost.sklearn import XGBClassifier
from sklearn.ensemble import RandomForestClassifier
import lightgbm as lgb


def model(features, test_features, encoding = 'ohe', n_folds = 5, im ='median', used_model = 'lgb_model'):

    """Train and test a model ('logistic_model', 'random_forest_model', 'xg_boost_model', 'lgb_model') using cross valida

    Parameters
    --------
        features (pd.DataFrame):
            dataframe of training features to use
            for training a model. Must include the TARGET column.
        test_features (pd.DataFrame):
            dataframe of testing features to use
            for making predictions with the model.
        encoding (str, default = 'ohe'):
            method for encoding categorical variables. Either 'ohe' for one-hot encoding or 'le' for integer label encodi
        n_folds (int, default = 5):
            number of folds to use for cross validation
        used_model:
            choose one of 4 following models ('logistic_model', 'random_forest_model', 'xg_boost_model', 'lgb_model')


    Return
    --------
        submission (pd.DataFrame):
            dataframe with `SK_ID_CURR` and `TARGET` probabilities
            predicted by the model.
        feature_importances (pd.DataFrame):
            dataframe with the feature importances from the model.
        valid_metrics (pd.DataFrame):
            dataframe with training and validation metrics (ROC AUC) for each fold and overall.
    """
    # Extract the ids:  features = train , test_features= test
    features, test_features = remove_missing_columns(features, test_features, threshold = 99.9)
```

```
    train_ids = features ['SK_ID_CURR']
    test_ids = test_features['SK_ID_CURR']

    # Extract the labels for training
    labels = features['TARGET']

    # Remove the ids and target
    train.shape
    test.shape
    features.shape
    test_features.shape
    features = features.drop(columns = ['SK_ID_CURR', 'TARGET'])
    test_features = test_features.drop(columns = ['SK_ID_CURR'])

    # One Hot Encoding

    if encoding =='ohe':
        features =pd.get_dummies(features)
        test_features = pd.get_dummies(test_features)

        # Align the data by columns:
        features, test_features = features.align(test_features, join='inner', axis = 1)
        # No categorical indices to record
        cat_indices = 'auto'

    # Integer Label Encoding
    elif encoding == 'le':
        # Create a label encoder
        label_encoder = LabelEncoder()
        # List for storing categorical indices
        cat_indices = []
        # Iterate through each column:
        for i, col in enumerate(features):
            if features[col].dtype == 'object':
                # Map the categorical features to intergers
                features[col] = label_encoder.fit_transform(np.array(features[col].astype(str)).reshape((-1,)))
                test_features[col] = label_encoder.transform(np.array(test_featutes).astype(str).reshape((-1,)))

                # Record the categorical indices
                cat_indices.append(i)

    # Catch error if label encoding scheme is not valid
```

```
    else:
        raise ValueError("Encoding must be either 'ohe' or 'le'")
    print('Training Data Shape: ', features.shape)
    print('Testing Data Shape: ', test_features.shape)

    # Extract feature names
    feature_names = list(features.columns)
    ##### Median imputation of missing values for 'logistic_model'|'random_forest_model'
    if used_model == 'logistic_model'or'random_forest_model':

        if im=='median':

            imputer = Imputer(strategy ='median') # can replaced by 'most_frequent' or 'constant'

            imputer.fit(features)
            features = imputer.transform(features)
            test_features = imputer.transform(test_features)
        elif im =='zero':
            features.fillna(0, inplace=True)
            test_features.fillna(0, inplace=True)
        elif im =='number':
            features.fillna(30000, inplace=True)
            test_features.fillna(30000, inplace=True)
        else:
            print("NA must be filled by 'median' or '0' or 'a number'")


    ##### Scale to 0-1 for logistic regression
    if used_model == 'logistic_model':
        scaler = MinMaxScaler(feature_range =(0,1))
        scaler.fit(features)
        features = scaler.transform(features)
        test_features = scaler.transform(test_features)


    # Convert to np arrays
    if used_model == 'xg_boost_model'or'lgb_model':
        features = np.array(features)
        test_features = np.array(test_features)


    # Create the kfold object
```

```
    k_fold = KFold(n_splits = n_folds, shuffle = False, random_state = 50)
    # Empty array for test predictions
    test_predictions = np.zeros(test_features.shape[0])
    # Empty array for feature importances
    feature_importance_values = np.zeros(len(feature_names))
    # Empty array for out of fold validation prediction
    out_of_fold = np.zeros(features.shape[0])
    train_prediction = np.zeros(features.shape[0])
    # Lists for recording validation and training scores
    valid_scores = []
    train_scores = []

    if used_model == 'logistic_model':
        for train_indices, valid_indices in k_fold.split(features):
            print(train_indices, valid_indices)
            train_features, train_labels = features[train_indices], labels[train_indices]

            valid_features, valid_labels = features[valid_indices], labels[valid_indices]
            # Create the model

            model = LogisticRegression(C=0.0001)
            # Train the model
            model.fit(train_features, train_labels)

            # Make predictions
            train_prediction = model.predict_proba(train_features)[:,1]
            train_auc = roc_auc_score(train_labels, train_prediction)
            train_scores.append(train_auc)
            valid_prediction = model.predict_proba(valid_features)[:,1]
            valid_auc = roc_auc_score(valid_labels, valid_prediction)
            valid_scores.append(valid_auc)
            test_predictions += model.predict_proba(test_features)[:,1]/k_fold.n_splits

            # Record the out of fold predictions
            out_of_fold[valid_indices] = model.predict_proba(valid_features)[:,1]

            # Clean up memory
            gc.enable()
            del model, train_features, valid_features
            gc.collect()

        # Overall validation score
```

```
        valid_auc = roc_auc_score(labels, out_of_fold)
        # Add the overall scores to the metrics
        valid_scores.append(valid_auc)
        train_scores.append(np.mean(train_scores))

        # creating dataframe of validation scores
        fold_names = list(range(n_folds))
        fold_names.append('overall')

        # Dataframe of validation scores
        metrics = pd.DataFrame({'fold': fold_names,
                                'train': train_scores,
                                'valid': valid_scores})
        return metrics, test_predictions

    elif used_model == 'random_forest_model':
        for train_indices, valid_indices in k_fold.split(features):
            train_features, train_labels = features[train_indices], labels[train_indices]
            valid_features, valid_labels = features[valid_indices], labels[valid_indices]
            # Create the model

            model = RandomForestClassifier(n_estimators = 1000, random_state = 50, verbose = 1, n_jobs = -1, max_depth= 1
            # Train the model
            model.fit(train_features, train_labels)
            #model.fit(features, labels)

            # Record the feature importances
            feature_importance_values += model.feature_importances_/k_fold.n_splits

            # Make predictions
            train_prediction = model.predict_proba(train_features)[:,1]
            train_auc = roc_auc_score(train_labels, train_prediction)
            train_scores.append(train_auc)
            valid_prediction = model.predict_proba(valid_features)[:,1]
            valid_auc = roc_auc_score(valid_labels, valid_prediction)
            valid_scores.append(valid_auc)
            test_predictions += model.predict_proba(test_features)[:,1]/k_fold.n_splits

            # Record the out of fold predictions
            out_of_fold[valid_indices] = model.predict_proba(valid_features)[:,1]

            # Clean up memory
```

```
            gc.enable()
            del model, train_features, valid_features
            gc.collect()

        # Overall validation score
        valid_auc = roc_auc_score(labels, out_of_fold)
        # Add the overall scores to the metrics
        valid_scores.append(valid_auc)
        train_scores.append(np.mean(train_scores))

        # creating dataframe of validation scores
        fold_names = list(range(n_folds))
        fold_names.append('overall')


        # Dataframe of validation scores
        metrics = pd.DataFrame({'fold': fold_names,
                                'train': train_scores,
                                'valid': valid_scores})

        # Make the feature importance dataframe
        feature_importances = pd.DataFrame({'feature': feature_names, 'importance': feature_importance_values })
        feature_importances = feature_importances.sort_values('importance', ascending = False)
        return metrics, feature_importances, test_predictions
    elif used_model == 'xg_boost_model':
        for train_indices, valid_indices in k_fold.split(features):
            train_features, train_labels = features[train_indices], labels[train_indices]
            valid_features, valid_labels = features[valid_indices], labels[valid_indices]
            # Create the model

            params = {'objective': 'binary:logistic',
                      'max_depth': 5,
                      'learning_rate': 0.05,# 0.005
                      'silent': False,
                      'n_estimators': 5000,
                      'n_jobs=':-1
                  }

            #params = {'objective': 'binary:logistic',
            #           'max_depth': 5,
            #           'learning_rate': 0.01,# 0.005
            #           'silent': False,
```

```
#           'n_estimators': 5000,
#           "gamma": 0.0,
#           "min_child_weight": 10, # default: 1
#           "subsample": 0.7,
#           "colsample_bytree": 0.7,  # default:  1.0
#           "colsample_bylevel": 0.5, # default: 1.0
#           "reg_alpha": 0.0,
#           "reg_lambda": 1.0,
#           "scale_pos_weight": 1.0,
#           "random_state": 0,
##           #
#           "silent": False,
#           "n_jobs": 16,
#           #
#           "tree_method": "gpu_hist", # default: auto
#           "grow_policy": "lossguide", # default depthwise
#           "max_leaves": 0, # default: 0(unlimited)
#           "max_bin": 256  # default: 256
#           }

model = XGBClassifier(**params)
# Train the model

model.fit(train_features, train_labels,eval_set = [(train_features, train_labels), (valid_features, valid_lab
                                                   eval_metric = 'auc', early_stopping_rounds = 100, verbose=
# record the best iteration

best_iteration = model.best_iteration

# Record the feature importances
feature_importance_values += model.feature_importances_/k_fold.n_splits

# Make predictions
train_prediction = model.predict_proba(train_features)[:,1]
train_auc = roc_auc_score(train_labels, train_prediction)
train_scores.append(train_auc)
valid_prediction = model.predict_proba(valid_features)[:,1]
valid_auc = roc_auc_score(valid_labels, valid_prediction)
valid_scores.append(valid_auc)
test_predictions += model.predict_proba(test_features, ntree_limit = best_iteration)[:,1]/k_fold.n_splits

# Record the out of fold predictions
```

```
            out_of_fold[valid_indices] = model.predict_proba(valid_features, ntree_limit = best_iteration)[:,1]

            # Clean up memory
            gc.enable()
            del model, train_features, valid_features
            gc.collect()

        # Overall validation score
        valid_auc = roc_auc_score(labels, out_of_fold)
        # Add the overall scores to the metrics
        valid_scores.append(valid_auc)
        train_scores.append(np.mean(train_scores))

        # creating dataframe of validation scores
        fold_names = list(range(n_folds))
        fold_names.append('overall')


        # Dataframe of validation scores
        metrics = pd.DataFrame({'fold': fold_names,
                                'train': train_scores,
                                'valid': valid_scores})

        # Make the feature importance dataframe
        feature_importances = pd.DataFrame({'feature': feature_names, 'importance': feature_importance_values })
        feature_importances = feature_importances.sort_values('importance', ascending = False)
        return metrics, feature_importances, test_predictions
    else:
        for train_indices, valid_indices in k_fold.split(features):

            # Training data for the fold
            train_features, train_labels = features[train_indices], labels[train_indices]
            # Validation data for the fold
            valid_features, valid_labels = features[valid_indices], labels[valid_indices]

            # Create the model
            model = lgb.LGBMClassifier(n_estimators=10000, objective = 'binary',
                                       class_weight = 'balanced', learning_rate = 0.05,
                                       reg_alpha = 0.1, reg_lambda = 0.1,
                                       subsample = 0.8, n_jobs = -1, random_state = 50)

            # Train the model
```

```
        model.fit(train_features, train_labels, eval_metric = 'auc',
              eval_set = [(valid_features, valid_labels), (train_features, train_labels)],
              eval_names = ['valid', 'train'], categorical_feature = cat_indices,
              early_stopping_rounds = 100, verbose = 200)

        # Record the best iteration
        best_iteration = model.best_iteration_

        # Record the feature importances
        feature_importance_values += model.feature_importances_ / k_fold.n_splits

        # Make predictions
        test_predictions += model.predict_proba(test_features, num_iteration = best_iteration)[:, 1] / k_fold.n_split

        # Record the out of fold predictions
        out_of_fold[valid_indices] = model.predict_proba(valid_features, num_iteration = best_iteration)[:, 1]

        # Record the best score
        valid_score = model.best_score_['valid']['auc']
        train_score = model.best_score_['train']['auc']

        valid_scores.append(valid_score)
        train_scores.append(train_score)

        # Clean up memory
        gc.enable()
        del model, train_features, valid_features
        gc.collect()

    # Make the submission dataframe
    submission = pd.DataFrame({'SK_ID_CURR': test_ids, 'TARGET': test_predictions})

    # Make the feature importance dataframe
    feature_importances = pd.DataFrame({'feature': feature_names, 'importance': feature_importance_values})

    # Overall validation score
    valid_auc = roc_auc_score(labels, out_of_fold)
    # test_auc = roc_auc_score(test_labels, test_predictions)

    # Add the overall scores to the metrics
    valid_scores.append(valid_auc)
    train_scores.append(np.mean(train_scores))
```

```
    # Needed for creating dataframe of validation scores
    fold_names = list(range(n_folds))
    fold_names.append('overall')

    # Dataframe of validation scores
    metrics = pd.DataFrame({'fold': fold_names,
                            'train': train_scores,
                            'valid': valid_scores})

    return metrics, feature_importances, test_predictions
```

In [ ]:
```
######### PART 3: Test  Models with 'train_after_stage2_1.csv' data ############
#########Test  Models with 'train_after_stage2_1.csv' data ####################
# os.chdir('C:\\Users\\Phong\\')
app_train = pd.read_csv('train_after_stage2_1.csv') # 1875 features
# app_train = convert_types(app_train, print_info = True)
print('Finish reading data, start spliting data into train, test')
train, test = split_train_test(app_train, 0.2)
train.to_csv('train_after_stage2_1_1.csv', index = False)
test.to_csv('test_after_stage2_1_1.csv', index = False)
gc.enable()
del app_train
gc.collect()
print('Finish record the data, read it again')
print('Eliminate infinity values and remove columns with missing values > 99%')
train = pd.read_csv('train_after_stage2_1_1.csv')
test = pd.read_csv('test_after_stage2_1_1.csv').drop(columns =['TARGET'])
test_labels =  pd.read_csv('test_after_stage2_1_1.csv')['TARGET']
train = train.replace([np.inf, -np.inf], np.nan)
test = test.replace([np.inf, -np.inf], np.nan)
train, test = remove_missing_columns(train, test, threshold = 99)

#from sklearn.preprocessing import MinMaxScaler, Imputer


# Logistic Model
print('Start running Logistic Model for train_after_stage2_1.csv data ')
auc_lg2, prediction_lg2= model(train, test, used_model ='logistic_model')
test_auc_lg2 = roc_auc_score(test_labels, prediction_lg2)
logic = ['No']*5
logic.append(test_auc_lg2)
auc_lg2['logic'] = logic
auc_lg2
# Random Forest Model
print('Start running Random Forest Model for train_after_stage2_1.csv data ')
auc_rf2, feature_importances_rf2, prediction_rf2= model(train, test, used_model ='random_forest_model')
test_auc_rf2 = roc_auc_score(test_labels, prediction_rf2)
logic = ['No']*5
logic.append(test_auc_rf2)
auc_rf2['random_forest'] = logic
auc_rf2
# Light Gradient Boosting Model
```

```
print('Start running Light Gradient Boosting Model for train_after_stage2_1.csv data ')
auc_lgb2, feature_importances_lgb2, prediction_lgb2 = model(train, test, used_model ='lgb_model') #
test_auc_lgb2 = roc_auc_score(test_labels, prediction_lgb2)
logic = ['No']*5
logic.append(test_auc_lgb2)
auc_lgb2['light gradient boosting'] = logic
auc_lgb2

# XG boosting Model
print('Start running XG boosting Model for train_after_stage2_1.csv data ')
auc_xg2, feature_importances_xg2, prediction_xg2 =  model(train, test, used_model ='xg_boost_model')
test_auc_xg2 = roc_auc_score(test_labels, prediction_xg2)
logic = ['No']*5
logic.append(test_auc_xg2)
auc_xg2['xg_boosting'] = logic
auc_xg2
print('Summary results for train_after_stage2_1.csv data ')
testing_summary_stage2 = pd.concat([auc_lg2,auc_rf2, auc_xg2, auc_lgb2 ], axis=1)
testing_summary_stage2
```

In [ ]:
```
######### PART 4: Test  Models with 'train_after_stage3_1.csv' data  ############
app_train = pd.read_csv('train_after_stage3_1.csv') # 'application_train.csv'
#app_train = convert_types(app_train, print_info = True)
print('Finish reading data, start spliting data into train, test')

train, test = split_train_test(app_train, 0.2)
train.to_csv('train_after_stage3_1_1.csv', index = False)
test.to_csv('test_after_stage3_1_1.csv', index = False)
gc.enable()
del app_train
gc.collect()
print('Finish record the data, read it again')
print('Eliminate infinity values and remove columns with missing values > 99%')

train = pd.read_csv('train_after_stage3_1_1.csv')
test = pd.read_csv('train_after_stage3_1_1.csv').drop(columns =['TARGET'])
test_labels =  pd.read_csv('train_after_stage3_1_1.csv')['TARGET']
train = train.replace([np.inf, -np.inf], np.nan)
test = test.replace([np.inf, -np.inf], np.nan)
train, test = remove_missing_columns(train, test, threshold = 99)


# Logistic Model
print('Start running Logistic Model for train_after_stage3_1.csv data ')
auc_lg3, prediction_lg3=model(train, test, used_model ='logistic_model')
test_auc_lg3 = roc_auc_score(test_labels, prediction_lg3)
logic = ['No']*5
logic.append(test_auc_lg3)
auc_lg3['logic'] = logic
auc_lg3
# Random Forest Model
print('Start running Random Forest Model for train_after_stage3_1.csv data ')
auc_rf3, feature_importances_rf3, prediction_rf3 =  model(train, test, used_model ='random_forest_model')
test_auc_rf3 = roc_auc_score(test_labels, prediction_rf3)
logic = ['No']*5
logic.append(test_auc_rf3)
auc_rf3['random_forest'] = logic
auc_rf3
# Light Gradient Boosting Model
print('Start running Light Gradient Boosting Model for train_after_stage3_1.csv data ')
auc_lgb3, feature_importances_lgb3, prediction_lgb3 = model(train, test, used_model ='lgb_model')#
```

```
test_auc_lgb3 = roc_auc_score(test_labels, prediction_lgb3)
logic = ['No']*5
logic.append(test_auc_lgb3)
auc_lgb3['light gradient boosting'] = logic
auc_lgb3
# XG boosting Model
print('Start running XG boosting Model for train_after_stage3_1.csv data ')
auc_xg3, feature_importances_xg3, prediction_xg3 =  model(train, test, used_model ='xg_boost_model')
test_auc_xg3 = roc_auc_score(test_labels, prediction_xg3)

logic = ['No']*5
logic.append(test_auc_xg3)
auc_xg3['xg_boosting'] = logic
auc_xg3

print('Summary results for train_after_stage3_1.csv data ')
testing_summary_stage3 = pd.concat([auc_lg3,auc_rf3, auc_xg3,auc_lgb3 ], axis=1)
testing_summary_stage3

print('Summary results for train_after_stage2_1.csv and train_after_stage3_1.csv data ')
testing_summary = pd.concat([testing_summary_stage2, testing_summary_stage3], axis=0)

testing_summary
```

In [ ]: