Home Credit Default Risk Part 3: Model Training

After we finish feature engineering, we will move forward to machine learning modelling. We will start with simple model like Logistic Regression, Linear SVM, etc. Then we will ensemble like Random Forest, Boosting techniques, and finally stacking techniques. We will choose the model which give the best score.

Table of contents

Loading libries

In [3]:

```
#import usefull DataStructures
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.linear_model import SGDClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import ExtraTreesClassifier
from sklearn.calibration import CalibratedClassifierCV

#other modelling libraries
from bayes_opt import BayesianOptimization #https://github.com/fmfn/BayesianOptimizatio
import xgboost as xgb
from xgboost import XGBClassifier
from xgboost import XGBRegressor

import lightgbm as lgb
from lightgbm import LGBMClassifier
from lightgbm import LGBMRegressor
```

loading saved processed data from part 2

In [9]:

```
#loading saved processed data from part 2
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" ) )
```

In [5]:

```
#standardizing the data
scaler = StandardScaler()
x_train_std = scaler.fit_transform(train_data)
x_test_std = scaler.fit_transform(test_data)
y_train = target_train

#replacing nan values with 0
x_train_std[np.isnan(x_train_std)] = 0
x_test_std[np.isnan(x_test_std)] = 0
```

1. Defining Utility Functions and Classes

- `modelling`

  Class for Doing Hyperparameter tuning to find best set of hyperparameters, building models on best hyperparams and displaying results on best hyperparameters.

It has 4 methods:

1. init method
2. random_search_cv method
3. train_on_best_params method
4. proba_to_class method
5. tune_threshold method
6. results_on_best_params method
7. feat_importances_show method

- `Boosting`

Class for Boosting Ensembles and displaying results. Contains 6 methods:

1. init method
2. train method
3. proba_to_class method
4. tune_threshold method
5. results method
6. feat_importance_show

In [6]:

```
class modelling:
    '''
    Class for Doing Hyperparameter tuning to find best set of hyperparameters,
    building models on best hyperparams and displaying results on best hyperparameters.

    It has 4 methods:
        1. init method
        2. random_search_cv method
        3. train_on_best_params method
        4. proba_to_class method
        5. tune_threshold method
        6. results_on_best_params method
        7. feat_importances_show method
    '''
    def __init__(self, base_model, x_train, y_train, x_test, calibration = False,
                 calibration_method = 'isotonic', calibration_cv = 4, k_folds = 4,
                 random_state = 982):
        '''
        Function to initialize the class members.

        Inputs:
            self
            base_model: estimator/classifier
                The base model to be used for the modelling purpose
            x_train: numpy array
                Training standardized data
            y_train: numpy array
                Training class labels
            x_test: numpy array
                Test standardized data
            calibration: bool, default = False
                Whether to calibrate the model for generating class probabilities
            calibration_method: str, default = 'isotonic'
```

```
                The type of calibration to use, i.e. sigmoid or isotonic
            calibration_cv: int, default = 4
                Number of cross-validation folds for calibrating the probabilities
            k_folds: int, default = 4
                Number of cross-validation folds for training and tuning the model
            random_state: int, default = 982
                Random state for StratifiedKFold for reproducibility

        Returns:
            None
        '''
        self.base_model = base_model
        self.num_folds = k_folds
        self.kfolds = StratifiedKFold(n_splits = k_folds, shuffle = True,
                                      random_state = random_state)
        self.x_train = x_train
        self.y_train = y_train
        self.x_test = x_test
        self.calibration = calibration
        if self.calibration:
            self.calibration_method = calibration_method
            self.calibration_cv = calibration_cv

    def random_search_cv(self, hyperparams_dict, n_iter = 30, verbose = True, n_jobs =
                         random_state = 843):
        '''
        Function to do RandomizedSearchCV on training data.

        Inputs:
            self
            hyperparams_dict: dict
                Dictionary of hyperparameters to tune
            n_iter: int, default = 30
                Number of iterations to perform for random search
            verbose: bool, default = True
                Whether to keep verbosity or not
            n_jobs: int, default = 1
                Number of cores to use for Random Search
            random_state: int, default = 843
                Random state for reproducibility of RandomizedSearchCV

        Returns:
            None
        '''

        if verbose:
            start = datetime.now()
            print(f"Doing Randomized Search CV on Classifier with {n_iter} random initi
        rscv = RandomizedSearchCV(self.base_model, hyperparams_dict, n_iter = n_iter, s
                                  cv = self.kfolds, return_train_score = True, verbose
                                  n_jobs = n_jobs, random_state = random_state)
        rscv.fit(self.x_train, self.y_train)
        if verbose:
            print("Done.")
            print(f"Time elapsed = {datetime.now() - start}")

        #getting the Search Results
        self.tuning_results = pd.DataFrame(rscv.cv_results_)
        #best model
        self.best_model = rscv.best_estimator_
```

```
        gc.collect()

    def train_on_best_params(self, verbose = True):
        '''
        Function to train the model on best hyperparameters obtained from previous meth
        Generates Cross-Validation predictions as Out-of-fold predictions

        Inputs:
            self
            verbose: bool, default = True
                Whether to keep verbosity or not

        Returns:
            None
        '''

        if verbose:
            print("Fitting Classifier on best parameters\n")
            print(f"{self.num_folds}-Fold Cross Validation")
            start = datetime.now()

        self.cv_preds_probas = np.zeros(self.x_train.shape[0])
        #we will select a best threshold for each fold of cross-validation and average
        #folds to find the optimal threshold
        self.best_threshold_train = 0
        for fold_number, (train_indices, val_indices) in enumerate(self.kfolds.split(se
            if verbose:
                print(f"\tFitting Fold {fold_number}")

            self.best_model.fit(self.x_train[train_indices], self.y_train[train_indices
            if not self.calibration:
                self.train_preds_probas = self.best_model.predict_proba(self.x_train[tr
                self.cv_preds_probas[val_indices] = self.best_model.predict_proba(self.
            else:
                #fitting the calibration Classifier over the base model for calibrated 
                self.calibrated_classifier = CalibratedClassifierCV(self.best_model, me
                                                                    cv = self.calibrati
                self.calibrated_classifier.fit(self.x_train[train_indices], self.y_trai

                self.train_preds_probas = self.calibrated_classifier.predict_proba(self
                self.cv_preds_probas[val_indices] = self.calibrated_classifier.predict_

            #tuning the threshold for optimal TPR and FPR from ROC Curve
            self.best_threshold_train += self.tune_threshold(self.y_train[train_indices

        #converting the class probabilities to class labels
        self.cv_preds_class = self.proba_to_class(self.cv_preds_probas, self.best_thres

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

        gc.collect()

    def proba_to_class(self, proba, threshold):
        '''
        Function to convert a given probability to class label based on a threshold val

        Inputs:
```

```
            self
            proba: numpy array
                Probabilities of class label = 1
            threshold: int
                Threshold probability to be considered as Positive or Negative Class La

        Returns:
            Converted Class Label
        '''
        return np.where(proba >= threshold, 1, 0)

    def tune_threshold(self, true_labels, predicted_probas):
        '''
        Function to find the optimal threshold for maximizing the TPR and minimizing th
        This is found out by using the J Statistic, which is J = TPR - FPR.
        Reference: https://machinelearningmastery.com/threshold-moving-for-imbalanced-c

        Inputs:
            self
            true_labels: numpy array or pandas series
                True Class Labels
            predicted_probas: numpy array
                Predicted Probability of Positive Class label

        Returns:
            Threshold probability.
        '''
        fpr, tpr, threshold = roc_curve(true_labels, predicted_probas)
        j_stat = tpr - fpr
        index_for_best_threshold = np.argmax(j_stat)

        return threshold[index_for_best_threshold]

    def results_on_best_params(self, model_name):
        '''
        Function to train the whole data on best parameters and display the results.

        Inputs:
            self
            model_name: str
                model name to get feature importances.

        Returns:
            None
        '''
        #we have to fit the whole model for optimal test predictions
        self.best_model.fit(self.x_train, self.y_train)
        if not self.calibration:
            self.train_preds_probas = self.best_model.predict_proba(self.x_train)[:,1]
            self.test_preds_probas = self.best_model.predict_proba(self.x_test)[:,1]
        else:
            #fitting calibration model over whole training data for test predictions
            self.calibrated_classifier.fit(self.x_train, self.y_train)
            self.train_preds_probas = self.calibrated_classifier.predict_proba(self.x_t
            self.test_preds_probas = self.calibrated_classifier.predict_proba(self.x_te

        #getting class labels from probabilities
        self.train_preds_class = self.proba_to_class(self.train_preds_probas, self.best
        self.test_preds_class = self.proba_to_class(self.test_preds_probas, self.best_t
```

```
        #feature importances
        if model_name == 'linear':
            self.feat_imp = self.best_model.coef_[0]
        else:
            self.feat_imp = self.best_model.feature_importances_

        print("=" * 100)
        print(f"\nThe best selected Threshold as per the J-Statistic, which is J = TPR
        print("Train Results:")
        print(f"\tROC-AUC Score = {roc_auc_score(self.y_train, self.train_preds_probas)
        print(f"\tPrecision Score = {precision_score(self.y_train, self.train_preds_cla
        print(f"\tRecall Score = {recall_score(self.y_train, self.train_preds_class)}")
        print("CV Results:")
        print(f"\tROC-AUC Score = {roc_auc_score(self.y_train, self.cv_preds_probas)}")
        print(f"\tPrecision Score = {precision_score(self.y_train, self.cv_preds_class)
        print(f"\tRecall Score = {recall_score(self.y_train, self.cv_preds_class)}")

        print('=' * 100)
        print("Confusion Matrix of CV data:")
        conf_mat = confusion_matrix(self.y_train, self.cv_preds_class)
        conf_mat = pd.DataFrame(conf_mat, columns = ['Predicted_0','Predicted_1'], inde
        plt.figure(figsize = (7,6))
        plt.title('Confusion Matrix Heatmap')
        sns.heatmap(conf_mat, annot = True, fmt = 'g', linewidth = 0.5, annot_kws = {'s
        plt.show()

        print('=' * 100)
        print("Distribution of Original Class Labels and Predicted CV and Test Class La
        plt.figure(figsize = (20,6))
        plt.subplot(1,3,1)
        plt.title('Class Distribution of Original Dataset')
        sns.countplot(self.y_train)
        plt.subplot(1,3,2)
        plt.title('Class Distribution of predicted Class Labels on CV')
        sns.countplot(self.cv_preds_class)
        plt.subplot(1,3,3)
        plt.title('Class Distribution of predicted Test Dataset')
        sns.countplot(self.test_preds_class)
        plt.show()
        print('=' * 100)

        gc.collect()

    def feat_importances_show(self, feature_names, num_features, figsize = (10,15)):
        '''
        Function to display the top most important features.

        Inputs:
            self
            feature_names: numpy array
                Names of features of training set
            num_features: int
                Number of top features importances to display
            figsize: tuple, default = (10,15)
                Size of figure to be displayed

        Returns:
            None
        '''
```

```
        #getting the top features indices and their names
        top_feats_indices = np.argsort(self.feat_imp)[::-1][:num_features]
        feat_importance_top = self.feat_imp[top_feats_indices]
        column_names = feature_names[top_feats_indices]

        #plotting a horizontal bar plot of feature importances
        plt.figure(figsize = figsize)
        sns.barplot(feat_importance_top, list(range(num_features)), orient = 'h')
        plt.yticks(list(range(50)), column_names)
        plt.title(f'Top {num_features} features as per classifier')
        plt.xlabel('Feature Importance')
        plt.ylabel('Feature Names')
        plt.grid()
        plt.show()
        print('=' * 100)

        gc.collect()
```

In [23]:

```
class Boosting:
    '''
    Class for Boosting Ensembles and displaying results. Contains 6 methods:

        1. init method
        2. train method
        3. proba_to_class method
        4. tune_threshold method
        5. results method
        6. feat_importance_show
    '''

    def __init__(self, x_train, y_train, x_test, params, num_folds = 3, random_state =
                 verbose = True, save_model_to_pickle = False):
        '''
        Function to initialize the class members.

        Inputs:
            self
            x_train: DataFrame
                Training DataFrame
            y_train: DataFrame
                Training Class labels
            x_test: DataFrame
                Test DataFrame
            params: dict
                Parameters for the boosting ensemble
            num_folds: int, default = 3
                Number of folds for k-Fold Cross Validation
            random_state: int, default = 33
                Random State for Splitting the data for K-Fold Cross Validation
            verbose: bool, default = True
                Whether to keep verbosity or not
            save_model_to_pickle: bool, default = False
                Whether to save the model to pickle file or not

        Returns:
            None
        '''
```

```
        self.x_train = x_train
        self.y_train = y_train
        self.x_test = x_test
        self.params = params
        self.num_folds = num_folds
        self.stratified_cv = StratifiedKFold(n_splits = num_folds, shuffle = True, rand
        self.verbose = verbose
        self.save_model = save_model_to_pickle

    def train(self, booster, verbose = 400, early_stopping = 200, pickle_name = ''):
        '''
        Function to train the Classifier on given parameters. It fits the classifier fo
        Cross Validation, uses Out-of-Fold Predictions. The test predictions are averag

        Inputs:
            self
            booster: str
                Whether the booster is 'xgboost' or 'lightgbm'
            verbose: int, default = 400
                Number of boosting rounds for printint boosting results.
            early_stopping: int, default = 200
                Number of boosting rounds to look for early stopping
            pickle_name: str, default = ''
                The string to add to end of pickle file of model, if any

        Returns:
            None
        '''

        self.train_preds_proba_mean = np.zeros(self.x_train.shape[0])
        #out-of-fold cv predictions
        self.cv_preds_proba = np.zeros(self.x_train.shape[0])
        self.test_preds_proba_mean = np.zeros(self.x_test.shape[0])
        #best threshold will be
        self.best_threshold_train = 0
        self.feature_importance = pd.DataFrame()
        self.feature_importance['features'] = self.x_train.columns
        self.feature_importance['gain'] = np.zeros(self.x_train.shape[1])

        if self.verbose:
            print(f"Fitting the {booster} on Training Data with {self.num_folds} fold c
            start = datetime.now()

        for fold_number, (train_indices, cv_indices) in enumerate(self.stratified_cv.sp
            if self.verbose:
                print(f"\n\tFold Number {fold_number}\n")

            x_tr = self.x_train.iloc[train_indices]
            y_tr = self.y_train.iloc[train_indices]
            x_cv = self.x_train.iloc[cv_indices]
            y_cv = self.y_train.iloc[cv_indices]

            if booster == 'xgboost':
                clf = XGBClassifier(**self.params)
            else:
                clf = LGBMClassifier(**self.params)

            clf.fit(x_tr, y_tr, eval_set = [(x_tr, y_tr), (x_cv, y_cv)], eval_metric =
                     verbose = verbose, early_stopping_rounds = 200)
```

```
            if booster == 'xgboost':
                self.train_preds_proba_mean[train_indices] = clf.predict_proba(x_tr, nt
                self.cv_preds_proba[cv_indices] = clf.predict_proba(x_cv, ntree_limit =
                self.test_preds_proba_mean += clf.predict_proba(self.x_test, ntree_limi

                #feature importance
                gain_fold = clf.get_booster().get_score(importance_type = 'gain')
                feat_imp = pd.DataFrame()
                feat_imp['features'] = gain_fold.keys()
                feat_imp['gain'] = gain_fold.values()

            else: #lightgbm
                self.train_preds_proba_mean[train_indices] = clf.predict_proba(x_tr, nu
                self.cv_preds_proba[cv_indices] = clf.predict_proba(x_cv, num_iteration
                self.test_preds_proba_mean += clf.predict_proba(self.x_test, num_iterat

                #feature importance
                gain_fold = clf.booster_.feature_importance(importance_type='gain')
                feat_imp = pd.DataFrame()
                feat_imp['features'] = self.x_train.columns
                feat_imp['gain'] = gain_fold

            #tuning the threshold for optimal TPR and FPR from ROC Curve
            self.best_threshold_train += self.tune_threshold(self.y_train[train_indices
            #concatenating the feature importance of each fold to original df
            self.feature_importance = pd.concat([self.feature_importance, feat_imp], ax

            if self.save_model:
                #saving the model into a pickle file
                with open(f'./data/clf_{booster}_fold_{fold_number}_model_{pickle_name}
                    pickle.dump(clf, f)

        #mean feature importance averaged over all folds
        self.feature_importance = self.feature_importance.groupby('features', as_index
        #sorting the feature importance
        self.feature_importance = self.feature_importance.sort_values(by = 'gain', asce

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

    def proba_to_class(self, proba, threshold):
        '''
        Function to convert a given probability to class label based on a threshold val

        Inputs:
            self
            proba: numpy array
                Probabilities of class label = 1
            threshold: int
                Threshold probability to be considered as Positive or Negative Class La

        Returns:
            Converted Class Label
        '''

        return np.where(proba >= threshold, 1, 0)

    def tune_threshold(self, true_labels, predicted_probas):
```

```
        '''
        Function to find the optimal threshold for maximizing the TPR and minimizing th
        This is found out by using the J Statistic, which is J = TPR - FPR.
        Reference: https://machinelearningmastery.com/threshold-moving-for-imbalanced-c

        Inputs:
            self
            true_labels: numpy array or pandas series
                True Class Labels
            predicted_probas: numpy array
                Predicted Probability of Positive Class label

        Returns:
            Threshold probability.
        '''

        fpr, tpr, threshold = roc_curve(true_labels, predicted_probas)
        j_stat = tpr - fpr
        index_for_best_threshold = np.argmax(j_stat)

        return threshold[index_for_best_threshold]

    def results(self, roc_auc = True, precision_recall = True, show_conf_matrix = True,
        '''
        Function to display the final results of Train, CV and Test Dataset.

        Inputs:
            self

        Returns:
            None
        '''

        #getting the crisp class labels
        self.train_preds_class = self.proba_to_class(self.train_preds_proba_mean, self.
        self.cv_preds_class = self.proba_to_class(self.cv_preds_proba, self.best_thresh
        self.test_preds_class = self.proba_to_class(self.test_preds_proba_mean, self.be
        print("=" * 100)
        print("Train Results:")
        print(f"\nThe best selected Threshold as per the J-Statistic, which is J = TPR
        if roc_auc:
            print(f"\tROC-AUC Score = {roc_auc_score(self.y_train, self.train_preds_pro
        if precision_recall:
            print(f"\tPrecision Score = {precision_score(self.y_train, self.train_preds
            print(f"\tRecall Score = {recall_score(self.y_train, self.train_preds_class
        print("CV Results:")
        if roc_auc:
            print(f"\tROC-AUC Score = {roc_auc_score(self.y_train, self.cv_preds_proba)
        if precision_recall:
            print(f"\tPrecision Score = {precision_score(self.y_train, self.cv_preds_cl
            print(f"\tRecall Score = {recall_score(self.y_train, self.cv_preds_class)}"

        if show_conf_matrix:
            print('=' * 100)
            print("Confusion, Precision and Recall Matrix on CV data:")
            conf_mat = confusion_matrix(self.y_train, self.cv_preds_class)
            conf_mat = pd.DataFrame(conf_mat, columns = ['Predicted_0','Predicted_1'],
            plt.figure(figsize = (7,6))
            plt.title('Confusion Matrix Heatmap')
            sns.heatmap(conf_mat, annot = True, fmt = 'g', linewidth = 0.5, annot_kws =
```

```
            plt.show()

        if cv_test_distribution:
            print('=' * 100)
            print("Distribution of Original Class Labels and Predicted CV and Test Clas
            plt.figure(figsize = (20,6))
            plt.subplot(1,3,1)
            plt.title('Class Distribution of Original Dataset')
            sns.countplot(self.y_train)
            plt.subplot(1,3,2)
            plt.title('Class Distribution of predicted Class Labels on CV')
            sns.countplot(self.cv_preds_class)
            plt.subplot(1,3,3)
            plt.title('Class Distribution of predicted Test Dataset')
            sns.countplot(self.test_preds_class)
            plt.show()
        print('=' * 100)

        gc.collect()

    def feat_importances_show(self, num_features, figsize = (10,15)):
        '''
        Function to display the top most important features.

        Inputs:
            self
            num_features: int
                Number of top features importances to display
            figsize: tuple, default = (10,15)
                Size of figure to be displayed

        Returns:
            None
        '''

        plt.figure(figsize = figsize)
        sns.barplot(self.feature_importance['gain'].iloc[:num_features],
                    self.feature_importance['features'].iloc[:num_features], orient = '
        plt.title(f'Top {num_features} features as per classifier')
        plt.xlabel('Feature Importance')
        plt.ylabel('Feature Names')
        plt.grid()
        plt.show()
        print('=' * 100)

        gc.collect()
```

2. Modelling

Now, we will start the model building.

1. We will start with a random model, to create a baseline and thus compare our sensible models with these results.
2. Then we will move to Logistic Regression and Linear SVM. We are not using Kernel SVM for high time complexities. The Classifier such as Logistic Regression and Linear Regression really

perform well when the data is high dimensional.

3. We will then move to ensembles, starting with Bagging techniques such as RandomForest and ExtraTrees Classifiers. We will also use boosting Classifiers to further compare the performances. Since the bagging will be done for very deep Decision Trees, the train time complexity may be high for such a high dimensionality.
4. Since we are using AUC as our KPI for tuning the models, and it is a case of imbalanced classification, we might have to do threshold tuning for some of the models, which do not give exact probabilities. We will use ROC Curve for threshold moving.
5. Lastly, we will do stacking and blending and see if they help to improve the models further.

Note: We are handling the NaN values by replacing them with 0s only for Sklearn models. The boosting methods like XGBoost, LightGBM inherently handle NaN values as categories, and thus we don't need to explicitly impute any value to them.

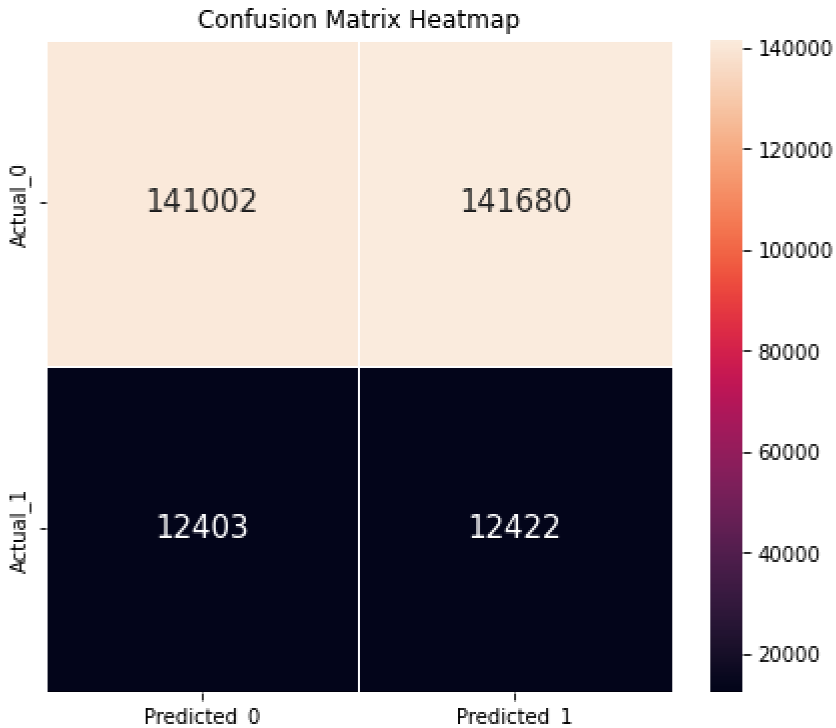
2.1 Random model

This model randomly generates a probability value between 0 to 1, for each of the datapoint. This would serve as a baseline model, and that our any sensible model should not perform worse than this.

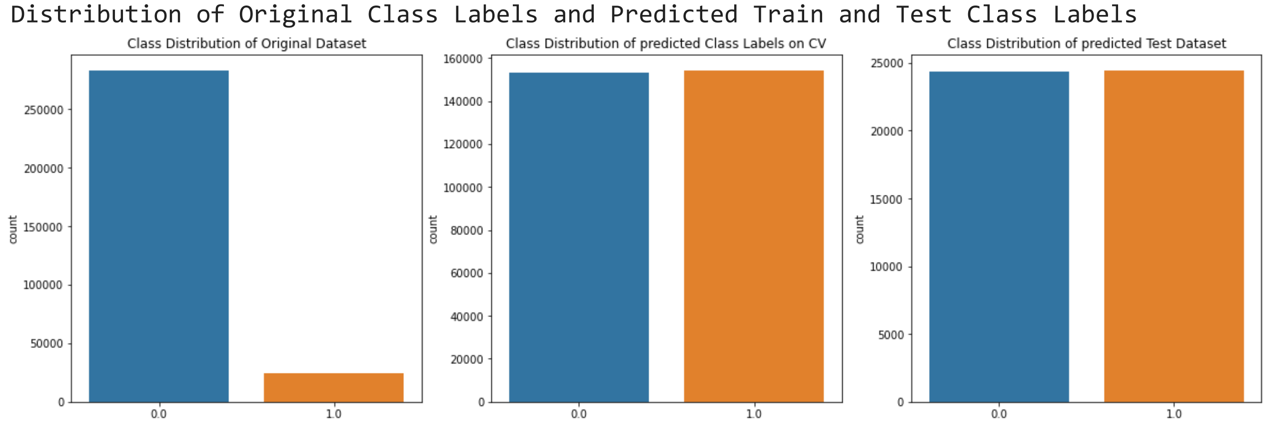
In [8]:

```
#predicted proability for train and test datapoints
predicted_proba_train = np.random.uniform(0, 1, len(train_data))
predicted_proba_test = np.random.uniform(0, 1, len(test_data))

print("=" * 100)
print("Training Dataset Results:")
print(f"\tROC-AUC Score = {roc_auc_score(target_train, predicted_proba_train)}")
print(f"\tPrecision Score = {precision_score(target_train, np.round(predicted_proba_tra
print(f"\tRecall Score = {recall_score(target_train, np.round(predicted_proba_train))}"
print('=' * 100)

print("Confusion Matrix of Training data:")
conf_mat = confusion_matrix(target_train, np.round(predicted_proba_train))
conf_mat = pd.DataFrame(conf_mat, columns = ['Predicted_0','Predicted_1'], index = ['Ac
plt.figure(figsize = (7,6))
plt.title('Confusion Matrix Heatmap')
sns.heatmap(conf_mat, annot = True, fmt = 'g', linewidth = 0.5, annot_kws = {'size' : 1
plt.show()
print('=' * 100)

print("Distribution of Original Class Labels and Predicted Train and Test Class Labels"
plt.figure(figsize = (20,6))
plt.subplot(1,3,1)
plt.title('Class Distribution of Original Dataset')
sns.countplot(target_train)
plt.subplot(1,3,2)
plt.title('Class Distribution of predicted Class Labels on CV')
sns.countplot(np.round(predicted_proba_train))
plt.subplot(1,3,3)
plt.title('Class Distribution of predicted Test Dataset')
sns.countplot(np.round(predicted_proba_test))
plt.show()
print('=' * 100)
```

```
============================================================================================
Training Dataset Results:
        ROC-AUC Score = 0.5005383162928095
        Precision Score = 0.08060894732060583
        Recall Score = 0.5003826787512589
============================================================================================
Confusion Matrix of Training data:
```




```
============================================================================================
Distribution of Original Class Labels and Predicted Train and Test Class Labels
```




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

In [69]:

```python
# #submitting the result
# pd.DataFrame({'SK_ID_CURR': skid_test, 'TARGET' : predicted_proba_test}).to_csv('Rand
# !kaggle competitions submit -c home-credit-default-risk -f Random_Model.csv -m "Rando
# print('Successfully submitted to Home Credit Default Risk')
```

```
Successfully submitted to Home Credit Default Risk
'kaggle' is not recognized as an internal or external command,
```

```
operable program or batch file.
```

In [10]:
```
random_model_df = pd.DataFrame({'SK_ID_CURR': skid_test, 'TARGET' : predicted_proba_tes
random_model_df['SK_ID_CURR'] = random_model_df['SK_ID_CURR'].astype(int)
```

In [11]:
```
random_model_df.dtypes
```

Out[11]:
```
SK_ID_CURR      int32
TARGET        float64
dtype: object
```

In [12]:
```
random_model_df.to_csv('./submissions/Random_Model.csv',index = False)
```

```
    Private Score      Public Score
```

title

- From the above Confusion Matrix, we see that it predicts almost equal numbers of Positive and Negative Class Labels. This implies the complete randomness of the model.
- From the Distributions of predicted Class Labels too, we see this behaviour.
- The AUC for the random model on training dataset comes out to be close to 0.5, and the F1-Score close to 0.14.
- The similar AUC is seen from the Test Dataset too, as seen from above submission.
- This sets a baseline for all the metrics here.

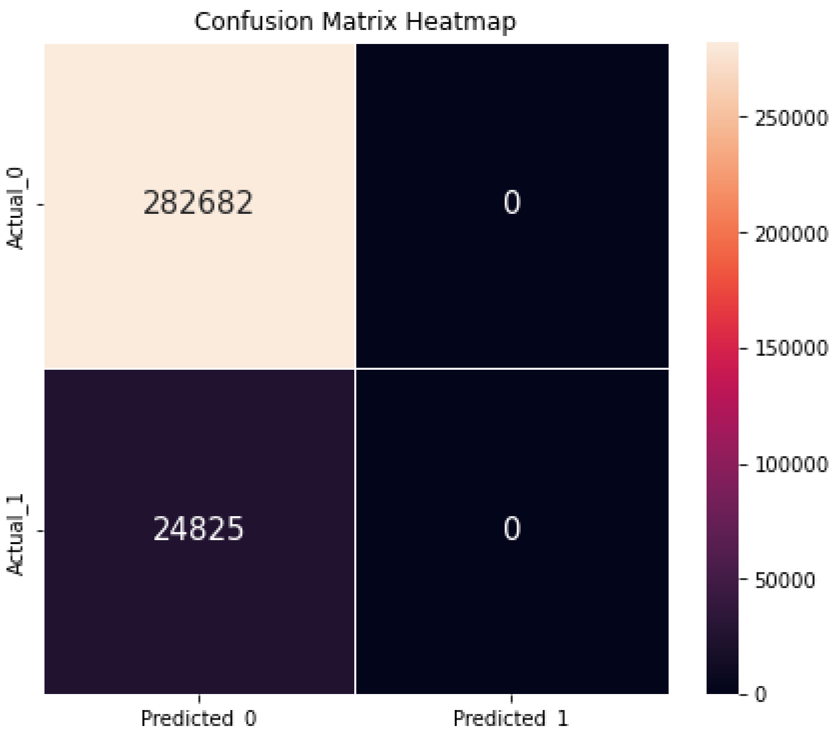
2.2 Dominant class model

1. In this model, we will predict everything as the dominant Class, which is Class Label 0 (non-defaulter) in our case here. We will see the metrics for this model too, and again our sensible models need to perform better than this.
2. Since we are calculating the ROC-AUC value as well, so we will randomly predict a probability between 0 to 0.5 with a threshold of 0.5, so that all points get classified as 0.

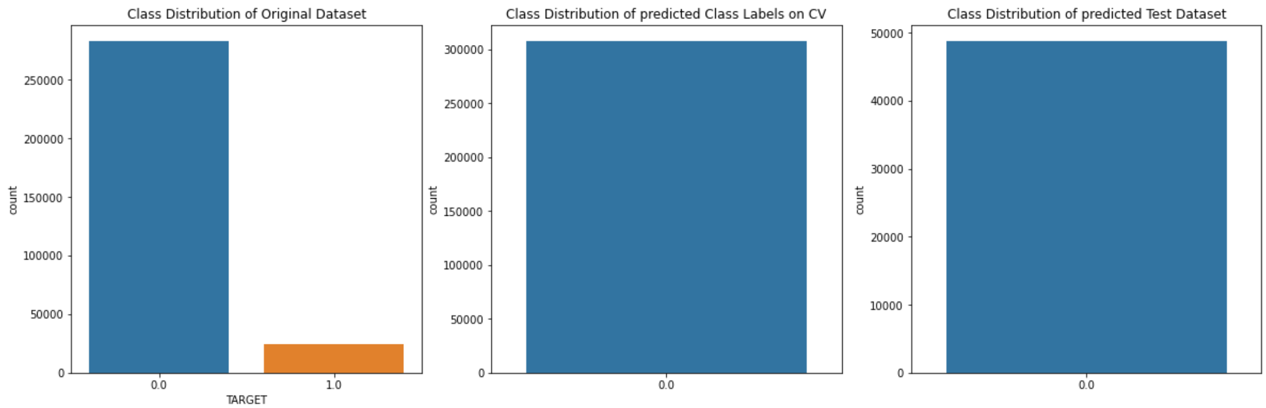
In [13]:
```
#predicting everything with probability less than 0.5, i.e majority class
predicted_proba_train = np.random.uniform(0, 0.5, len(train_data))
predicted_proba_test = np.random.uniform(0, 0.5, len(test_data))

print("=" * 100)
print("Training Dataset Results:")
print(f"\tROC-AUC Score = {roc_auc_score(target_train, predicted_proba_train)}")
print(f"\tPrecision Score = {precision_score(target_train, np.round(predicted_proba_tra
print(f"\tRecall Score = {recall_score(target_train, np.round(predicted_proba_train))}"
print('=' * 100)

print("Confusion Matrix of Training data:")
conf_mat = confusion_matrix(target_train, np.round(predicted_proba_train))
conf_mat = pd.DataFrame(conf_mat, columns = ['Predicted_0','Predicted_1'], index = ['Ac
plt.figure(figsize = (7,6))
```

```
plt.title('Confusion Matrix Heatmap')
sns.heatmap(conf_mat, annot = True, fmt = 'g', linewidth = 0.5, annot_kws = {'size' : 1
plt.show()
print('=' * 100)

print("Distribution of Original Class Labels and Predicted Train and Test Class Labels"
plt.figure(figsize = (20,6))
plt.subplot(1,3,1)
plt.title('Class Distribution of Original Dataset')
sns.countplot(target_train)
plt.subplot(1,3,2)
plt.title('Class Distribution of predicted Class Labels on CV')
sns.countplot(np.round(predicted_proba_train))
plt.subplot(1,3,3)
plt.title('Class Distribution of predicted Test Dataset')
sns.countplot(np.round(predicted_proba_test))
plt.show()
print('=' * 100)
```

```
====================================================================================================
Training Dataset Results:
        ROC-AUC Score = 0.5004273653769836
        Precision Score = 0.0
        Recall Score = 0.0
====================================================================================================
Confusion Matrix of Training data:
```




```
====================================================================================================
Distribution of Original Class Labels and Predicted Train and Test Class Labels
```




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

In [14]:
```
dominant_model_df = pd.DataFrame({'SK_ID_CURR': skid_test, 'TARGET' : predicted_proba_t
dominant_model_df['SK_ID_CURR'] = dominant_model_df['SK_ID_CURR'].astype(int)
```

In [15]:
```
random_model_df.dtypes
```

Out[15]:
```
SK_ID_CURR      int32
TARGET        float64
dtype: object
```

In [16]:
```
dominant_model_df.to_csv('./submissions/Dominant_Class_Model.csv',index = False)
```

Model	Private Score	Public Score
Dominant_Class_Model	0.49054	0.50009
Random_Model	0.49735	0.48234

title

- From the above results, we observe that again the AUC remains somewhat the same, which is 0.5.
- However, the F1-Score is now 0, because we are not predicting anything as a positive class.
- The confusion matrix also shows all points as negative, i.e it has high number of false negtives, but zero false positives.

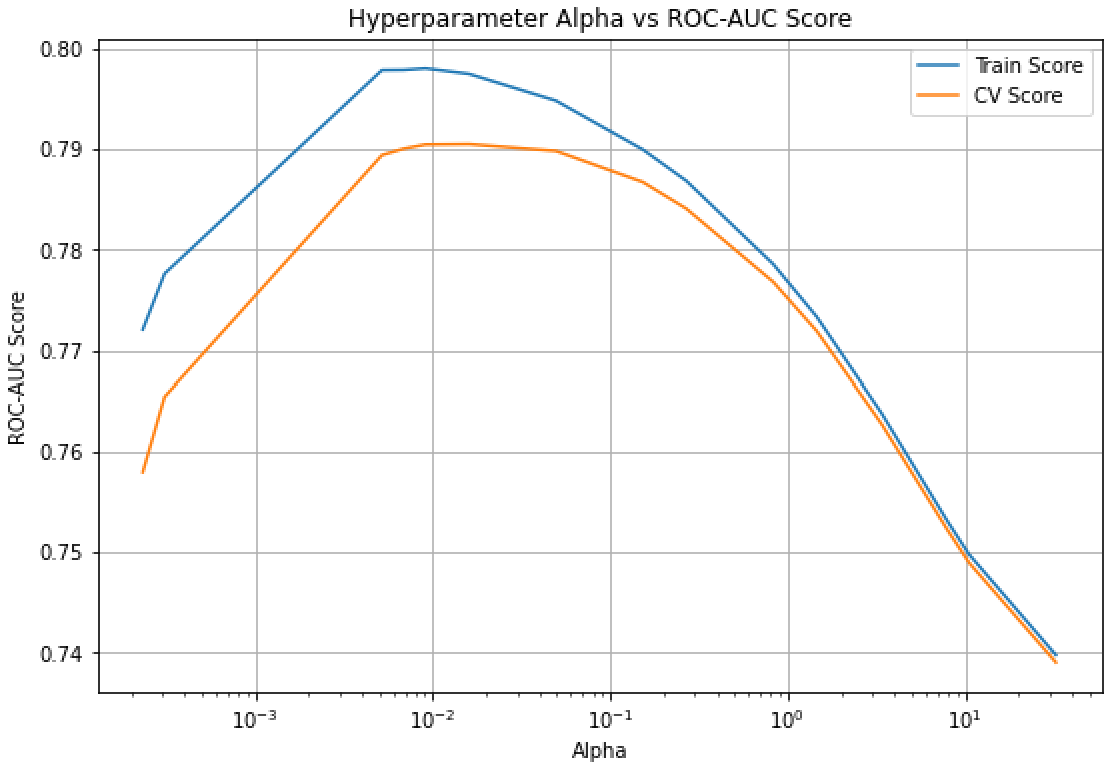
2.3 Logistic Regression L2 Regularization

- In this section, we will use SGDClassifier with log-loss and l2 penalty. We will use GridSearchCV for Hyperparameter Optimization.
- We haven't split the data into train and cv yet. We will create the cross-validation data on the fly, i.e during hyperparameter tuning.
- For overall predictions on CV, we will use Stratified K-Fold method with predictions as Out-of-fold predictions.
- Since we are optimizing the hyperparameters for AUC and also this is the case of imbalanced dataset, we will tune the threshold for best TPR and least TPR, by using J-Statistic.

In [18]:

```
params = {
    'loss' : 'log',
    'penalty' : 'l2',
    'random_state' : 98,
    'class_weight' : 'balanced',
    'n_jobs' : -1
}
clf = SGDClassifier(**params)
hyperparams = {'alpha' : np.logspace(-4,2)}

sgd_lr_l2 = modelling(clf, x_train_std, y_train, x_test_std)

#lets do randomized search cv first
#if set n_jobs=-1, will send error with memory: can't allocate xx G data
#so don't use all cores
sgd_lr_l2.random_search_cv(hyperparams, n_iter = 15, n_jobs = 6)


#visualizing the cv results
cv_results = sgd_lr_l2.tuning_results
cv_results = cv_results.sort_values('param_alpha')

#plotting the train and cv scores
plt.figure(figsize = (9,6))
plt.plot(cv_results['param_alpha'], cv_results['mean_train_score'], label = 'Train Scor
plt.plot(cv_results['param_alpha'], cv_results['mean_test_score'], label = 'CV Score')
plt.title('Hyperparameter Alpha vs ROC-AUC Score')
plt.xlabel('Alpha')
plt.ylabel('ROC-AUC Score')
plt.legend()
plt.grid()
plt.xscale('log')
plt.show()
```

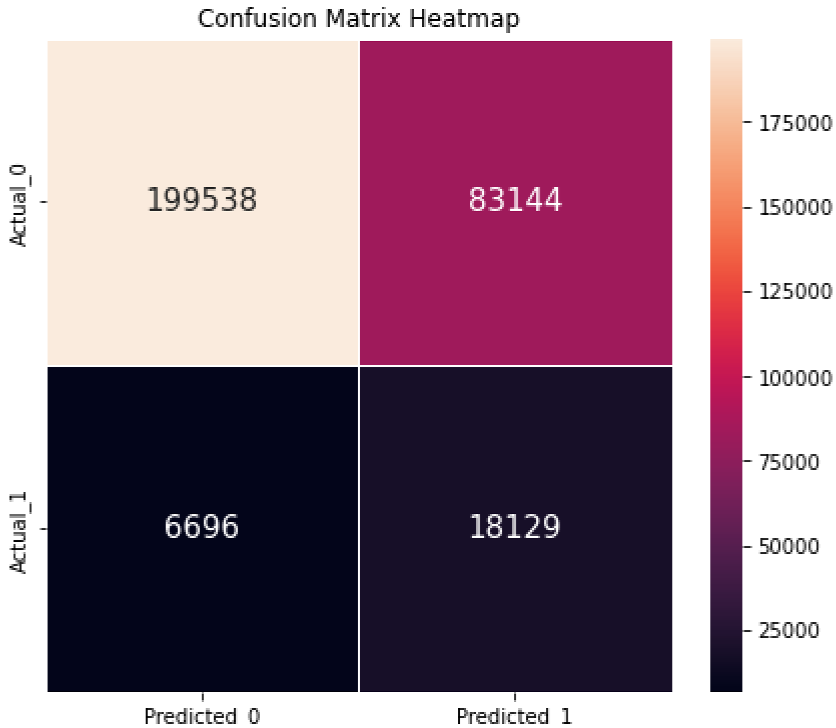
```
Doing Randomized Search CV on Classifier with 15 random initializations...
Fitting 4 folds for each of 15 candidates, totalling 60 fits
Done.
Time elapsed = 0:04:13.407590
```



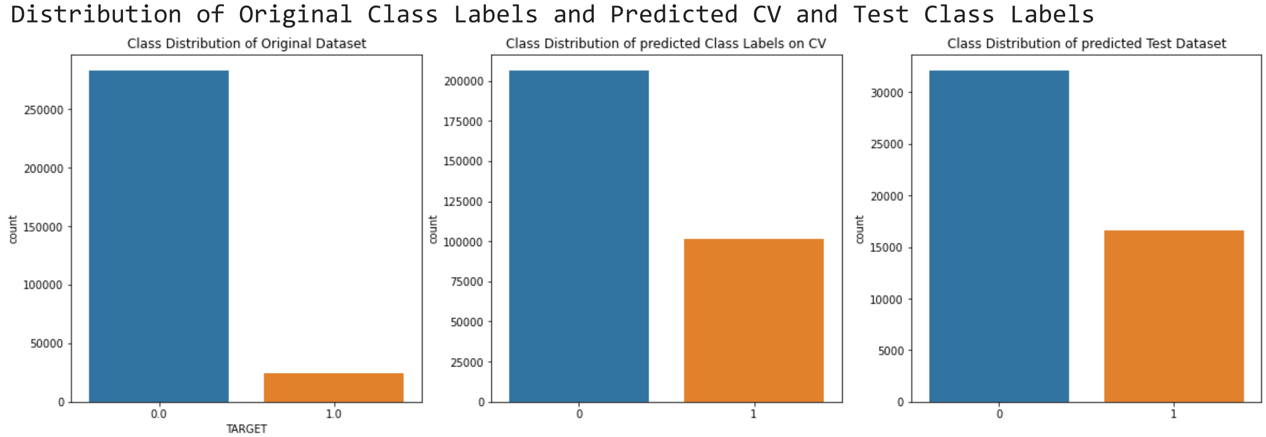
In [19]:

```
#training on best parameters
sgd_lr_l2.train_on_best_params()
#showing the results
sgd_lr_l2.results_on_best_params('linear')
#plotting feature importances
sgd_lr_l2.feat_importances_show(train_data.columns, num_features = 50)
```

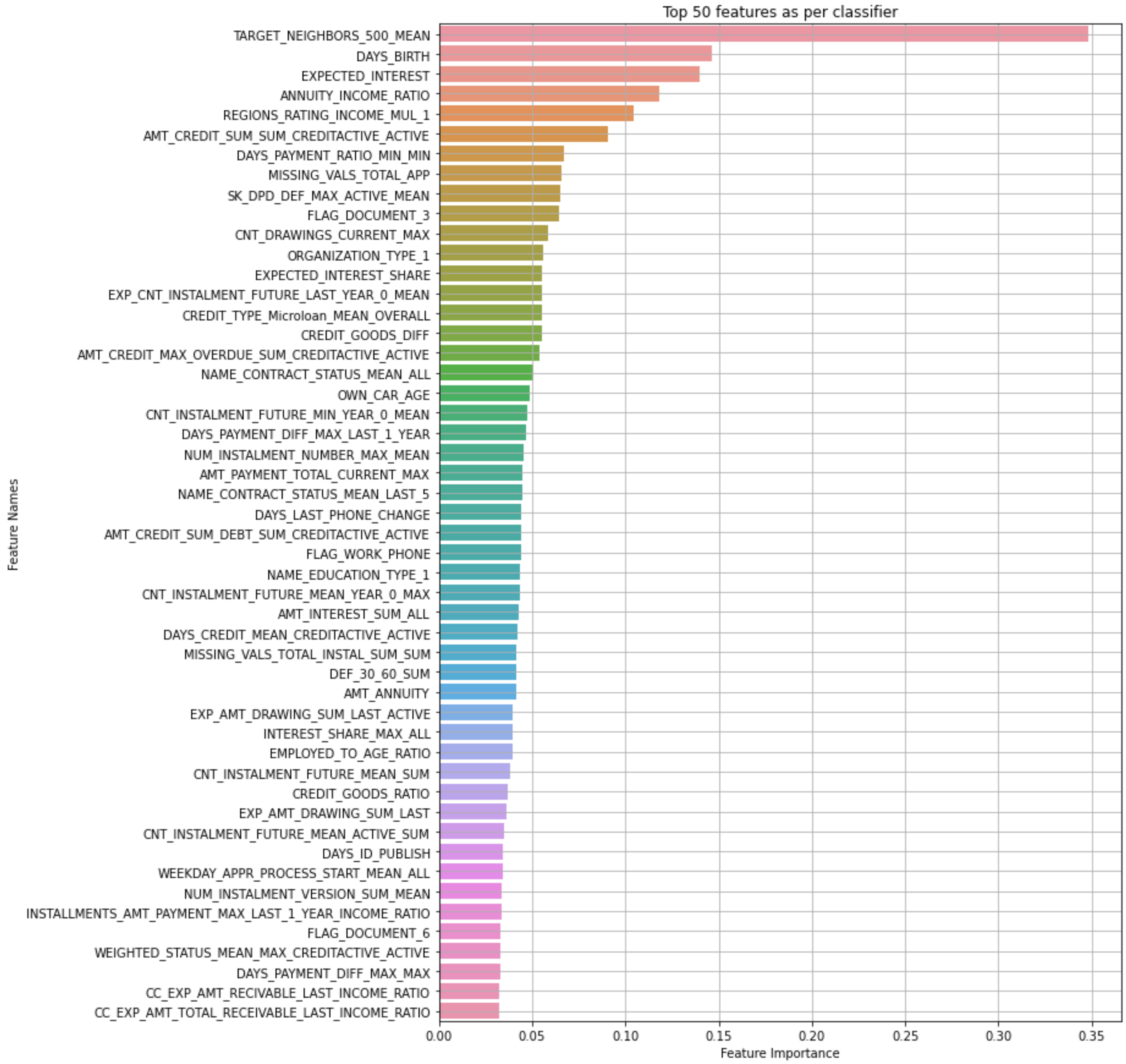
```
Fitting Classifier on best parameters

4-Fold Cross Validation
        Fitting Fold 1
        Fitting Fold 2
        Fitting Fold 3
        Fitting Fold 4
Done.
Time elapsed = 0:00:56.374000
========================================================================================
============

The best selected Threshold as per the J-Statistic, which is J = TPR - FPR, is = 0.47576
99396726839

Train Results:
        ROC-AUC Score = 0.797013753023273
        Precision Score = 0.1816067944903147
        Recall Score = 0.7424773413897281
CV Results:
        ROC-AUC Score = 0.7904871750921737
        Precision Score = 0.17901118758208012
        Recall Score = 0.7302719033232629
========================================================================================
============
Confusion Matrix of CV data:
```




```
============================================================================================================
Distribution of Original Class Labels and Predicted CV and Test Class Labels
```




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




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

In [20]:

```
LogReg_l2_model_df = pd.DataFrame({'SK_ID_CURR': skid_test, 'TARGET' : sgd_lr_l2.test_p
LogReg_l2_model_df['SK_ID_CURR'] = LogReg_l2_model_df['SK_ID_CURR'].astype(int)
LogReg_l2_model_df.to_csv('./submissions/SGD_LR_L2_penalty.csv',index = False)
```

Model	Private Score	Public Score
SGD_LR_L2_penalty	0.77894	0.78355

title

In [21]:

```
with open('./data/SGD_LR_L2_Penalty_CV_Preds.pkl', 'wb') as f:
    pickle.dump(sgd_lr_l2.cv_preds_probas, f)
with open('./data/SGD_LR_L2_Penalty_Test_Preds.pkl', 'wb') as f:
    pickle.dump(sgd_lr_l2.test_preds_probas, f)
with open('./data/SGD_LR_L2_Penalty_Model.pkl', 'wb') as f:
    pickle.dump(sgd_lr_l2.best_model, f)
```

Results Discussion

1. We first tuned the hyperparameter 'alpha' for Logistic Regression using RandomizedSearchCV, with 4-fold Cross-Validation.
2. We then used the best obtained model to fit on the training dataset.
    - From the AUC Scores, we see that the Train and CV AUC Scores are very much close to each other. This implies not much of an overfit.
    - The test AUC as per Kaggle Comes out to be a little less than CV AUC. There isn't a big difference between the CV and test AUC, this implies that the CV and Test dataset are very much similar.
    - The optimal threshold for decision for probability is 0.47 which is close to 0.5. This is because Logistic Regression actually returns true class probability inherently.
    - From the confusion matrix, we see that there are lots of False Positive Results. While the number of False Negatives are actually lower.
    - We see that the precision of our model is very low. However the recall value is actually good. This is what we want, i.e. no Defaulter should be missed even if some non-defaulters get classified as Defaulters, because then they may apply again, but if a Defaulter gets missed by the model, then that could cause lots of loss for the company.
    - If we look at the distribution of predicted class labels vs actual class labels, we see that there are lots more positive class labels predicted than there actually are. This again implies The low value of precision.
    - We also see that the predicted class labels among the CV and Test Datasets follow very similar distribution, which implies that the model is performing similarly on both CV and Test Data.
    - We have plotted the top 50 features as per the classifier. The highest scoring feature comes out to be the Mean of 500 Neighbors' Target values. We also see that another engineered feature, i.e. expected interest also scores quite high. This means that our generated features are actually helping in classification task.

2.4 Linear SVM

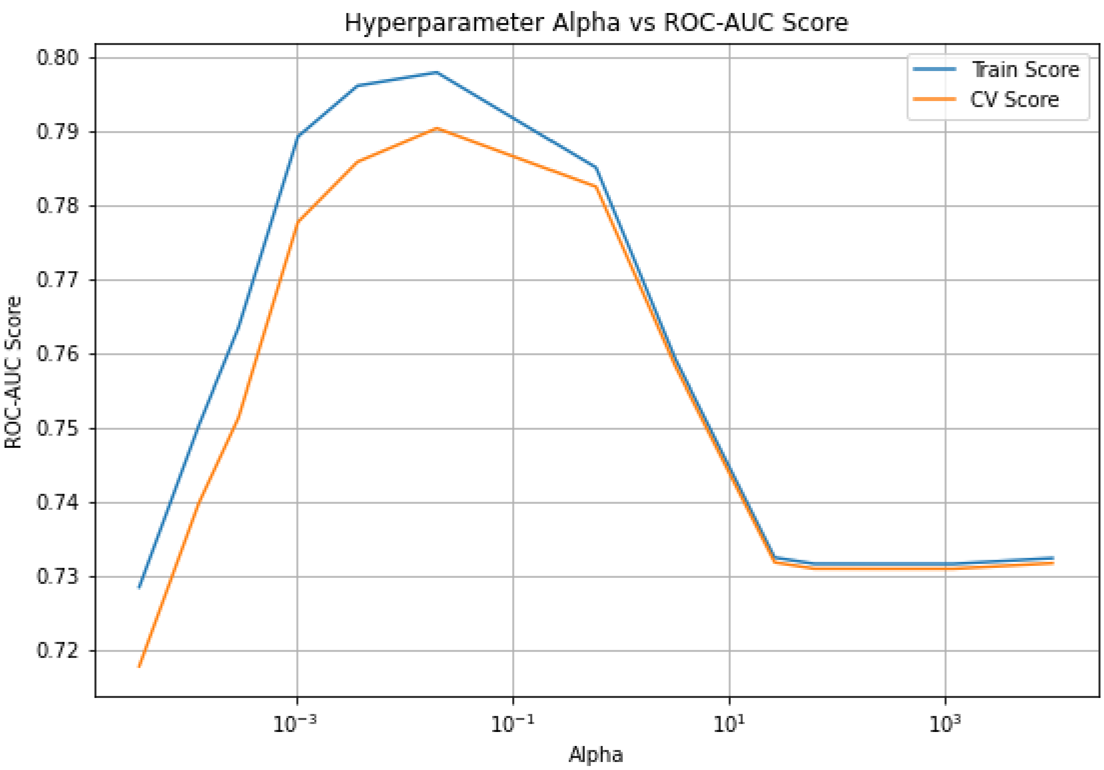
In this section, we will again train an SGDClassifier with L2 penalty but with hinge loss, for the case of Linear SVM. We won't be using Kernel SVM because of the high train time complexity, as the number of datapoints is very high in our case.

In [22]:

```
params = {
    'loss' : 'hinge',
    'class_weight' : 'balanced',
    'random_state' : 129,
    'n_jobs' : -1
}
clf = SGDClassifier(**params)
hyperparams = {
    'alpha' : np.logspace(-5,4)
}
linear_svm = modelling(clf, x_train_std, y_train, x_test_std, calibration = True)
```

```
#lets do randomized search cv first
linear_svm.random_search_cv(hyperparams, n_iter = 15, n_jobs = 6, random_state = 693)

#visualizing the cv results
cv_results = linear_svm.tuning_results
cv_results = cv_results.sort_values('param_alpha')

#plotting the train and cv scores
plt.figure(figsize = (9,6))
plt.plot(cv_results['param_alpha'], cv_results['mean_train_score'], label = 'Train Scor
plt.plot(cv_results['param_alpha'], cv_results['mean_test_score'], label = 'CV Score')
plt.title('Hyperparameter Alpha vs ROC-AUC Score')
plt.xlabel('Alpha')
plt.ylabel('ROC-AUC Score')
plt.legend()
plt.grid()
plt.xscale('log')
plt.show()
```

```
Doing Randomized Search CV on Classifier with 15 random initializations...
Fitting 4 folds for each of 15 candidates, totalling 60 fits
Done.
Time elapsed = 0:06:04.187286
```

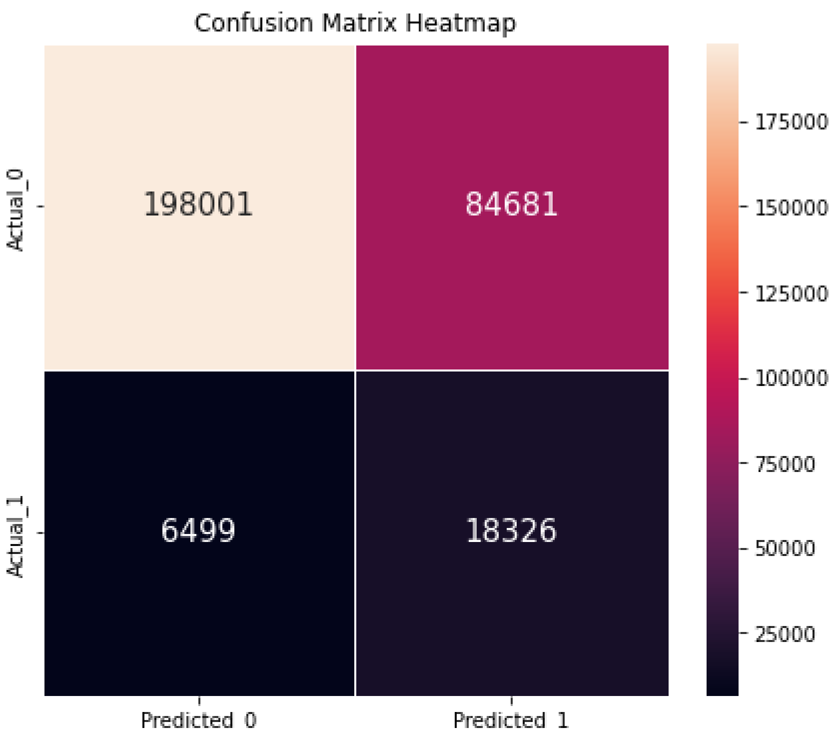


In [24]:

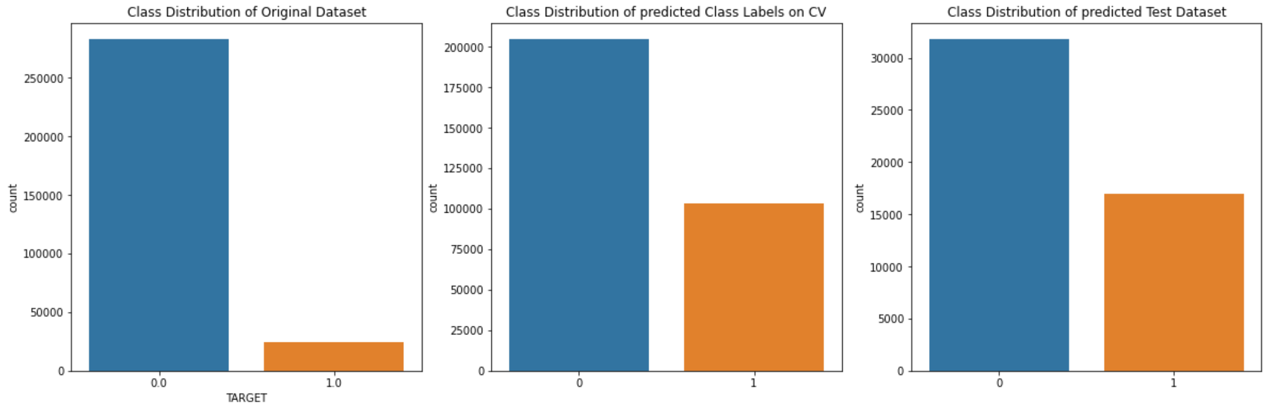
```
#training on best parameters
linear_svm.train_on_best_params()
#showing the results
linear_svm.results_on_best_params('linear')
#plotting feature importances
linear_svm.feat_importances_show(train_data.columns, num_features = 50)
```

```
Fitting Classifier on best parameters

4-Fold Cross Validation
	Fitting Fold 1
```

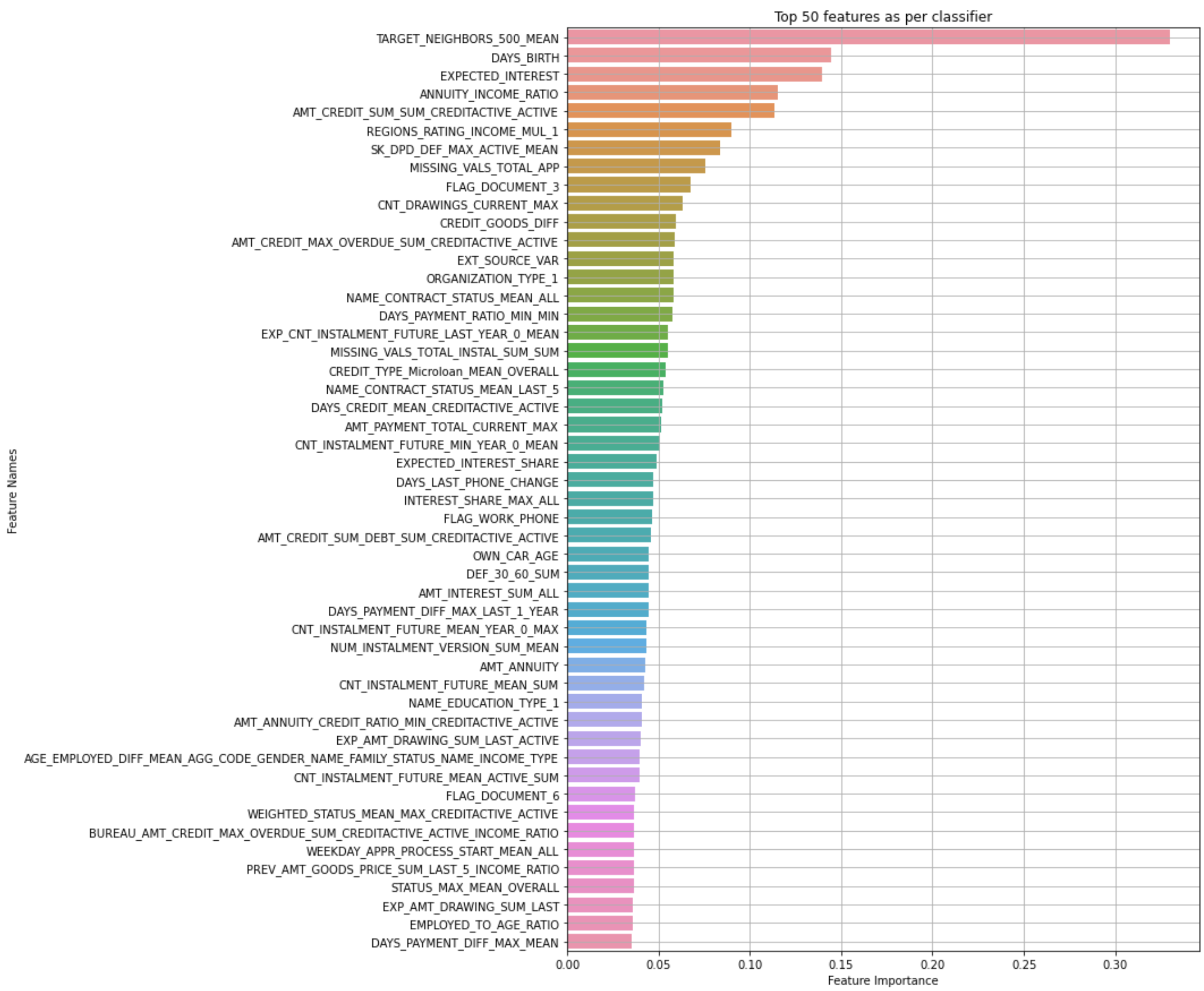
```
        Fitting Fold 2
        Fitting Fold 3
        Fitting Fold 4
Done.
Time elapsed = 0:04:01.878000
==========================================================================================
============

The best selected Threshold as per the J-Statistic, which is J = TPR - FPR, is = 0.07807
758049442576

Train Results:
        ROC-AUC Score = 0.7980842935520804
        Precision Score = 0.1796175517134918
        Recall Score = 0.7499295065458207
CV Results:
        ROC-AUC Score = 0.7911491768177968
        Precision Score = 0.1779102391099634
        Recall Score = 0.738207452165156
==========================================================================================
============
Confusion Matrix of CV data:
```




```
==========================================================================================
============
Distribution of Original Class Labels and Predicted CV and Test Class Labels
```




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




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

In [25]:

```
Linear_SVM_model_df = pd.DataFrame({'SK_ID_CURR': skid_test, 'TARGET' : linear_svm.test
Linear_SVM_model_df['SK_ID_CURR'] = Linear_SVM_model_df['SK_ID_CURR'].astype(int)
Linear_SVM_model_df.to_csv('./submissions/SGD_Linear_SVM.csv',index = False)
```

Model	Private Score	Public Score
SGD_Linear_SVM	0.77924	0.78397

title

In [26]:
```
with open('./data/SGD_Linear_SVM_CV_Preds.pkl', 'wb') as f:
    pickle.dump(linear_svm.cv_preds_probas, f)
with open('./data/SGD_Linear_SVM_Test_Preds.pkl', 'wb') as f:
    pickle.dump(linear_svm.test_preds_probas, f)
with open('./data/SGD_Linear_SVM_Model.pkl', 'wb') as f:
    pickle.dump(linear_svm.best_model, f)
```

Results Discussion

1. The results for Linear SVM without dual formulation look very similar to the Logistic Regression Results. However, there is a very slight improvement in performance.
2. Both the Train and CV AUC Scores for Linear SVM are slightly higher than that of Logistic Regression. The same can be seen from the Test AUC aswell, from the Kaggle.
3. One interesting thing to note here is that the threshold probability comes out to be as low as 0.07.
4. The Precision and Recall values are also almost the same, just slightly higher for Logisitc regression though.
5. From the Confusion Matrix, we observe that the True Positives for Linear SVM are slightly lesser than that of Logistic Regression.
6. The top 50 important features are also almost identical to the Logistic Regression.

Thus, we can conclude that the Logistic Regression and Linear SVM are behaving very similarly, as expected.

2.5 Random Forest Classifier

In this section, we will use Bagging technique and train a Random Forest Classifier. We will be using Randomized Search technique to tune some of the hyperparameters of the RandomForestClassifier.

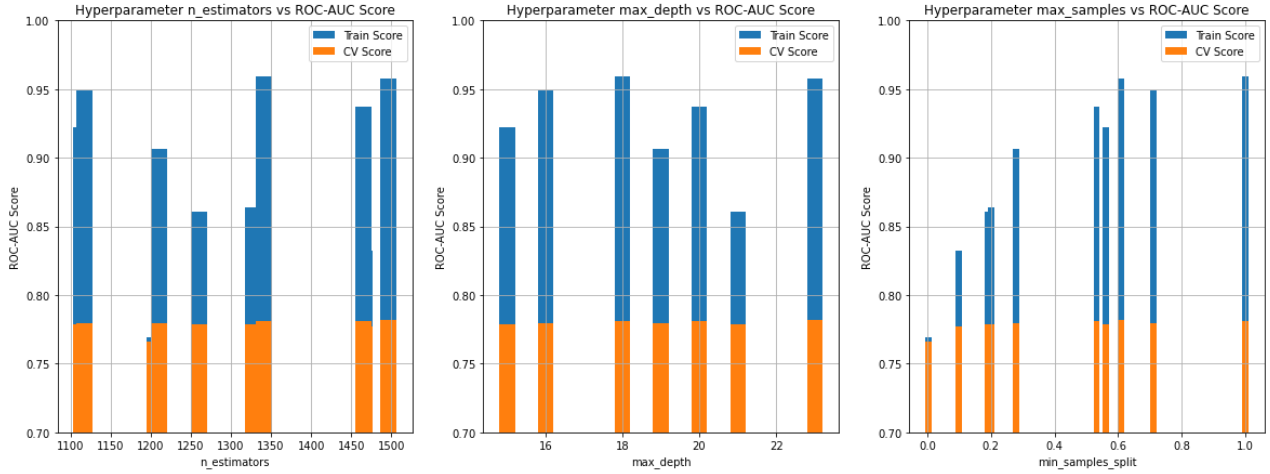
In [27]:
```
params = {
    'n_jobs' : -1,
    'random_state' : 210,
    'class_weight' : 'balanced_subsample',
    'verbose' : 0
}
clf = RandomForestClassifier(**params)

# hyperparams = {
#     'n_estimators' : list(range(500, 1500)),
#     'max_depth' : list(range(10,30)),
#     'min_samples_split' : list(range(5,50)),
#     'min_samples_leaf' : list(range(2,50)),
#     'max_samples' : uniform(0,1)
# }

#narrower search
hyperparams = {
    'n_estimators' : list(range(1100, 1500)),
    'max_depth' : list(range(15,25)),
    'min_samples_split' : list(range(30,50)),
    'min_samples_leaf' : list(range(15,30)),
```

```
    'max_samples' : uniform(0,1)
}

rf = modelling(clf, x_train_std, y_train, x_test_std)

#lets do randomized search cv first
rf.random_search_cv(hyperparams, n_iter = 10, n_jobs = -1, random_state = 4281)
```

```
Doing Randomized Search CV on Classifier with 10 random initializations...
Fitting 4 folds for each of 10 candidates, totalling 40 fits
Done.
Time elapsed = 3:55:25.514972
```

In [28]:

```
cv_results = rf.tuning_results

#lets plot the ROC-AUC for different Hyperparameters
plt.figure(figsize = (20,7))
plt.subplot(1,3,1)
cv_results = cv_results.sort_values('param_n_estimators')
plt.bar(cv_results['param_n_estimators'], cv_results['mean_train_score'], width = 20, l
plt.bar(cv_results['param_n_estimators'], cv_results['mean_test_score'], width = 20, la
plt.ylim(0.7,1)
plt.title('Hyperparameter n_estimators vs ROC-AUC Score')
plt.xlabel('n_estimators')
plt.ylabel('ROC-AUC Score')
plt.legend()
plt.grid()

plt.subplot(1,3,2)
cv_results = cv_results.sort_values('param_max_depth')
plt.bar(cv_results['param_max_depth'], cv_results['mean_train_score'], width = 0.4, lab
plt.bar(cv_results['param_max_depth'], cv_results['mean_test_score'], width = 0.4, labe
plt.ylim(0.7,1)
plt.title('Hyperparameter max_depth vs ROC-AUC Score')
plt.xlabel('max_depth')
plt.ylabel('ROC-AUC Score')
plt.legend()
plt.grid()

plt.subplot(1,3,3)
cv_results = cv_results.sort_values('param_min_samples_split')
plt.bar(cv_results['param_max_samples'], cv_results['mean_train_score'], width = 0.02,
plt.bar(cv_results['param_max_samples'], cv_results['mean_test_score'], width = 0.02, l
plt.ylim(0.7,1)
plt.title('Hyperparameter max_samples vs ROC-AUC Score')
plt.xlabel('min_samples_split')
plt.ylabel('ROC-AUC Score')

plt.legend()
plt.grid()
plt.show()
```



In [31]:

```
cv_results.to_csv('./data/rf_cv_results.csv')
# cv_results=pd.read_csv('rf_cv_results.csv')
cv_results.head()
```

Out[31]:

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_max_depth	param_max_samples
1	172.818499	18.803812	60.754003	2.092862	23	0.003868
2	2248.117265	8.870125	94.410002	2.755005	21	0.191401
5	3469.097749	13.100123	101.399250	0.738515	19	0.279677
9	4373.854249	32.071679	28.145252	4.589286	23	0.609612
8	6145.497248	30.303439	43.015752	2.631186	16	0.710215

5 rows × 23 columns

In [32]:

```
cv_results['Train_CV_Score_Difference'] = cv_results['mean_train_score'] - cv_results['

#lets look at the differences between the train and CV scores
print("Checking out the differences between Train and CV Scores for each set of hyperpa
display(cv_results[['params', 'mean_test_score', 'mean_train_score', 'Train_CV_Score_Di
```

```
Checking out the differences between Train and CV Scores for each set of hyperparameter
s:
```

	params	mean_test_score	mean_train_score	Train_CV_Score_Difference
1	{'max_depth': 23, 'max_samples': 0.00386762788...	0.766001	0.769010	0.003010

	params	mean_test_score	mean_train_score	Train_CV_Score_Difference
2	{'max_depth': 21, 'max_samples': 0.19140070731...	0.779091	0.860373	0.081281
5	{'max_depth': 19, 'max_samples': 0.27967732027...	0.779874	0.906499	0.126624
9	{'max_depth': 23, 'max_samples': 0.60961236108...	0.781735	0.957508	0.175773
8	{'max_depth': 16, 'max_samples': 0.71021498401...	0.779294	0.949437	0.170142
4	{'max_depth': 16, 'max_samples': 0.09938395385...	0.777006	0.832712	0.055706
3	{'max_depth': 18, 'max_samples': 0.99942209291...	0.781471	0.959529	0.178059
7	{'max_depth': 15, 'max_samples': 0.56001305553...	0.778704	0.921882	0.143179
6	{'max_depth': 20, 'max_samples': 0.53224896818...	0.781173	0.937533	0.156359
0	{'max_depth': 19, 'max_samples': 0.19983254792...	0.779115	0.863697	0.084582

We observe that the minimum gap between Train and CV Score is 0.003004, the second minimum is 0.55537. Because the second minimum set has higher train score, I will check the both sets.

In [34]:

```
cv_results['params'][1]
# cv_results.iloc[0]
```

Out[34]:

```
{'max_depth': 23,
 'max_samples': 0.003867627880455915,
 'min_samples_leaf': 29,
 'min_samples_split': 33,
 'n_estimators': 1205}
```

In [ ]:

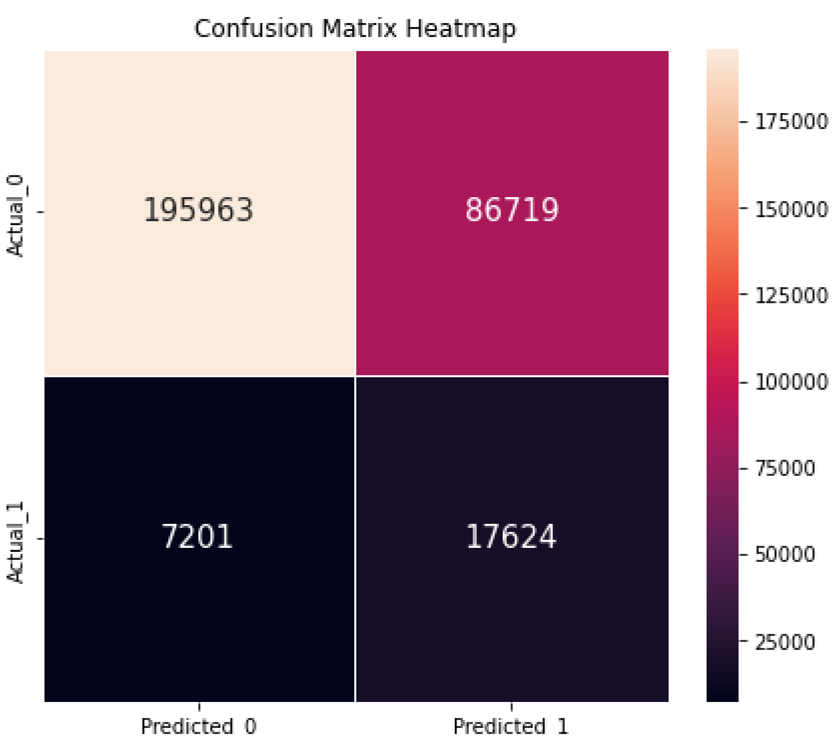
```
# #because I reload the notebook, the rf from the random search cell is lost, so I need
# params = {
#     'n_jobs' : -1,
#     'random_state' : 210,
#     'class_weight' : 'balanced_subsample',
#     'verbose' : 0
# }
# clf = RandomForestClassifier(**params)
# rf = modelling(clf, x_train_std, y_train, x_test_std)
```

In [35]:

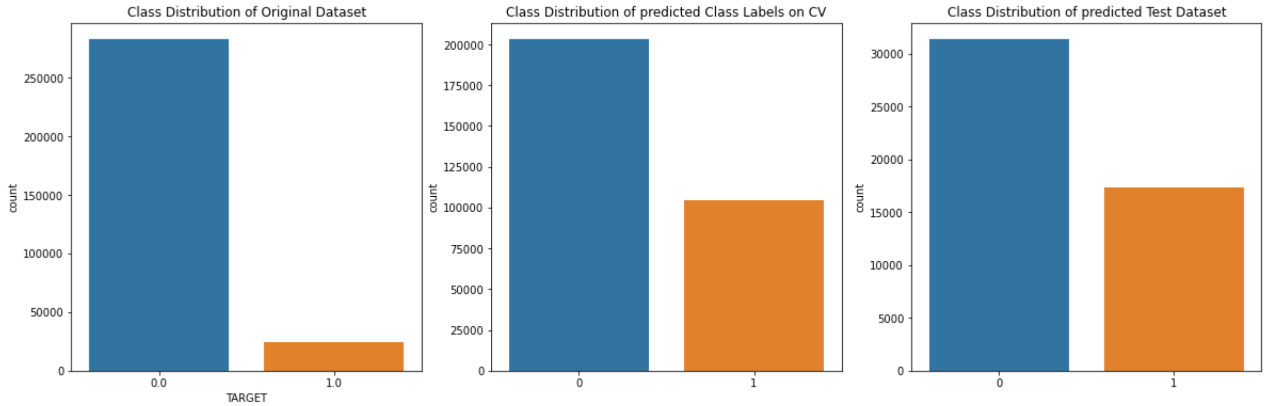
```
#reinstantiating the best model of rf class with these parameters
rf.best_model = RandomForestClassifier(class_weight = 'balanced_subsample', max_depth =
                                       max_samples = 0.0038676, min_samples_leaf = 29,
                                       min_samples_split = 33, n_estimators = 1205, n_j
                                       random_state = 210, verbose = 0)
#training on best parameters
rf.train_on_best_params()
```

```
#showing the results
rf.results_on_best_params('random_forest')
#plotting feature importances
rf.feat_importances_show(train_data.columns, num_features = 50)
```

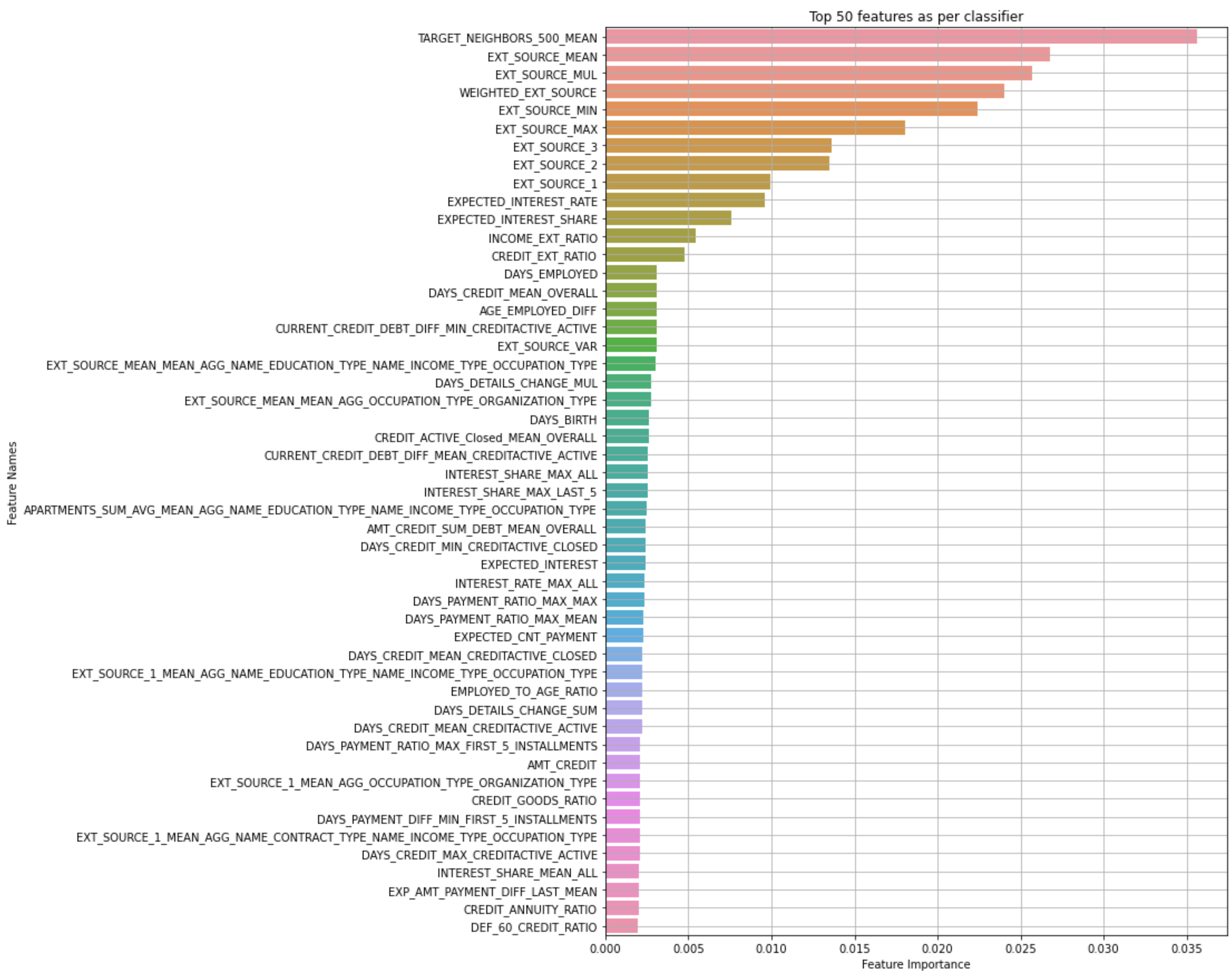
```
Fitting Classifier on best parameters

4-Fold Cross Validation
        Fitting Fold 1
        Fitting Fold 2
        Fitting Fold 3
        Fitting Fold 4
Done.
Time elapsed = 0:01:37.565001
==========================================================================================
============

The best selected Threshold as per the J-Statistic, which is J = TPR - FPR, is = 0.33303
026774676714

Train Results:
        ROC-AUC Score = 0.7696044579124289
        Precision Score = 0.1709197776937178
        Recall Score = 0.7098489425981873
CV Results:
        ROC-AUC Score = 0.7659740326603871
        Precision Score = 0.16890447849879722
        Recall Score = 0.7099295065458208
==========================================================================================
============
Confusion Matrix of CV data:
```




```
==========================================================================================
============
Distribution of Original Class Labels and Predicted CV and Test Class Labels
```




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




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

In [36]:

```
#saving the result
Random_Forest_model_df = pd.DataFrame({'SK_ID_CURR': skid_test, 'TARGET' : rf.test_pred
Random_Forest_model_df['SK_ID_CURR'] = Random_Forest_model_df['SK_ID_CURR'].astype(int)
Random_Forest_model_df.to_csv('./submissions/Random_Forest_cv0.csv',index = False)
```

In [ ]:

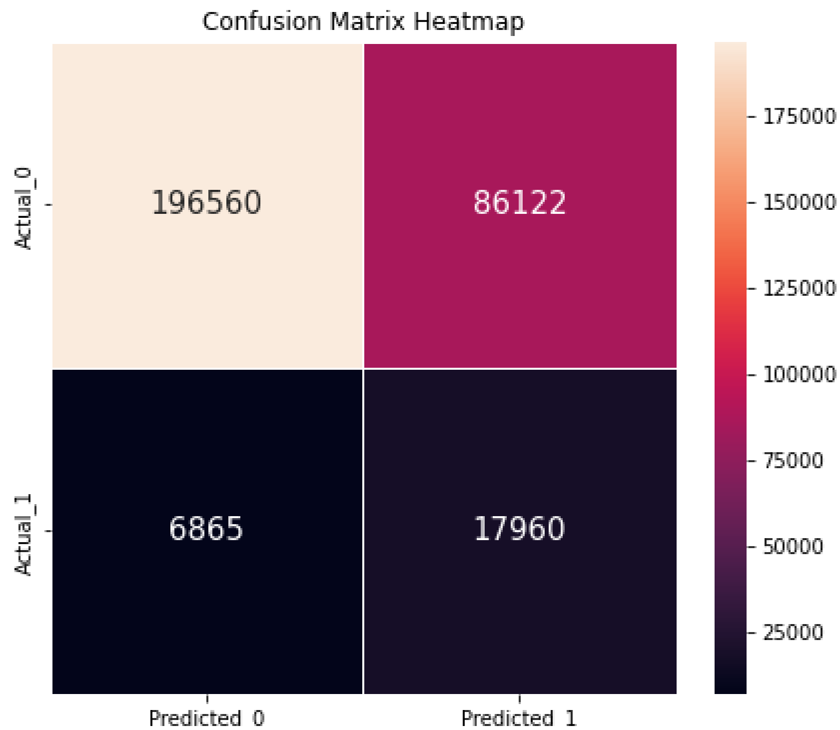
```
cv_results['params'][5]
```

In [38]:

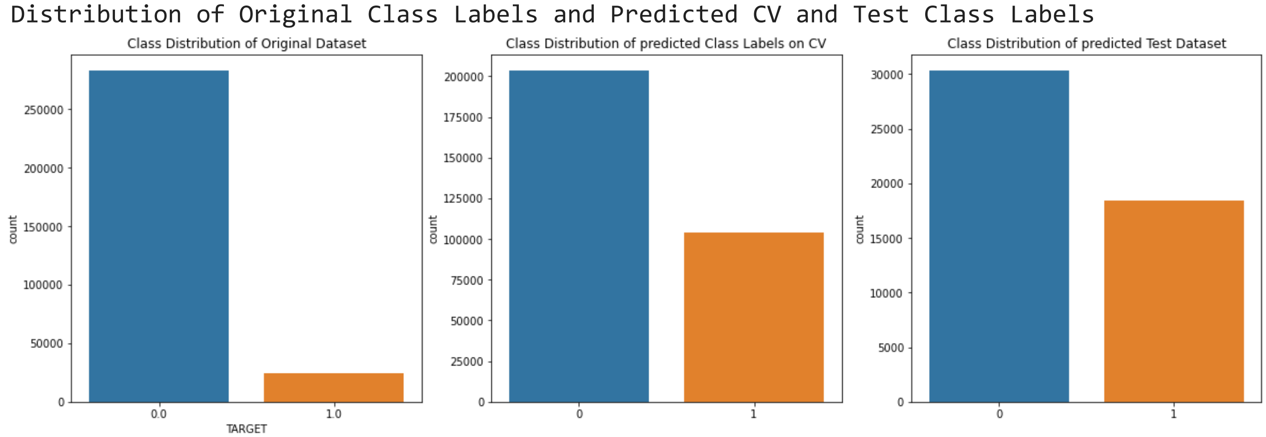
```
#reinstantiating the best model of rf class with these parameters
```

```
rf.best_model = RandomForestClassifier(class_weight = 'balanced_subsample', max_depth =
                                       max_samples = 0.09938395, min_samples_leaf = 22,
                                       min_samples_split = 37, n_estimators = 1466, n_j
                                       random_state = 210, verbose = 0)
#training on best parameters
rf.train_on_best_params()
#showing the results
rf.results_on_best_params('random_forest')
#plotting feature importances
rf.feat_importances_show(train_data.columns, num_features = 50)
```

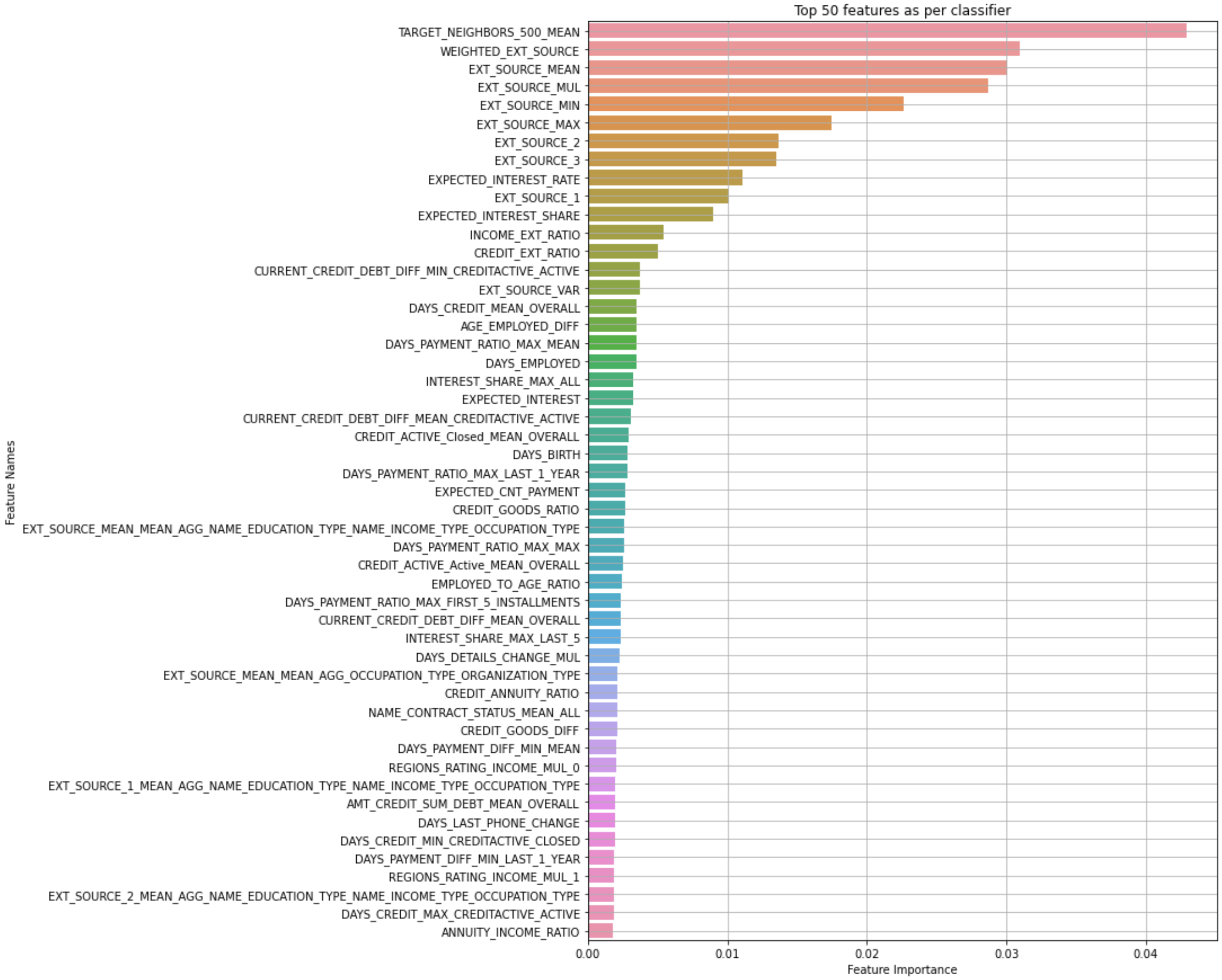
```
Fitting Classifier on best parameters

4-Fold Cross Validation
        Fitting Fold 1
        Fitting Fold 2
        Fitting Fold 3
        Fitting Fold 4
Done.
Time elapsed = 0:07:24.525001
==========================================================================================
============

The best selected Threshold as per the J-Statistic, which is J = TPR - FPR, is = 0.30622
384506817973

Train Results:
        ROC-AUC Score = 0.831314053369661
        Precision Score = 0.18841988992003625
        Recall Score = 0.8039476334340383
CV Results:
        ROC-AUC Score = 0.7769832016109426
        Precision Score = 0.17255625372302608
        Recall Score = 0.7234642497482376
==========================================================================================
============
Confusion Matrix of CV data:
```




```
================================================================================================
============
Distribution of Original Class Labels and Predicted CV and Test Class Labels
```




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



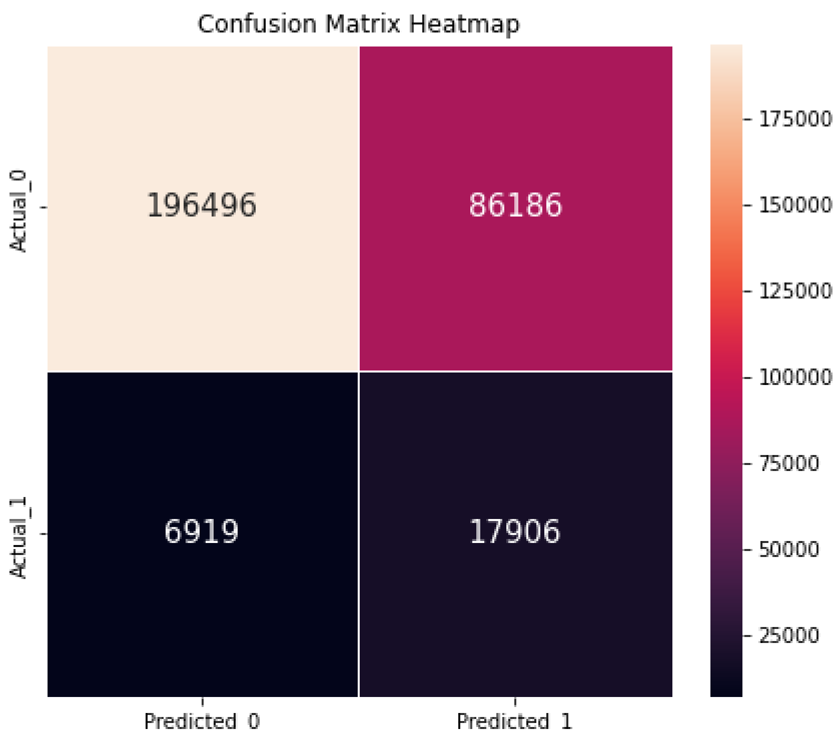

```
==========================================================================================
============
```

In [39]:
```
#saving the result
Random_Forest_model_df = pd.DataFrame({'SK_ID_CURR': skid_test, 'TARGET' : rf.test_pred
Random_Forest_model_df['SK_ID_CURR'] = Random_Forest_model_df['SK_ID_CURR'].astype(int)
Random_Forest_model_df.to_csv('./submissions/Random_Forest_cv5.csv',index = False)
```

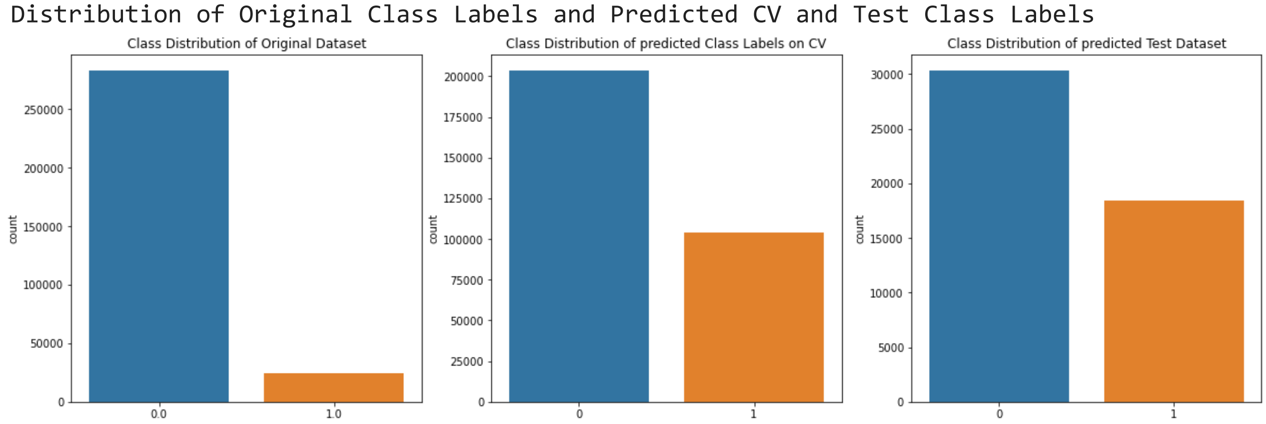
In [40]:
```
#reinstantiating the best model of rf class with these parameters
rf.best_model = RandomForestClassifier(class_weight = 'balanced_subsample', max_depth =
                                       max_samples = 0.105493, min_samples_leaf = 26,
                                       min_samples_split = 44, n_estimators = 1403, n_j
                                       random_state = 210, verbose = 0)
#training on best parameters
rf.train_on_best_params()
#showing the results
rf.results_on_best_params('random_forest')
#plotting feature importances
rf.feat_importances_show(train_data.columns, num_features = 50)
```

```
Fitting Classifier on best parameters

4-Fold Cross Validation
        Fitting Fold 1
        Fitting Fold 2
        Fitting Fold 3
```

```
        Fitting Fold 4
Done.
Time elapsed = 0:07:37.412001
========================================================================================
============

The best selected Threshold as per the J-Statistic, which is J = TPR - FPR, is = 0.31554
39666436208

Train Results:
        ROC-AUC Score = 0.8323048475402987
        Precision Score = 0.19064941546848904
        Recall Score = 0.8001208459214502
CV Results:
        ROC-AUC Score = 0.777431915513504
        Precision Score = 0.1720209045844061
        Recall Score = 0.721289023162135
========================================================================================
============
Confusion Matrix of CV data:
```

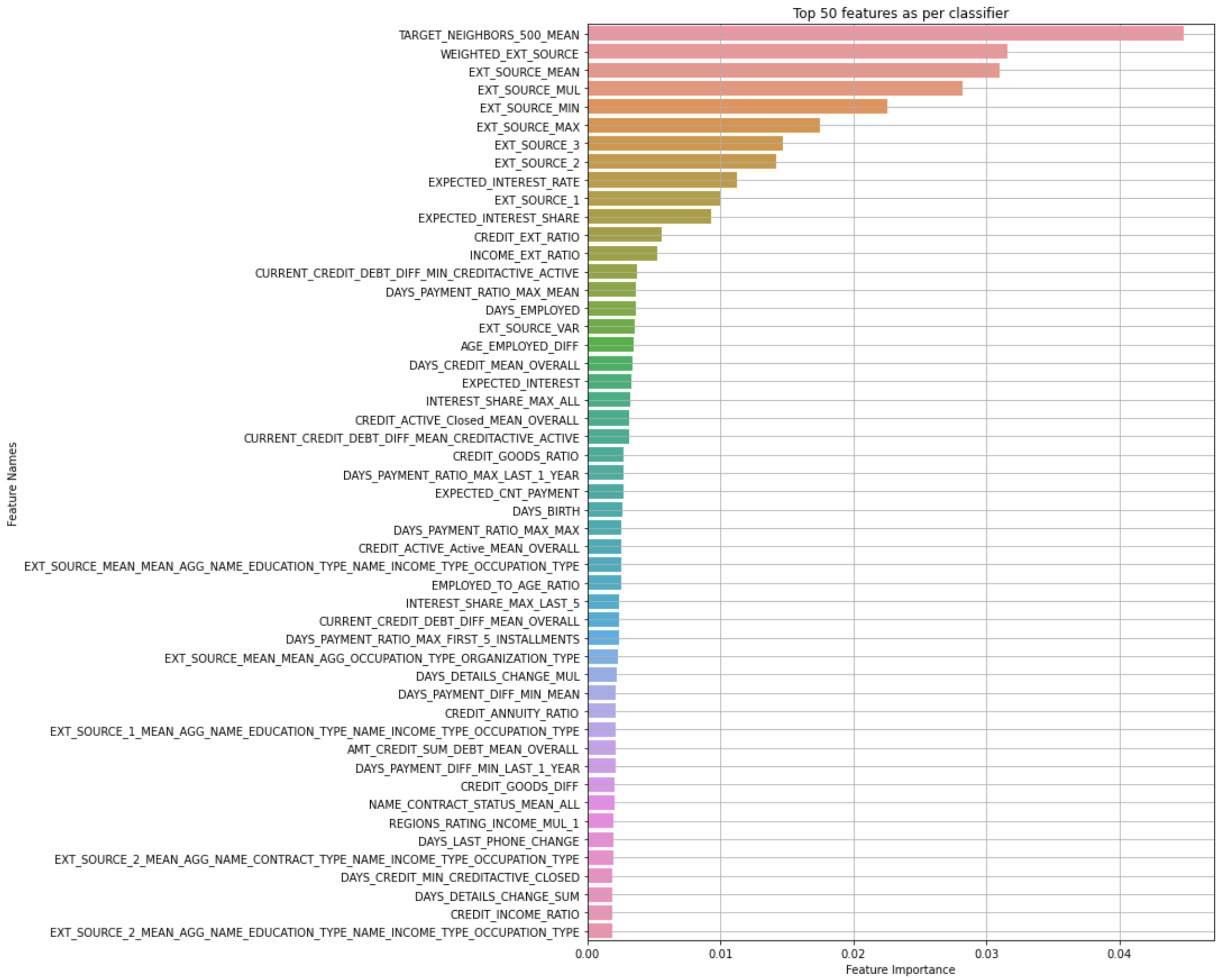



```
========================================================================================
============
Distribution of Original Class Labels and Predicted CV and Test Class Labels
```




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

============



==
============

In [41]:
```
Random_Forest_model_df = pd.DataFrame({'SK_ID_CURR': skid_test, 'TARGET' : rf.test_pred
Random_Forest_model_df['SK_ID_CURR'] = Random_Forest_model_df['SK_ID_CURR'].astype(int)
Random_Forest_model_df.to_csv('./submissions/Random_Forest_others.csv',index = False)
```

Model	Private Score	Public Score
Random_Forest_cv0	0.75370	0.76250
Random_Forest_cv5	0.76360	0.77055
Random_Forest_others	0.76396	0.77103

Random_Forest_cv0.

title

Random_Forest_cv5.

title

Random_Forest_others.

title

In [42]:
```
with open('./data/Random_Forest_CV_Preds.pkl', 'wb') as f:
    pickle.dump(rf.cv_preds_probas, f)

with open('./data/Random_Forest_Test_Preds.pkl', 'wb') as f:
    pickle.dump(rf.test_preds_probas, f)

with open('./data/Random_Forest_Model.pkl', 'wb') as f:
    pickle.dump(rf.best_model, f)
```

2.6 ExtraTreesClassifier

This classifier is very much similar to the RandomForestClassifier, except that it only looks at only some random values for Information Gain while splitting the data for numeric features.

In [44]:
```
params = {
    'n_jobs' : 6,
    'bootstrap' : True,
    'random_state' : 210,
    'class_weight' : 'balanced_subsample',
    'verbose' : 0
}
clf = ExtraTreesClassifier(**params)
# hyperparams = {
#     'n_estimators' : list(range(500, 1500)),
#     'max_depth' : list(range(10,30)),
#     'min_samples_split' : list(range(5,20)),
#     'min_samples_leaf' : list(range(2,20)),
#     'max_samples' : uniform(0,1)
# }

hyperparams = {
    'n_estimators' : list(range(800, 1200)),
    'max_depth' : list(range(15,30)),
    'min_samples_split' : list(range(5,20)),
    'min_samples_leaf' : list(range(10,20)),
    'max_samples' : uniform(0,1)
}

extra_trees = modelling(clf, x_train_std, y_train, x_test_std)

#lets do randomized search cv first
extra_trees.random_search_cv(hyperparams, n_iter = 10, n_jobs = 6, random_state = 4281)
```

```
Doing Randomized Search CV on Classifier with 10 random initializations...
Fitting 4 folds for each of 10 candidates, totalling 40 fits
exception calling callback for <Future at 0x2909d645bb0 state=finished raised Terminated
WorkerError>
Traceback (most recent call last):
  File "D:\chengxu\Anaconda\lib\site-packages\joblib\externals\loky\_base.py", line 625,
in _invoke_callbacks
    callback(self)
  File "D:\chengxu\Anaconda\lib\site-packages\joblib\parallel.py", line 359, in __call__
    self.parallel.dispatch_next()
  File "D:\chengxu\Anaconda\lib\site-packages\joblib\parallel.py", line 794, in dispatch
_next
    if not self.dispatch_one_batch(self._original_iterator):
```

```
  File "D:\chengxu\Anaconda\lib\site-packages\joblib\parallel.py", line 861, in dispatch
_one_batch
    self._dispatch(tasks)
  File "D:\chengxu\Anaconda\lib\site-packages\joblib\parallel.py", line 779, in _dispatc
h
    job = self._backend.apply_async(batch, callback=cb)
  File "D:\chengxu\Anaconda\lib\site-packages\joblib\_parallel_backends.py", line 531, i
n apply_async
    future = self._workers.submit(SafeFunction(func))
  File "D:\chengxu\Anaconda\lib\site-packages\joblib\externals\loky\reusable_executor.p
y", line 177, in submit
    return super(_ReusablePoolExecutor, self).submit(
  File "D:\chengxu\Anaconda\lib\site-packages\joblib\externals\loky\process_executor.p
y", line 1115, in submit
    raise self._flags.broken
joblib.externals.loky.process_executor.TerminatedWorkerError: A worker process managed b
y the executor was unexpectedly terminated. This could be caused by a segmentation fault
while calling the function or by an excessive memory usage causing the Operating System
to kill the worker.


---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_23696/2291538437.py in <module>
     26 
     27 #lets do randomized search cv first
---> 28 extra_trees.random_search_cv(hyperparams, n_iter = 10, n_jobs = 6, random_state
= 4281)

~\AppData\Local\Temp/ipykernel_23696/3645602195.py in random_search_cv(self, hyperparams
_dict, n_iter, verbose, n_jobs, random_state)
     83                                     cv = self.kfolds, return_train_score = True,
 verbose = 2,
     84                                     n_jobs = n_jobs, random_state = random_state)
---> 85         rscv.fit(self.x_train, self.y_train)
     86         if verbose:
     87             print("Done.")

D:\chengxu\Anaconda\lib\site-packages\sklearn\utils\validation.py in inner_f(*args, **kw
args)
     61             extra_args = len(args) - len(all_args)
     62             if extra_args <= 0:
---> 63                 return f(*args, **kwargs)
     64 
     65             # extra_args > 0

D:\chengxu\Anaconda\lib\site-packages\sklearn\model_selection\_search.py in fit(self, X,
y, groups, **fit_params)
    839                 return results
    840 
--> 841             self._run_search(evaluate_candidates)
    842 
    843             # multimetric is determined here because in the case of a callable

D:\chengxu\Anaconda\lib\site-packages\sklearn\model_selection\_search.py in _run_search
(self, evaluate_candidates)
   1631     def _run_search(self, evaluate_candidates):
   1632         """Search n_iter candidates from param_distributions"""
-> 1633         evaluate_candidates(ParameterSampler(
   1634             self.param_distributions, self.n_iter,
   1635             random_state=self.random_state))
```

```
D:\chengxu\Anaconda\lib\site-packages\sklearn\model_selection\_search.py in evaluate_candidates(candidate_params, cv, more_results)
    793                                 n_splits, n_candidates, n_candidates * n_splits))
    794 
--> 795             out = parallel(delayed(_fit_and_score)(clone(base_estimator),
    796                                                    X, y,
    797                                                    train=train, test=test,

D:\chengxu\Anaconda\lib\site-packages\joblib\parallel.py in __call__(self, iterable)
   1054 
   1055             with self._backend.retrieval_context():
-> 1056                 self.retrieve()
   1057             # Make sure that we get a last message telling us we are done
   1058             elapsed_time = time.time() - self._start_time

D:\chengxu\Anaconda\lib\site-packages\joblib\parallel.py in retrieve(self)
    933             try:
    934                 if getattr(self._backend, 'supports_timeout', False):
--> 935                     self._output.extend(job.get(timeout=self.timeout))
    936                 else:
    937                     self._output.extend(job.get())

D:\chengxu\Anaconda\lib\site-packages\joblib\_parallel_backends.py in wrap_future_result(future, timeout)
    540         AsyncResults.get from multiprocessing."""
    541         try:
--> 542             return future.result(timeout=timeout)
    543         except CfTimeoutError as e:
    544             raise TimeoutError from e

D:\chengxu\Anaconda\lib\concurrent\futures\_base.py in result(self, timeout)
    438                     return self.__get_result()
    439 
--> 440                 self._condition.wait(timeout)
    441 
    442                 if self._state in [CANCELLED, CANCELLED_AND_NOTIFIED]:

D:\chengxu\Anaconda\lib\threading.py in wait(self, timeout)
    310         try:    # restore state no matter what (e.g., KeyboardInterrupt)
    311             if timeout is None:
--> 312                 waiter.acquire()
    313                 gotit = True
    314             else:

KeyboardInterrupt:
```

In [ ]:

2.7 XGBoost GPU

For both XGBoost and LightGBM boosting, we have too many hyperparameters to tune, and doing GridSearchCV or RandomizedSearchCV can be too expesive on such a big dataset for finding an optimal solution. That is why we will be using the Bayesian Optimization Technique to tune the hyperparameters, which works by looking at the results on previous hyperparameters while assigning new hyperparameters. It tries to model on the Cost Function which is dependent on all the hyperparameters.

Bayesian Optimization

In [45]:
```
def xgb_evaluation(max_depth, min_child_weight, gamma, subsample, colsample_bytree, col
                   colsample_bynode, reg_alpha, reg_lambda):
    '''
    Objective function for Bayesian Optimization of XGBoost's Hyperparamters. Takes the
    returns the Cross-Validation AUC as output.

    Inputs: Hyperparamters to be tuned.
        max_depth, min_child_weight, gamma, subsample, colsample_bytree, colsample_byle
        colsample_bynode, reg_alpha, reg_lambda

    Returns:
        CV ROC-AUC Score
    '''
    params = {
        'learning_rate' : 0.01,
        'n_estimators' : 10000,
        'tree_method' : 'gpu_hist',
        'gpu_id' : 0,
        'max_depth' : int(round(max_depth)),
        'min_child_weight' : int(round(min_child_weight)),
        'subsample': subsample,
        'gamma' : gamma,
        'colsample_bytree' : colsample_bytree,
        'colsample_bylevel' : colsample_bylevel,
        'colsample_bynode' : colsample_bynode,
        'reg_alpha' : reg_alpha,
        'reg_lambda' : reg_lambda,
        'random_state' : 51412
    }

    #defining the Cross-Validation Strategy
    stratified_cv = StratifiedKFold(n_splits = 3, shuffle = True, random_state = 33)
    cv_preds = np.zeros(train_data.shape[0])

    #iterating over each fold, training the model, and making Out of Fold Predictions
    for train_indices, cv_indices in stratified_cv.split(train_data, target_train):

        x_tr = train_data.iloc[train_indices]
        y_tr = target_train.iloc[train_indices]
        x_cv = train_data.iloc[cv_indices]
        y_cv = target_train.iloc[cv_indices]

        xgbc = XGBClassifier(**params)
        xgbc.fit(x_tr, y_tr, eval_set= [(x_cv,y_cv)],
                 eval_metric='auc', verbose = False, early_stopping_rounds=200)

        cv_preds[cv_indices] = xgbc.predict_proba(x_cv, ntree_limit = xgbc.get_booster(
        gc.collect()

    return roc_auc_score(target_train, cv_preds)
```

In [46]:
```
#defining the optimizer and the hyperparameters along with ranges of values.
```

```
bopt_xgb =  BayesianOptimization(xgb_evaluation, {'max_depth' : (5,15),
                                                  'min_child_weight' : (5,80),
                                                  'gamma' : (0.2,1),
                                                  'subsample' : (0.5,1),
                                                  'colsample_bytree' : (0.5,1),
                                                  'colsample_bylevel' : (0.3,1),
                                                  'colsample_bynode' : (0.3,1),
                                                  'reg_alpha' : (0.001, 0.3),
                                                  'reg_lambda' : (0.001, 0.3)},
                                 random_state = 55)

bopt_xgb.maximize(n_iter = 6, init_points = 4) #cost 2h:2min:49sec
```

```
|   iter    |  target   | colsam... | colsam... | colsam... |   gamma   | max_depth | mi
n_ch... | reg_alpha | reg_la... | subsample |
-------------------------------------------------------------------------------------------
---------------------------------------------
|  1        |  0.8025   |  0.3652   |  0.9802   |  0.7419   |  0.394    |  10.31    |  2
6.42    |  0.2589   |  0.01329  |  0.5542   |
|  2        |  0.8016   |  0.837    |  0.336    |  0.8879   |  0.2073   |  11.18    |  6
6.4     |  0.2697   |  0.2957   |  0.7484   |
|  3        |  0.8017   |  0.5466   |  0.907    |  0.6984   |  0.7149   |  5.205    |  6
5.61    |  0.1295   |  0.168    |  0.8896   |
|  4        |  0.8024   |  0.5818   |  0.9356   |  0.825    |  0.7818   |  12.8     |  3
1.3     |  0.006878 |  0.1082   |  0.6516   |
|  5        |  0.8027   |  0.8305   |  0.4952   |  0.8452   |  0.9234   |  14.86    |
 5.011    |  0.2194   |  0.2374   |  0.5385   |
|  6        |  0.806    |  0.6235   |  0.588    |  0.5098   |  0.4871   |  5.001    |
 5.005    |  0.2844   |  0.2423   |  0.5224   |
|  7        |  0.8056   |  0.3      |  0.3      |  0.5      |  0.2      |  5.0      |  1
0.6     |  0.001    |  0.2608   |  1.0      |
|  8        |  0.8022   |  0.5324   |  1.0      |  0.7012   |  0.9499   |  5.0      |  4
6.34    |  0.3      |  0.3      |  0.7665   |
|  9        |  0.806    |  1.0      |  1.0      |  1.0      |  1.0      |  5.0      |
 7.365    |  0.001    |  0.001    |  0.5      |
|  10       |  0.8051   |  1.0      |  0.3      |  1.0      |  1.0      |  5.0      |
 5.0      |  0.001    |  0.3      |  1.0      |
===========================================================================================
=============================================
```

In [47]:

```
#extracting the best parameters
target_values = []
for result in bopt_xgb.res:
    target_values.append(result['target'])
    if result['target'] == max(target_values):
        best_params = result['params']

print("Best Hyperparameters for XGBoost are:\n")
print(best_params)
```

```
Best Hyperparameters for XGBoost are:

{'colsample_bylevel': 1.0, 'colsample_bynode': 1.0, 'colsample_bytree': 1.0, 'gamma': 1.
0, 'max_depth': 5.0, 'min_child_weight': 7.364705574739114, 'reg_alpha': 0.001, 'reg_lam
bda': 0.001, 'subsample': 0.5}
```

In [51]:

```
#optimizer.max can replace above method
bopt_xgb.max
```

Out[51]:
```
{'target': 0.8060150055418316,
 'params': {'colsample_bylevel': 1.0,
  'colsample_bynode': 1.0,
  'colsample_bytree': 1.0,
  'gamma': 1.0,
  'max_depth': 5.0,
  'min_child_weight': 7.364705574739114,
  'reg_alpha': 0.001,
  'reg_lambda': 0.001,
  'subsample': 0.5}}
```

In [56]:
```
params_custom = {
        'learning_rate' : 0.01,
        'n_estimators' : 10000,
        'tree_method' : 'gpu_hist',
        'gpu_id' : 0,
        'random_state' : 51412,
}
params={**params_custom, **best_params, 'max_depth': 5} #max_depth must be int
params
```

Out[56]:
```
{'learning_rate': 0.01,
 'n_estimators': 10000,
 'tree_method': 'gpu_hist',
 'gpu_id': 0,
 'random_state': 51412,
 'colsample_bylevel': 1.0,
 'colsample_bynode': 1.0,
 'colsample_bytree': 1.0,
 'gamma': 1.0,
 'max_depth': 5,
 'min_child_weight': 7.364705574739114,
 'reg_alpha': 0.001,
 'reg_lambda': 0.001,
 'subsample': 0.5}
```

Training on Optimized Hyper-parameters

In [57]:
```
#training on optimized hyperparameters
# params = {
#          'learning_rate' : 0.01,
#          'n_estimators' : 10000,
#          'tree_method' : 'gpu_hist',
#          'gpu_id' : 0,
#          'max_depth' : 5,
#          'min_child_weight' : 80,
#          'subsample': 0.7748690361169297,
#          'gamma' : 0.6992119556036338,
#          'colsample_bytree' : 0.7132522544384412,
#          'colsample_bylevel' : 0.5699318202360774,
#          'colsample_bynode' : 0.7635378153900353,
#          'reg_alpha' : 0.16095798232508268,
#          'reg_lambda' : 0.035645255471151 41,
#          'random_state' : 51412
#      }

xgb_boosting = Boosting(train_data, target_train, test_data, params, random_state = 98,
xgb_boosting.train(booster = 'xgboost')
```

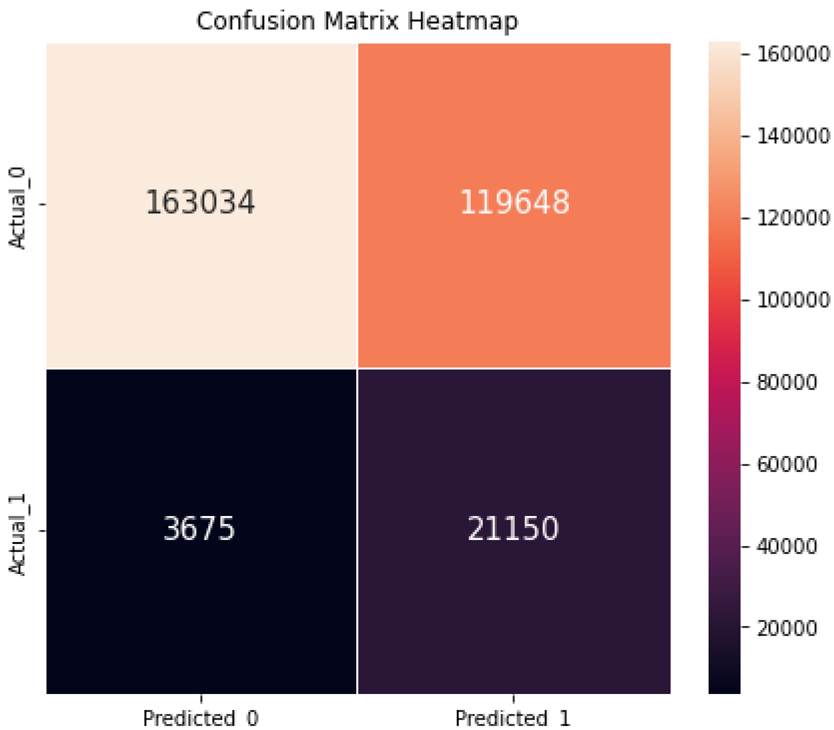
```
Fitting the xgboost on Training Data with 3 fold cross validation, and using Out-Of-Fold
s Predictions for Cross-Validation

        Fold Number 1

[0]     validation_0-auc:0.75488        validation_1-auc:0.75890
[400]   validation_0-auc:0.79939        validation_1-auc:0.78865
[800]   validation_0-auc:0.82469        validation_1-auc:0.80054
[1200]  validation_0-auc:0.84024        validation_1-auc:0.80461
[1600]  validation_0-auc:0.85284        validation_1-auc:0.80628
[2000]  validation_0-auc:0.86399        validation_1-auc:0.80736
[2400]  validation_0-auc:0.87376        validation_1-auc:0.80784
[2800]  validation_0-auc:0.88289        validation_1-auc:0.80815
[3200]  validation_0-auc:0.89129        validation_1-auc:0.80822
[3269]  validation_0-auc:0.89277        validation_1-auc:0.80817

        Fold Number 2

[0]     validation_0-auc:0.75779        validation_1-auc:0.75129
[400]   validation_0-auc:0.80063        validation_1-auc:0.78199
[800]   validation_0-auc:0.82523        validation_1-auc:0.79614
[1200]  validation_0-auc:0.84092        validation_1-auc:0.80062
[1600]  validation_0-auc:0.85345        validation_1-auc:0.80262
[2000]  validation_0-auc:0.86436        validation_1-auc:0.80359
[2400]  validation_0-auc:0.87426        validation_1-auc:0.80428
[2800]  validation_0-auc:0.88355        validation_1-auc:0.80457
[3200]  validation_0-auc:0.89182        validation_1-auc:0.80481
[3486]  validation_0-auc:0.89699        validation_1-auc:0.80472

        Fold Number 3

[0]     validation_0-auc:0.75739        validation_1-auc:0.74838
[400]   validation_0-auc:0.80207        validation_1-auc:0.78161
[800]   validation_0-auc:0.82674        validation_1-auc:0.79483
[1200]  validation_0-auc:0.84201        validation_1-auc:0.79919
[1600]  validation_0-auc:0.85437        validation_1-auc:0.80124
[2000]  validation_0-auc:0.86514        validation_1-auc:0.80248
[2400]  validation_0-auc:0.87469        validation_1-auc:0.80290
[2800]  validation_0-auc:0.88361        validation_1-auc:0.80312
[3200]  validation_0-auc:0.89206        validation_1-auc:0.80337
[3600]  validation_0-auc:0.89969        validation_1-auc:0.80348
[4000]  validation_0-auc:0.90687        validation_1-auc:0.80355
[4128]  validation_0-auc:0.90899        validation_1-auc:0.80357
Done.
Time elapsed = 0:10:31.412189
```

In [58]:

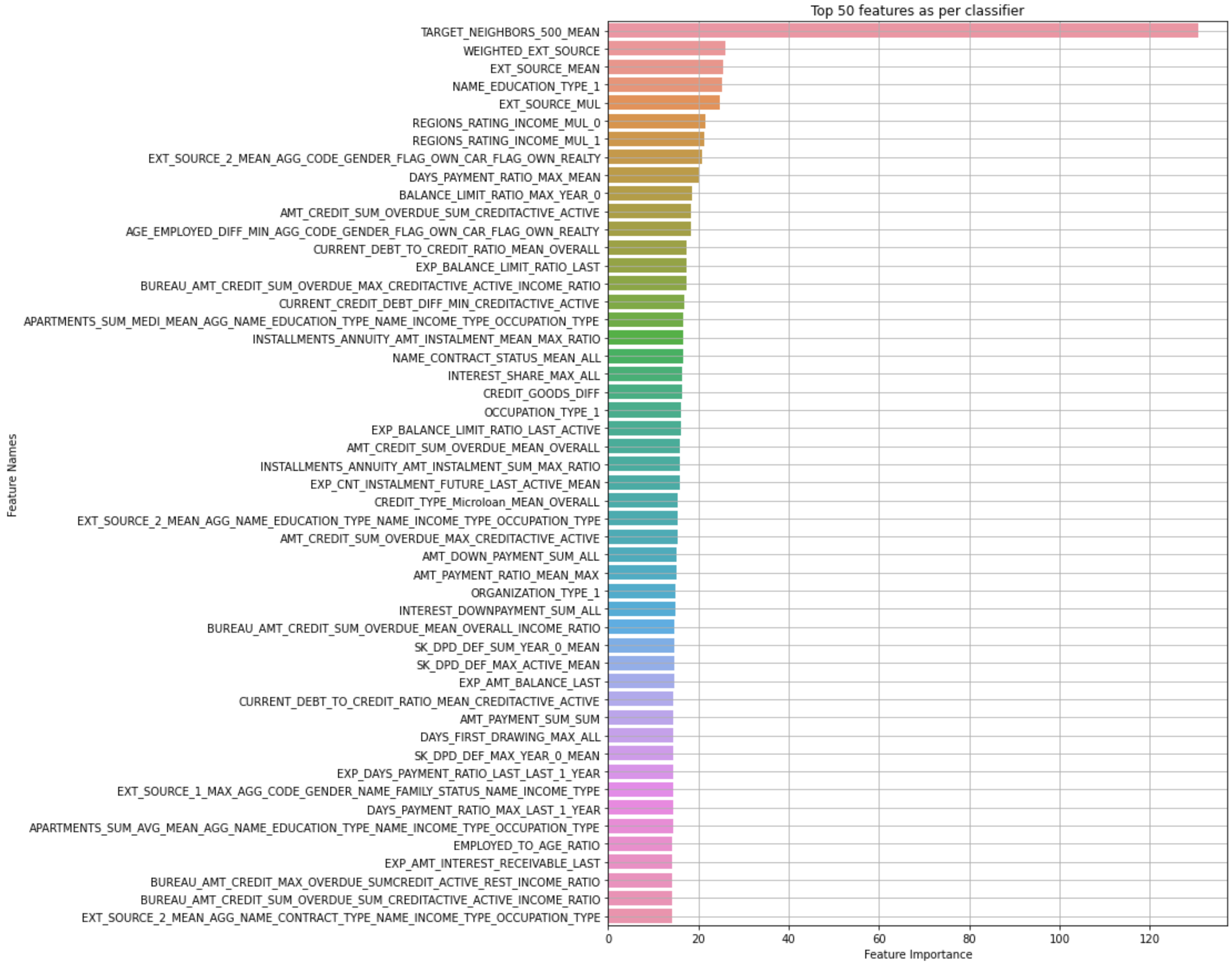
```python
#displaying the results and metrics
xgb_boosting.results()
#displaying top 50 important features
xgb_boosting.feat_importances_show(50)
```

```
==========================================================================================
============
Train Results:

The best selected Threshold as per the J-Statistic, which is J = TPR - FPR, is = 0.04854
641606410345

        ROC-AUC Score = 0.9009970237392284
```

```
        Precision Score = 0.26861325255781904
        Recall Score = 0.832306143001007
CV Results:
        ROC-AUC Score = 0.8055309287396647
        Precision Score = 0.15021520192048182
        Recall Score = 0.851963746223565
==========================================================================================
============
Confusion, Precision and Recall Matrix on CV data:
```




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




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

In [59]:

```python
features_with_zero_importances = xgb_boosting.feature_importance[xgb_boosting.feature_i

print(f"There are {len(features_with_zero_importances)} features with Zero Gain in XGBC
print(features_with_zero_importances.features.values)
```

```
There are 44 features with Zero Gain in XGBClassifier. They are:

['NAME_CONTRACT_TYPE_1' 'REGION_RATING_CLIENT_W_CITY_1' 'FLAG_MOBIL'
 'NAME_EDUCATION_TYPE_0' 'NAME_CONTRACT_TYPE_0'
 'CURRENT_AMT_OVERDUE_DURATION_RATIO_MEAN_OVERALL' 'NAME_FAMILY_STATUS_0'
 'FLAG_EMP_PHONE' 'FLAG_OWN_CAR_1' 'CURRENCY_currency 4_MEAN_OVERALL'
 'DAYS_CREDIT_ENDDATE_MEAN_OVERALL'
 'APARTMENTS_SUM_AVG_MEAN_AGG_CODE_GENDER_FLAG_OWN_CAR_FLAG_OWN_REALTY'
 'REGION_RATING_CLIENT_1'
 'CREDIT_TYPE_Loan for the purchase of equipment_MEAN_OVERALL'
 'CURRENT_DEBT_TO_CREDIT_RATIO_MINCREDIT_ACTIVE_REST'
 'FONDKAPREMONT_MODE_0' 'CREDIT_DURATION_MEAN_OVERALL' 'HOUSETYPE_MODE_1'
 'APARTMENTS_SUM_MEDI_MIN_AGG_CODE_GENDER_FLAG_OWN_CAR_FLAG_OWN_REALTY'
 'FLAG_DOCUMENT_7' 'NAME_HOUSING_TYPE_1' 'FLAG_DOCUMENT_21'
 'NAME_TYPE_SUITE_0' 'CREDIT_TYPE_Real estate loan_MEAN_OVERALL'
 'EXT_SOURCE_3_MIN_AGG_CODE_GENDER_NAME_FAMILY_STATUS_NAME_INCOME_TYPE'
 'APARTMENTS_SUM_MEDI_MAX_AGG_CODE_GENDER_FLAG_OWN_CAR_FLAG_OWN_REALTY'
 'EXT_SOURCE_3_MIN_AGG_FLAG_OWN_CAR_FLAG_OWN_REALTY_NAME_INCOME_TYPE'
 'CREDIT_TYPE_Interbank credit_MEAN_OVERALL'
 'CREDIT_ACTIVE_Bad debt_MEAN_OVERALL'
```

```
'CNT_PROLONGED_DURATION_RATIO_MEAN_OVERALL' 'OCCUPATION_TYPE_0'
'CODE_GENDER_1'
'BUREAU_AMT_CREDIT_SUM_OVERDUE_MAX_CREDITACTIVE_CLOSED_INCOME_RATIO'
'CURRENCY_currency 3_MEAN_OVERALL' 'WALLSMATERIAL_MODE_1'
'CREDIT_TYPE_Loan for purchase of shares (margin lending)_MEAN_OVERALL'
'FLAG_DOCUMENT_15'
'APARTMENTS_SUM_MEDI_MAX_AGG_FLAG_OWN_CAR_FLAG_OWN_REALTY_NAME_INCOME_TYPE'
'FLAG_DOCUMENT_17' 'NAME_INCOME_TYPE_1' 'NAME_INCOME_TYPE_0'
'REGION_RATING_MIN' 'FLAG_DOCUMENT_19' 'REGION_RATING_CLIENT_W_CITY_0']
```

In [60]:

```
XGB_model_df = pd.DataFrame({'SK_ID_CURR': skid_test, 'TARGET' : xgb_boosting.test_pred
XGB_model_df['SK_ID_CURR'] = XGB_model_df['SK_ID_CURR'].astype(int)
XGB_model_df.to_csv('./submissions/XGBoost_3folds_final.csv',index = False)
```

Model	Private Score	Public Score
XGBoost_3folds_final	0.79733	0.79921

title

In [61]:

```
with open('./data/xgb_imp_feats.pkl','wb') as f:
    pickle.dump(xgb_boosting.feature_importance, f)
with open('./data/xgb_cv_preds.pkl', 'wb') as f:
    pickle.dump(xgb_boosting.cv_preds_proba, f)
with open('./data/xgb_test_preds.pkl', 'wb') as f:
    pickle.dump(xgb_boosting.test_preds_proba_mean, f)
```

Results Discussion

1. The XGBoost model with optimized Hyper-parameters seems to perform better than all the models used so far, by a margin. These results are reflected both for Cross-Validation, as well as Test Scores.
2. The single model of XGBoost gives us a private Score of 0.79733 which is about top 7.6% in private leaderboard.
3. The CV and Test AUC are also very close to each other, which implies similar distribution of training and test data.
4. We notice here that the Recall Value for CV dataset is actually higher than Train Dataset, but the precision values are very low.
5. Looking at the Confusion Matrix, we see that There are very few False Negatives, but the Numbers of False Positives is quite high. This is the trade-off between the recall and precision that we have to take.
6. We have tuned the threshold by using the ROC-AUC curve, which tries to maximize the True Positive Rate (which we can see from the Confusion Matrix as well) and also to minimize the False Positive Rate, which is close to 0.047 here.

Feature Importances

1. The Tree Based methods give sensible feature importances. We see that the highest scoring feature according to Gini Gain is TARGET Mean of 500 neighbors feature. We see lots of features generated from EXT_SOURCE as top scoring features. Among the other features, we

notice that the features generated from Categorical interactions in application_train table have also scored high by the model.

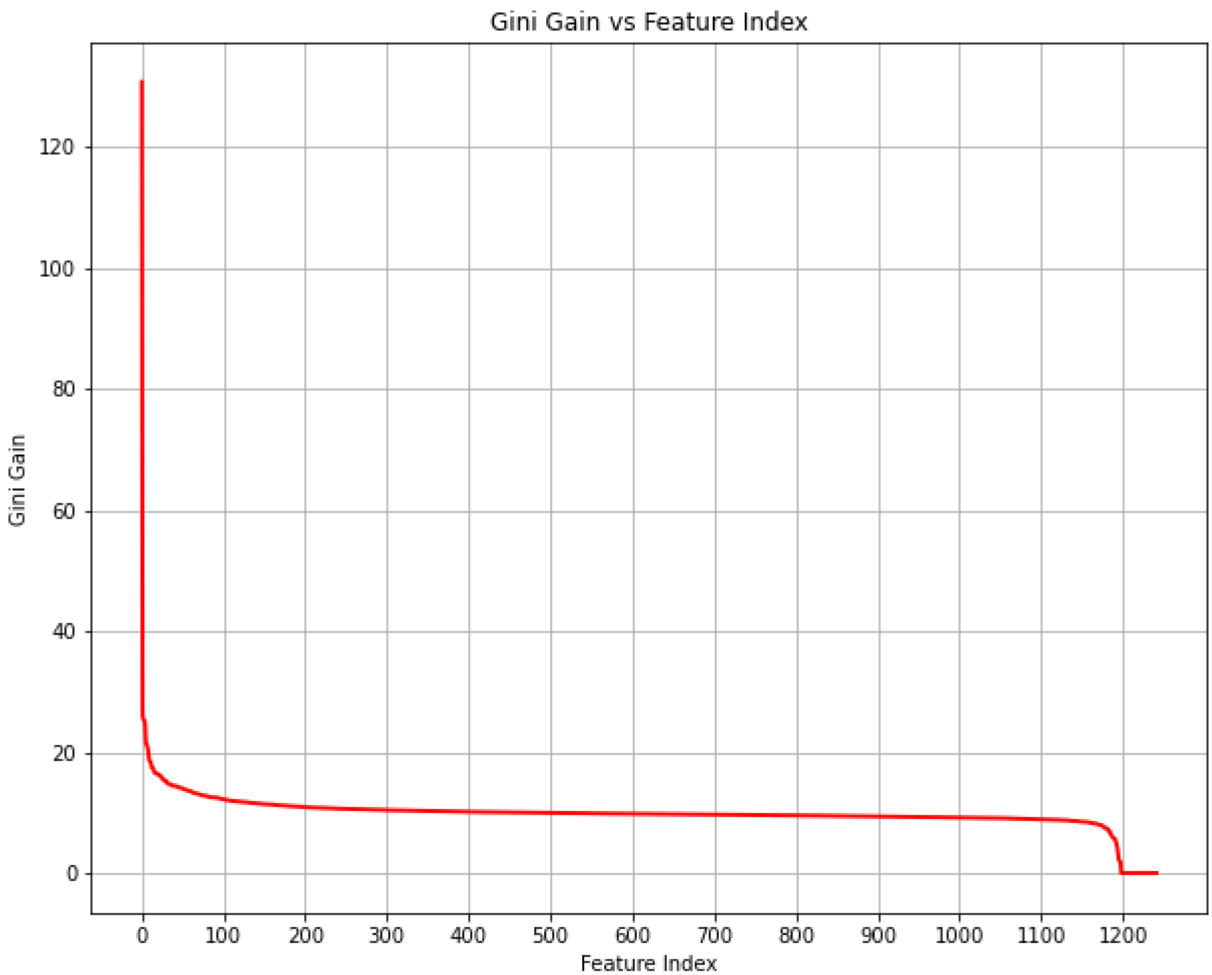
2. Among the 1236 features, we find that 44 features have exactly 0 Gini Gain, and we would be better off removing them if we were to use this classifier.

Analysing the Feature Importances From XGBoost

Let us analyze the feature importances of the features obtained from XGBoost Model.

In [62]:

```
#plotting the importance vs feature index for the features
plt.figure(figsize = (10,8))
plt.plot(list(range(len(xgb_boosting.feature_importance))), xgb_boosting.feature_import
plt.xlabel('Feature Index')
plt.ylabel('Gini Gain')
plt.xticks(list(range(0,1300,100)))
plt.grid()
plt.title("Gini Gain vs Feature Index")
plt.show()
```

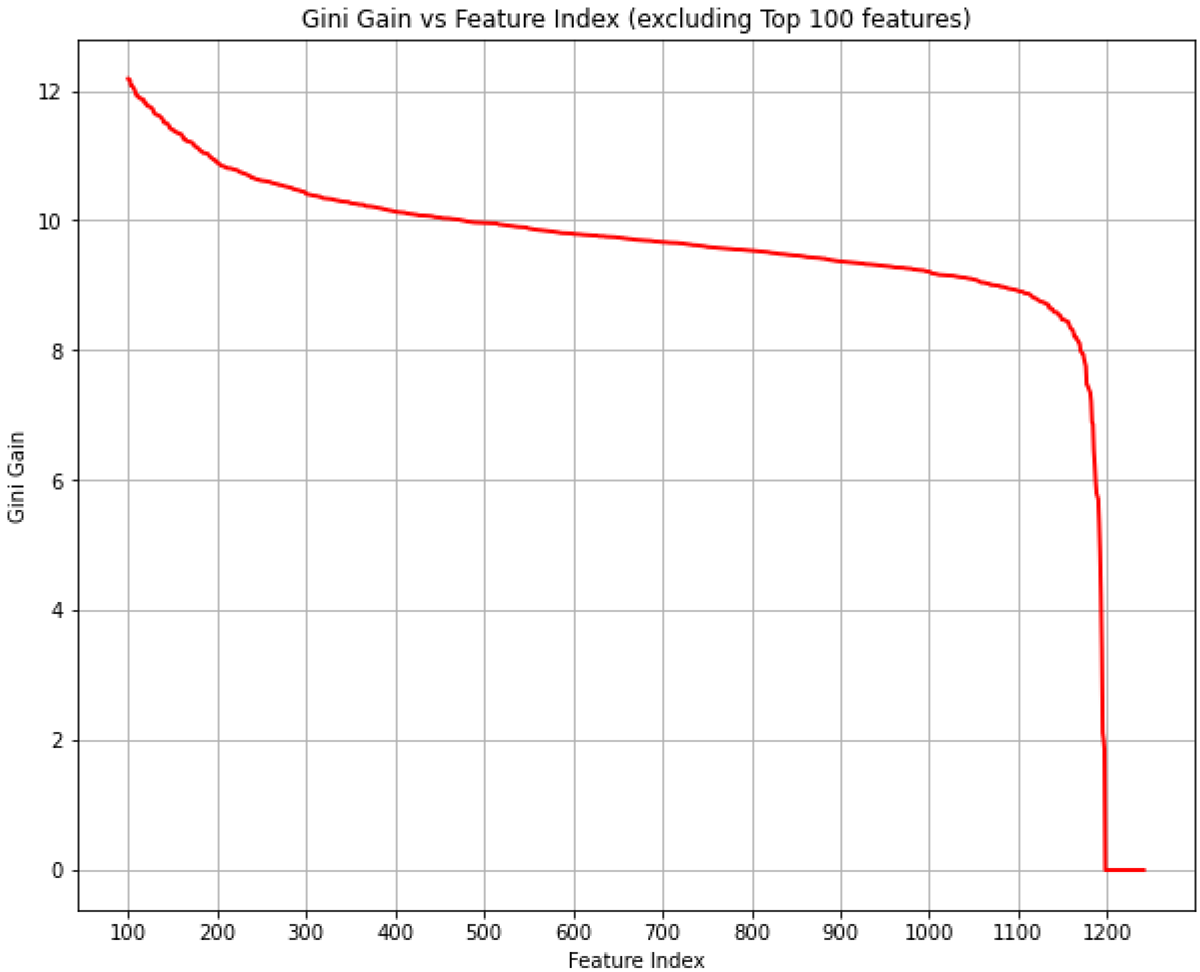


From the above figure we notice that there is an elbow point at about 100 features, after which the gain falls very slowly. Let us try to exclude these first 100 highest scoring features, and try to observe more features.

In [63]:

```
#plotting the importance vs feature index for the features
plt.figure(figsize = (10,8))
```

```
plt.plot(list(range(len(xgb_boosting.feature_importance)-100)), xgb_boosting.feature_im
plt.xlabel('Feature Index')
plt.ylabel('Gini Gain')
plt.xticks(ticks = list(range(0,1200,100)), labels = list(range(100,1300,100)))
plt.grid()
plt.title("Gini Gain vs Feature Index (excluding Top 100 features)")
plt.show()
```



Here again, we see a point of elbow of drop in feature importance/gini gain at feature index between 500-600 features. So we will take the top 600 features and try to model the XGBoostClassifier with same parameters on this reduced set of parameters, and compare the performance.

2.8 XGBoost GPU on Reduced Features

In [64]:
```
#reducing the feature set
#taking the top 600 scoring features
train_data_600 = train_data[xgb_boosting.feature_importance.features.iloc[:600]]
test_data_600 = test_data[xgb_boosting.feature_importance.features.iloc[:600]]

print(f"Training Data Shape with reduced feature set = {train_data_600.shape}")
print(f"Test Data Shape with reduced feature set = {test_data_600.shape}")
```

```
Training Data Shape with reduced feature set = (307507, 600)
Test Data Shape with reduced feature set = (48744, 600)
```

Training the model

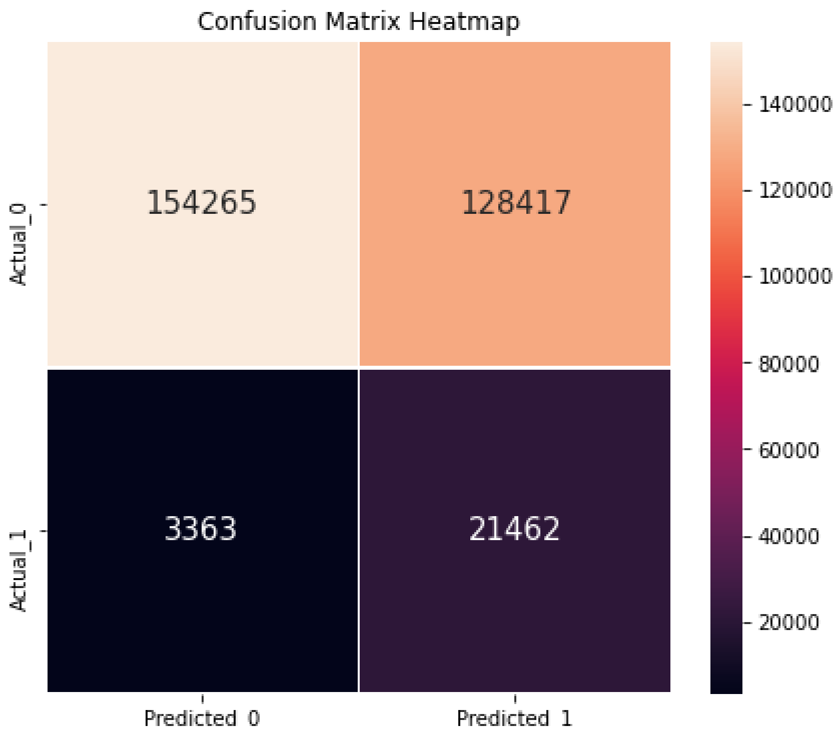
In [65]:
```
#training on the already optimized params above
xgb_boosting_600 = Boosting(train_data_600, target_train, test_data_600, params, random
xgb_boosting_600.train(booster = 'xgboost', pickle_name = '600feats')
```

```
Fitting the xgboost on Training Data with 3 fold cross validation, and using Out-Of-Fold
s Predictions for Cross-Validation

        Fold Number 1

[0]     validation_0-auc:0.75535        validation_1-auc:0.75941
[400]   validation_0-auc:0.79894        validation_1-auc:0.78876
[800]   validation_0-auc:0.82369        validation_1-auc:0.80060
[1200]  validation_0-auc:0.83839        validation_1-auc:0.80460
[1600]  validation_0-auc:0.85020        validation_1-auc:0.80639
[2000]  validation_0-auc:0.86070        validation_1-auc:0.80722
[2400]  validation_0-auc:0.86973        validation_1-auc:0.80772
[2800]  validation_0-auc:0.87821        validation_1-auc:0.80809
[3200]  validation_0-auc:0.88589        validation_1-auc:0.80817
[3556]  validation_0-auc:0.89258        validation_1-auc:0.80821

        Fold Number 2

[0]     validation_0-auc:0.75784        validation_1-auc:0.75117
[400]   validation_0-auc:0.80011        validation_1-auc:0.78230
[800]   validation_0-auc:0.82405        validation_1-auc:0.79655
[1200]  validation_0-auc:0.83873        validation_1-auc:0.80117
[1600]  validation_0-auc:0.85061        validation_1-auc:0.80332
[2000]  validation_0-auc:0.86094        validation_1-auc:0.80456
[2400]  validation_0-auc:0.87031        validation_1-auc:0.80502
[2800]  validation_0-auc:0.87901        validation_1-auc:0.80529
[3200]  validation_0-auc:0.88693        validation_1-auc:0.80530
[3233]  validation_0-auc:0.88750        validation_1-auc:0.80530

        Fold Number 3

[0]     validation_0-auc:0.75705        validation_1-auc:0.74857
[400]   validation_0-auc:0.80167        validation_1-auc:0.78179
[800]   validation_0-auc:0.82551        validation_1-auc:0.79499
[1200]  validation_0-auc:0.84023        validation_1-auc:0.79941
[1600]  validation_0-auc:0.85198        validation_1-auc:0.80137
[2000]  validation_0-auc:0.86210        validation_1-auc:0.80249
[2400]  validation_0-auc:0.87107        validation_1-auc:0.80310
[2800]  validation_0-auc:0.87929        validation_1-auc:0.80329
[2868]  validation_0-auc:0.88058        validation_1-auc:0.80330
Done.
Time elapsed = 0:05:00.447999
```

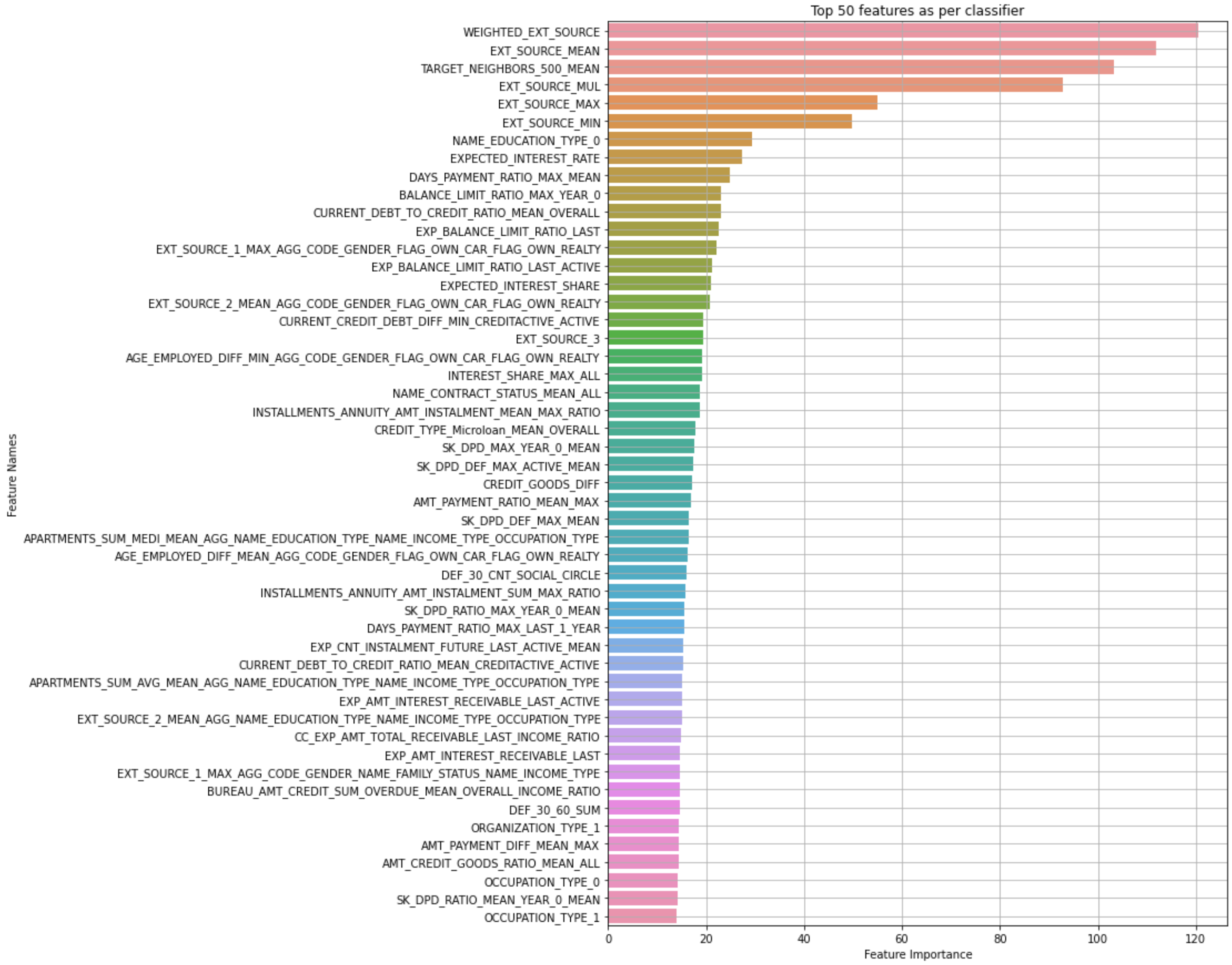
In [147…
```
#displaying the results and metrics
xgb_boosting_600.results()
#displaying top 50 important features
xgb_boosting_600.feat_importances_show(50)
```

```
==========================================================================================
============
Train Results:
```

```
The best selected Threshold as per the J-Statistic, which is J = TPR - FPR, is = 0.04608
752330144246

        ROC-AUC Score = 0.8602796863474593
        Precision Score = 0.23351674756115143
        Recall Score = 0.7829607250755287
CV Results:
        ROC-AUC Score = 0.8012951862548242
        Precision Score = 0.14319551104557676
        Recall Score = 0.8645317220543807
========================================================================================
============
Confusion, Precision and Recall Matrix on CV data:
```




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




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

In [66]:

```
with open('./data/xgb_imp_feats_600feats.pkl','wb') as f:
    pickle.dump(xgb_boosting_600.feature_importance, f)
with open('./data/xgb_cv_preds_600feats.pkl', 'wb') as f:
    pickle.dump(xgb_boosting_600.cv_preds_proba, f)
with open('./data/xgb_test_preds_600feats.pkl', 'wb') as f:
    pickle.dump(xgb_boosting_600.test_preds_proba_mean, f)
with open('./data/Final_XGBOOST_Selected_features', 'wb') as f:
    pickle.dump(xgb_boosting_600.feature_importance.features.values, f)
```

In [67]:

```
XGB600_model_df = pd.DataFrame({'SK_ID_CURR': skid_test, 'TARGET' : xgb_boosting_600.te
XGB600_model_df['SK_ID_CURR'] = XGB600_model_df['SK_ID_CURR'].astype(int)
XGB600_model_df.to_csv('./submissions/XGBoost_600_features.csv',index = False)
```

Model	Private Score	Public Score
XGBoost_600_features	0.79744	0.79768

title

Results Discussion

1. From the results, we observe that the gap between CV and Train ROC-AUC Score has reduced, which means a little bit lesser over-fit compared to the model with all 1236 features.

2. We also notice that the Recall has improved, while the Precision has reduced very slightly.
3. The above point can also be realised by looking at the Confusion Matrix, which reported a higher number of True Positives now.
4. The Private Score has also improved by removing the features. This implies that there might have been some noisy features, which were causing performance degradation.
5. The Feature Importances remain more or less the same.

Thus, we can conclude that by removing the feature sets, we have both improved the performance, and reduced the computaional complexity of our model.

2.9 LightGBM

Bayesian Optimization

In [68]:
```
def lgbm_evaluation(num_leaves, max_depth, min_split_gain, min_child_weight,
                    min_child_samples, subsample, colsample_bytree, reg_alpha, reg_lamb
    '''
    Objective function for Bayesian Optimization of LightGBM's Hyperparamters. Takes th
    returns the Cross-Validation AUC as output.

    Inputs: Hyperparamters to be tuned.
        num_leaves, max_depth, min_split_gain, min_child_weight,
        min_child_samples, subsample, colsample_bytree, reg_alpha, reg_lambda

    Returns:
        CV ROC-AUC Score
    '''

    params = {
        'objective' : 'binary',
        'boosting_type' : 'gbdt',
        'learning_rate' : 0.005,
        'n_estimators' : 10000,
        'n_jobs' : 6,
        'num_leaves' : int(round(num_leaves)),
        'max_depth' : int(round(max_depth)),
        'min_split_gain' : min_split_gain,
        'min_child_weight' : min_child_weight,
        'min_child_samples' : int(round(min_child_samples)),
        'subsample': subsample,
        'subsample_freq' : 1,
        'colsample_bytree' : colsample_bytree,
        'reg_alpha' : reg_alpha,
        'reg_lambda' : reg_lambda,
        'verbosity' : -1,
        'seed' : 266
    }
    stratified_cv = StratifiedKFold(n_splits = 3, shuffle = True, random_state = 33)

    cv_preds = np.zeros(train_data.shape[0])
    for train_indices, cv_indices in stratified_cv.split(train_data, target_train):

        x_tr = train_data.iloc[train_indices]
        y_tr = target_train.iloc[train_indices]
        x_cv = train_data.iloc[cv_indices]
```

```
        y_cv = target_train.iloc[cv_indices]

        lgbm_clf = lgb.LGBMClassifier(**params)
        lgbm_clf.fit(x_tr, y_tr, eval_set= [(x_cv, y_cv)],
                eval_metric='auc', verbose = False, early_stopping_rounds=200)

        cv_preds[cv_indices] = lgbm_clf.predict_proba(x_cv, num_iteration = lgbm_clf.be

    return roc_auc_score(target_train, cv_preds)
```

In [69]:

```
bopt_lgbm = BayesianOptimization(lgbm_evaluation, {'num_leaves' : (35,50),
                                                   'max_depth' : (6,11),
                                                   'min_split_gain' : (0, 0.1),
                                                   'min_child_weight' : (15,50),
                                                   'min_child_samples' : (15,50),
                                                   'subsample' : (0.5,1),
                                                   'colsample_bytree' : (0.5,1),
                                                   'reg_alpha' : (0.01, 0.3),
                                                   'reg_lambda' : (0.01, 0.3)},
                                 random_state = 4976)

bayesian_optimization = bopt_lgbm.maximize(n_iter = 6, init_points = 4) #cost time: 7h
```

```
|   iter    |  target   | colsam... | max_depth | min_ch... | min_ch... | min_sp... | nu
m_le... | reg_alpha | reg_la... | subsample |
-----------------------------------------------------------------------------------------
---------------------------------------------
|  1        |  0.8062   |  0.9839   |  9.624    |  43.27    |  41.12    |  0.08223  |  4
3.73    |  0.1189   |  0.2078   |  0.6677   |
|  2        |  0.8062   |  0.5453   |  10.99    |  29.51    |  32.59    |  0.02383  |  4
5.87    |  0.126    |  0.1983   |  0.8343   |
|  3        |  0.8061   |  0.7313   |  9.478    |  34.64    |  37.44    |  0.0249   |  4
2.17    |  0.2435   |  0.113    |  0.5916   |
|  4        |  0.8057   |  0.5671   |  7.674    |  25.12    |  16.73    |  0.004937 |  3
8.89    |  0.1234   |  0.1009   |  0.9706   |
|  5        |  0.806    |  0.6163   |  10.29    |  42.64    |  40.52    |  0.0798   |  4
2.95    |  0.08014  |  0.07393  |  0.8568   |
|  6        |  0.8062   |  0.6623   |  8.873    |  27.45    |  27.55    |  0.01941  |  4
2.97    |  0.1332   |  0.2013   |  0.7704   |
|  7        |  0.8063   |  0.5624   |  7.805    |  39.31    |  46.8     |  0.002874 |
 38.15    |  0.1497   |  0.1483   |  0.7644   |
|  8        |  0.8063   |  0.7067   |  10.14    |  37.37    |  40.64    |  0.05523  |
 45.15    |  0.1032   |  0.04372  |  0.6152   |
|  9        |  0.8063   |  0.7883   |  6.16     |  35.82    |  43.78    |  0.05549  |  4
4.22    |  0.2576   |  0.2102   |  0.7711   |
|  10       |  0.8062   |  0.6714   |  8.987    |  45.52    |  29.74    |  0.000376 |  4
1.86    |  0.2362   |  0.1222   |  0.5774   |
=========================================================================================
=============================================
```

In [70]:

```
#extracting the best parameters
target_values = []
for result in bopt_lgbm.res:
    target_values.append(result['target'])
    if result['target'] == max(target_values):
        best_params = result['params']
```

```
print("Best Hyperparameters obtained are:\n")
print(best_params)
```

```
Best Hyperparameters obtained are:

{'colsample_bytree': 0.7066505590597474, 'max_depth': 10.13825022211375, 'min_child_samp
les': 37.36535403648989, 'min_child_weight': 40.638097605743226, 'min_split_gain': 0.055
22683601173706, 'num_leaves': 45.15147916081096, 'reg_alpha': 0.10315788390481063, 'reg_
lambda': 0.043722912800632646, 'subsample': 0.6151973197579237}
```

In [72]:

```
bopt_lgbm.max
```

Out[72]:

```
{'target': 0.806302850969016,
 'params': {'colsample_bytree': 0.7066505590597474,
  'max_depth': 10.13825022211375,
  'min_child_samples': 37.36535403648989,
  'min_child_weight': 40.638097605743226,
  'min_split_gain': 0.05522683601173706,
  'num_leaves': 45.15147916081096,
  'reg_alpha': 0.10315788390481063,
  'reg_lambda': 0.043722912800632646,
  'subsample': 0.6151973197579237}}
```

In [73]:

```
params_custom = {
        'objective' : 'binary',
        'boosting_type' : 'gbdt',
        'learning_rate' : 0.005,
        'n_estimators' : 10000,
        'n_jobs' : 6,
        'subsample_freq' : 1,
        'verbosity' : -1,
        'seed' : 266
    }
params_int = {'num_leaves': 45, 'max_depth': 10, 'min_child_samples': 37} #overwrite th
params = {**params_custom, **best_params, **params_int}
```

In [74]:

```
# params = {
#          'objective' : 'binary',
#          'boosting_type' : 'gbdt',
#          'learning_rate' : 0.005,
#          'n_estimators' : 10000,
#          'n_jobs' : -1,
#          'num_leaves' : 46,
#          'max_depth' : 11,
#          'min_split_gain' : 0.023832030441816324,
#          'min_child_weight' : 32.592575418812544,
#          'min_child_samples' : 30,
#          'subsample': 0.8343200716558421,
#          'subsample_freq' : 1,
#          'colsample_bytree' : 0.5453481803843285,
#          'reg_alpha' : 0.12598946525394644,
#          'reg_lambda' : 0.19826240162255976,
#          'verbosity' : -1,
#          'seed' : 266
#     }
lgbm_boosting = Boosting(train_data, target_train, test_data, params, random_state = 98
lgbm_boosting.train(booster = 'lightgbm') #cost 41min 49sec
```

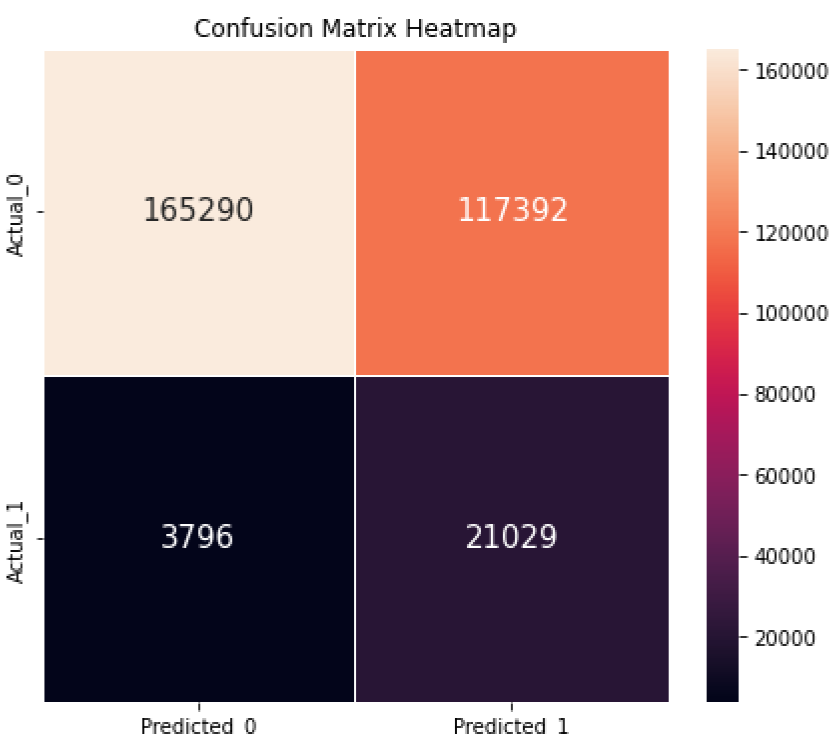
```
Fitting the lightgbm on Training Data with 3 fold cross validation, and using Out-Of-Fol
ds Predictions for Cross-Validation

        Fold Number 1

[400]   training's auc: 0.805496        training's binary_logloss: 0.233475     valid_
1's auc: 0.790214       valid_1's binary_logloss: 0.239054
[800]   training's auc: 0.826886        training's binary_logloss: 0.222778     valid_
1's auc: 0.799244       valid_1's binary_logloss: 0.233684
[1200]  training's auc: 0.842444        training's binary_logloss: 0.215789     valid_
1's auc: 0.803572       valid_1's binary_logloss: 0.231725
[1600]  training's auc: 0.855352        training's binary_logloss: 0.210083     valid_
1's auc: 0.80573        valid_1's binary_logloss: 0.230821
[2000]  training's auc: 0.866796        training's binary_logloss: 0.204956     valid_
1's auc: 0.806883       valid_1's binary_logloss: 0.230328
[2400]  training's auc: 0.876892        training's binary_logloss: 0.200283     valid_
1's auc: 0.80745        valid_1's binary_logloss: 0.230061
[2800]  training's auc: 0.885893        training's binary_logloss: 0.195945     valid_
1's auc: 0.807992       valid_1's binary_logloss: 0.229838
[3200]  training's auc: 0.894265        training's binary_logloss: 0.191807     valid_
1's auc: 0.8082 valid_1's binary_logloss: 0.22975
[3600]  training's auc: 0.902144        training's binary_logloss: 0.187763     valid_
1's auc: 0.808314       valid_1's binary_logloss: 0.229697

        Fold Number 2

[400]   training's auc: 0.806238        training's binary_logloss: 0.23291      valid_
1's auc: 0.784412       valid_1's binary_logloss: 0.240303
[800]   training's auc: 0.826881        training's binary_logloss: 0.222441     valid_
1's auc: 0.794825       valid_1's binary_logloss: 0.235137
[1200]  training's auc: 0.842231        training's binary_logloss: 0.215465     valid_
1's auc: 0.799925       valid_1's binary_logloss: 0.23307
[1600]  training's auc: 0.854893        training's binary_logloss: 0.20986      valid_
1's auc: 0.802427       valid_1's binary_logloss: 0.232093
[2000]  training's auc: 0.866113        training's binary_logloss: 0.204827     valid_
1's auc: 0.803832       valid_1's binary_logloss: 0.231565
[2400]  training's auc: 0.876207        training's binary_logloss: 0.200188     valid_
1's auc: 0.804711       valid_1's binary_logloss: 0.231251
[2800]  training's auc: 0.885467        training's binary_logloss: 0.195785     valid_
1's auc: 0.805231       valid_1's binary_logloss: 0.231053
[3200]  training's auc: 0.893898        training's binary_logloss: 0.19167      valid_
1's auc: 0.805573       valid_1's binary_logloss: 0.230925
[3600]  training's auc: 0.901632        training's binary_logloss: 0.187691     valid_
1's auc: 0.805796       valid_1's binary_logloss: 0.230849

        Fold Number 3

[400]   training's auc: 0.807367        training's binary_logloss: 0.232923     valid_
1's auc: 0.783388       valid_1's binary_logloss: 0.240151
[800]   training's auc: 0.828194        training's binary_logloss: 0.222327     valid_
1's auc: 0.793348       valid_1's binary_logloss: 0.235055
[1200]  training's auc: 0.843407        training's binary_logloss: 0.215378     valid_
1's auc: 0.797774       valid_1's binary_logloss: 0.233265
[1600]  training's auc: 0.856031        training's binary_logloss: 0.209714     valid_
1's auc: 0.800031       valid_1's binary_logloss: 0.232419
[2000]  training's auc: 0.867217        training's binary_logloss: 0.204671     valid_
1's auc: 0.801416       valid_1's binary_logloss: 0.231908
[2400]  training's auc: 0.877302        training's binary_logloss: 0.19998      valid_
1's auc: 0.802184       valid_1's binary_logloss: 0.231625
```

```
[2800]  training's auc: 0.886467        training's binary_logloss: 0.19553      valid_
1's auc: 0.802713       valid_1's binary_logloss: 0.231438
[3200]  training's auc: 0.89483 training's binary_logloss: 0.191381     valid_1's auc:
0.802908        valid_1's binary_logloss: 0.231369
[3600]  training's auc: 0.902667        training's binary_logloss: 0.187366     valid_
1's auc: 0.803053       valid_1's binary_logloss: 0.23133
Done.
Time elapsed = 0:41:49.661320
```

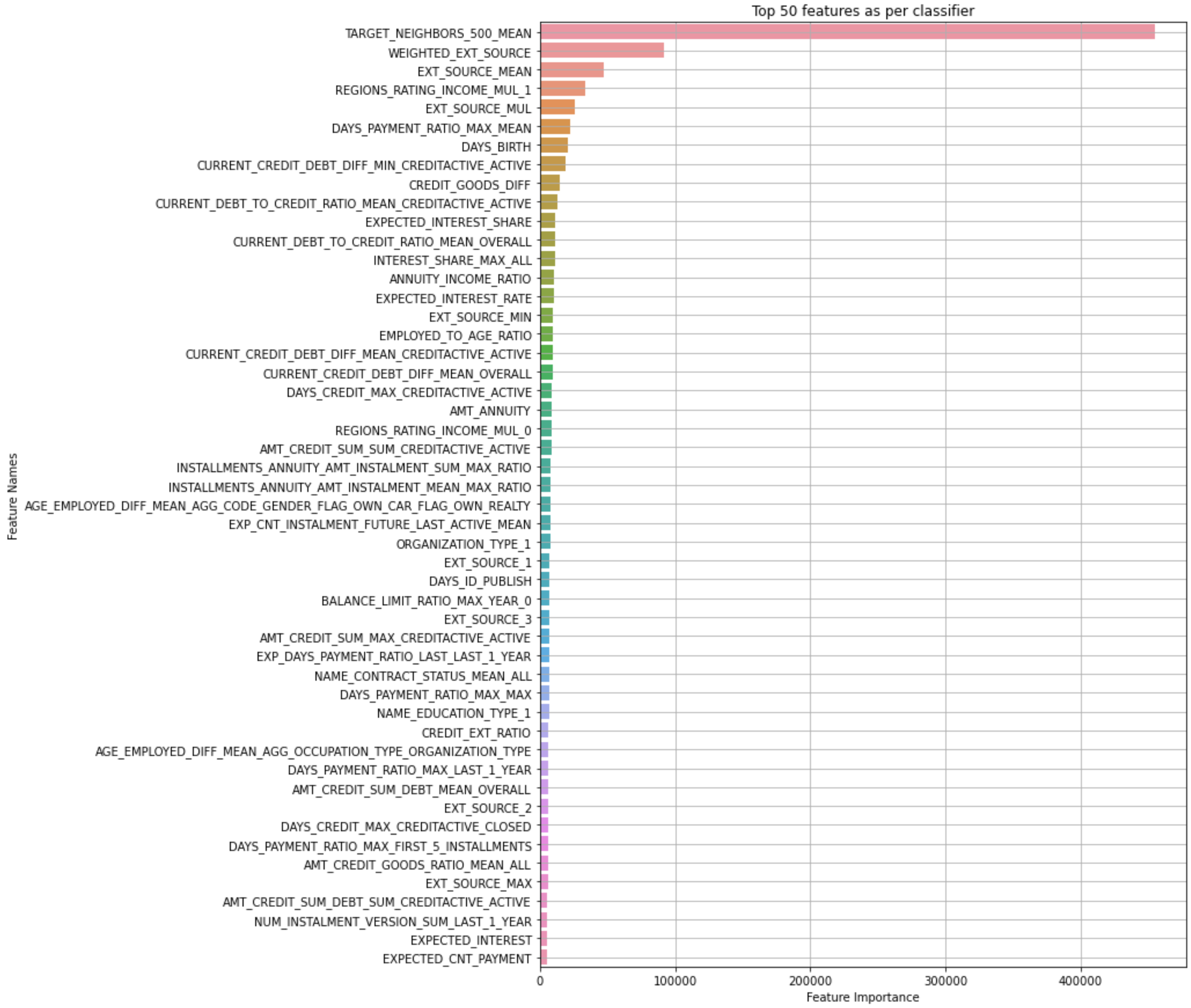
In [75]:

```
#displaying the results and metrics
lgbm_boosting.results()
#displaying top 50 important features
lgbm_boosting.feat_importances_show(50)
```

```
==========================================================================================
============
Train Results:

The best selected Threshold as per the J-Statistic, which is J = TPR - FPR, is = 0.04917
884705758045

        ROC-AUC Score = 0.8993801694320392
        Precision Score = 0.2714162520729685
        Recall Score = 0.8240886203423968
CV Results:
        ROC-AUC Score = 0.8057214637070115
        Precision Score = 0.15192059008387457
        Recall Score = 0.8470896273917422
==========================================================================================
============
Confusion, Precision and Recall Matrix on CV data:
```




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




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

In [76]:
```python
features_with_zero_importances  = lgbm_boosting.feature_importance[lgbm_boosting.featur

print(f"There are {len(features_with_zero_importances)} features with Zero Gain in LGBM
print(features_with_zero_importances.features.values)
```

```
There are 47 features with Zero Gain in LGBMClassifier. They are:

['RATE_INTEREST_PRIVILEGED_MEAN_LAST_5'
 'CREDIT_TYPE_Loan for business development_MEAN_OVERALL'
 'CREDIT_TYPE_Mobile operator loan_MEAN_OVERALL' 'NAME_INCOME_TYPE_1'
 'CREDIT_TYPE_Loan for the purchase of equipment_MEAN_OVERALL'
 'CREDIT_TYPE_Loan for purchase of shares (margin lending)_MEAN_OVERALL'
 'CREDIT_TYPE_Real estate loan_MEAN_OVERALL'
 'EXT_SOURCE_3_MIN_AGG_CODE_GENDER_NAME_FAMILY_STATUS_NAME_INCOME_TYPE'
 'CREDIT_TYPE_Interbank credit_MEAN_OVERALL'
 'CREDIT_TYPE_Another type of loan_MEAN_OVERALL'
 'CREDIT_DURATION_MEAN_OVERALL' 'CURRENCY_currency 1_MEAN_OVERALL'
 'EXT_SOURCE_3_MIN_AGG_FLAG_OWN_CAR_FLAG_OWN_REALTY_NAME_INCOME_TYPE'
 'CREDIT_ACTIVE_Bad debt_MEAN_OVERALL' 'CODE_GENDER_0'
 'CNT_PROLONGED_DURATION_RATIO_MEAN_OVERALL'
 'AMT_ANNUITY_CREDIT_RATIO_MINCREDIT_ACTIVE_REST'
 'AMT_ANNUITY_CREDIT_RATIO_MEANCREDIT_ACTIVE_REST'
 'CREDIT_TYPE_Unknown type of loan_MEAN_OVERALL' 'FLAG_DOCUMENT_17'
```

```
'CURRENCY_currency 2_MEAN_OVERALL' 'FLAG_DOCUMENT_15' 'FLAG_DOCUMENT_19'
'EXP_AMT_DRAWING_SUM_LAST_REST'
'BUREAU_AMT_CREDIT_MAX_OVERDUE_MAXCREDIT_ACTIVE_REST_INCOME_RATIO'
'FLAG_DOCUMENT_21'
'BUREAU_AMT_CREDIT_SUM_OVERDUE_MAX_CREDITACTIVE_CLOSED_INCOME_RATIO'
'FLAG_DOCUMENT_7' 'FLAG_DOCUMENT_9' 'FLAG_EMP_PHONE' 'FLAG_MOBIL'
'SK_DPD_SUM_YEAR_0' 'FLAG_OWN_CAR_1' 'CNT_DRAWINGS_OTHER_CURRENT_MAX'
'AMT_ANNUITY_CREDIT_RATIO_MAXCREDIT_ACTIVE_REST'
'SK_DPD_RATIO_MAX_COMPLETED_MEAN' 'FLAG_DOCUMENT_13'
'SK_DPD_DEF_SUM_YEAR_1' 'DAYS_CREDIT_ENDDATE_MEAN_OVERALL'
'FLAG_CONT_MOBILE' 'CURRENT_DEBT_TO_CREDIT_RATIO_MIN_CREDITACTIVE_CLOSED'
'CURRENT_DEBT_TO_CREDIT_RATIO_MINCREDIT_ACTIVE_REST'
'NAME_CONTRACT_TYPE_1' 'CURRENT_AMT_OVERDUE_DURATION_RATIO_MEAN_OVERALL'
'CURRENCY_currency 4_MEAN_OVERALL' 'CURRENCY_currency 3_MEAN_OVERALL'
'NAME_CONTRACT_TYPE_0']
```

In [77]:

```
lgbm_boosting_model_df = pd.DataFrame({'SK_ID_CURR': skid_test, 'TARGET' : lgbm_boostin
lgbm_boosting_model_df['SK_ID_CURR'] = lgbm_boosting_model_df['SK_ID_CURR'].astype(int)
lgbm_boosting_model_df.to_csv('./submissions/LGBM_3folds.csv',index = False)
```

Model	Private Score	Public Score
LGBM_3folds	0.79358	0.79470

title

In [78]:

```
with open('./data/lgbm_imp_feats.pkl','wb') as f:
    pickle.dump(lgbm_boosting.feature_importance, f)
with open('./data/lgbm_cv_preds.pkl', 'wb') as f:
    pickle.dump(lgbm_boosting.cv_preds_proba, f)
with open('./data/lgbm_test_preds.pkl', 'wb') as f:
    pickle.dump(lgbm_boosting.test_preds_proba_mean, f)
```

Results Discussion

- Firstly we see that the LightGBM model performs inferiorly as compared to XGBoost model, as observed from the results of Private Score on Test Data. This is contrary to the fact that LightGBM showed higher AUC value for CV dataset as compared to XGBoost.
- Again the Recall values are quite high, which is a good thing, but the precision values are again not good. The high recall value suggests that most of the actual defaulters are predicted by the model, but the low precision also indicates that model predicts lots of people as defaulters, while they actually aren't.
- We see from the Confusion Matrix that the number of True Positives are lesser in the Case of LightGBM compared to XGBoost. However, the total False Positives are also slightly lesser for LightGBM.
- We also observe that boosting methods show the highest number of True Positives among all the models. This is aided by the high Recall value for these models as well.

Feature Importances

- We see that the feature TARGET Mean of 500 neighbors scores quite high as compared to all other features. The difference between gain is very significant. This implies that this feature is indeed a very important feature.

- Similar to all other ensembles, here too, the EXT_SOURCE features score high again.
- One of the interesting high scoring feature for LightGBM is the REGIONS_RATING_INCOME_MUL_1 which is the multiplication of Region Rating of Client with his Income. It has been treated as a categorical feature and has been response encoded, and gives better result than as a numeric feature.
- We also observe that there are around 41 features which had 0 gain in LGBMClassifier for splitting. This means they are not useful for our classifier.

2.10 Stacking Classifiers

In this section, we will try to stack the best classifiers that we have used above, which include Logistic Regression, Linear SVM, Random Forest, XGBoost on reduced features and LightGBM. We will be training a LightGBM Classification Model on the Out-of-fold predictions for all the above classifiers. We will again use the BayesianOptimization Technique for tuning the hyperparameters.

In [86]:

```
def load_stacking_data(file_directory = '', verbose = True):
    '''
    Objective function to prepare the Stacking Training and Test Data. The training dat
    Out-Of-Fold predictions of each base learners, and the test data are the predicted
    on the Test data.

    Inputs:
        file_directory: str, default = ''
            The path of directory where the predictions are located. Insert '/' in the
        verbose: bool, default = True
            Whether to keep verbosity or not

    Returns:
        Training stacking data, Test stacking data, Trainig target values
    '''

    if verbose:
        print("Loading the CV (out-of-folds) and Test Predictions from Base Models...")
        start = datetime.now()

    global sgd_lr_l2_cv, sgd_svm_cv, rf_cv, xgb_cv, lgbm_cv
    global sgd_lr_l2_test, sgd_svm_test, rf_test, xgb_test, lgbm_test

    with open(file_directory + 'SGD_LR_L2_Penalty_CV_Preds.pkl', 'rb') as f:
        sgd_lr_l2_cv = pickle.load(f)
    with open(file_directory + 'SGD_Linear_SVM_CV_Preds.pkl', 'rb') as f:
        sgd_svm_cv = pickle.load(f)
    with open(file_directory + 'Random_Forest_CV_Preds.pkl', 'rb') as f:
        rf_cv = pickle.load(f)
    with open(file_directory + 'xgb_cv_preds_600feats.pkl', 'rb') as f:
        xgb_cv = pickle.load(f)
    with open(file_directory + 'lgbm_cv_preds.pkl', 'rb') as f:
        lgbm_cv = pickle.load(f)

    #Loading the Predicted Probabilities of Test Set
    sgd_lr_l2_test = pd.read_csv('./submissions/' + 'SGD_LR_L2_penalty.csv')['TARGET']
    sgd_svm_test = pd.read_csv('./submissions/' + 'SGD_Linear_SVM.csv')['TARGET']
    rf_test = pd.read_csv('./submissions/' + 'Random_Forest_cv0.csv')['TARGET']
    xgb_test = pd.read_csv('./submissions/' + 'XGBoost_600_features.csv')['TARGET']
    lgbm_test = pd.read_csv('./submissions/' + 'LGBM_3folds.csv')['TARGET']
```

```
    if verbose:
        print("Making the Training and Test Dataset for Stacking...")
    #defining the training and test datasets
    stack_train = pd.DataFrame(np.stack([sgd_lr_l2_cv, sgd_svm_cv, rf_cv, xgb_cv, lgbm_
                               columns = ['SGD_LR_L2', 'SGD_SVM', 'RF', 'XGB', 'LGBM'])
    stack_target = target_train.copy()
    stack_test = pd.concat([sgd_lr_l2_test, sgd_svm_test, rf_test, xgb_test, lgbm_test]
    stack_test.columns = ['SGD_LR_L2', 'SGD_SVM', 'RF', 'XGB', 'LGBM']

    if verbose:
        print("Done.")
        print(f"Time elapsed = {datetime.now() - start}")
        print(f"Shape of stacking training data = {stack_train.shape}")
        print(f"Shape of stacking test data = {stack_test.shape}")
        print(f"Shape of stacking training class labels = {stack_target.shape}")

    return stack_train, stack_test, stack_target
```

In [87]:

```
stack_train, stack_test, stack_target = load_stacking_data(file_directory='./data/')
```

```
Loading the CV (out-of-folds) and Test Predictions from Base Models...
Making the Training and Test Dataset for Stacking...
Done.
Time elapsed = 0:00:00.157002
Shape of stacking training data = (307507, 5)
Shape of stacking test data = (48744, 5)
Shape of stacking training class labels = (307507,)
```

In [88]:

```
stack_train.head()
```

Out[88]:

	SGD_LR_L2	SGD_SVM	RF	XGB	LGBM
0	0.894036	0.362628	0.596951	0.191602	0.232832
1	0.084533	0.009465	0.134516	0.011669	0.013201
2	0.314510	0.039542	0.200077	0.015869	0.017512
3	0.273914	0.033335	0.166268	0.024180	0.022110
4	0.473407	0.067371	0.290049	0.065830	0.055259

Bayesian Optimization

In [20]:

```
def lgbm_stacker_optimization(num_leaves, max_depth, min_split_gain, min_child_weight,
                              min_child_samples, subsample, colsample_bytree, reg_alpha,
    '''
    Function for Bayesian Optimization of LightGBM's Hyperparamters. Takes the hyperpar
    returns the Cross-Validation AUC as output.

    Inputs: Hyperparamters to be tuned.
        num_leaves, max_depth, min_split_gain, min_child_weight,
        min_child_samples, subsample, colsample_bytree, reg_alpha, reg_lambda

    Returns:
        CV ROC-AUC Score
```

```
    '''

    params = {
        'objective' : 'binary',
        'boosting_type' : 'gbdt',
        'learning_rate' : 0.005,
        'n_estimators' : 10000,
        'n_jobs' : -1,
        'num_leaves' : int(round(num_leaves)),
        'max_depth' : int(round(max_depth)),
        'min_split_gain' : min_split_gain,
        'min_child_weight' : min_child_weight,
        'min_child_samples' : int(round(min_child_samples)),
        'subsample': subsample,
        'subsample_freq' : 1,
        'colsample_bytree' : colsample_bytree,
        'reg_alpha' : reg_alpha,
        'reg_lambda' : reg_lambda,
        'verbosity' : -1,
        'seed' : 8956
    }
    stratified_cv = StratifiedKFold(n_splits = 4, shuffle = True, random_state = 96)

    cv_preds = np.zeros(stack_train.shape[0])
    for train_indices, cv_indices in stratified_cv.split(stack_train, stack_target):

        x_tr = stack_train.iloc[train_indices]
        y_tr = stack_target.iloc[train_indices]
        x_cv = stack_train.iloc[cv_indices]
        y_cv = stack_target.iloc[cv_indices]

        lgbm_clf = lgb.LGBMClassifier(**params)
        lgbm_clf.fit(x_tr, y_tr, eval_set= [(x_cv, y_cv)],
                eval_metric='auc', verbose = False, early_stopping_rounds=200)

        cv_preds[cv_indices] = lgbm_clf.predict_proba(x_cv, num_iteration = lgbm_clf.be

    return roc_auc_score(stack_target, cv_preds)
```

In [21]:

```
bopt_stacking = BayesianOptimization(lgbm_stacker_optimization, {'num_leaves' : (2,50),
                                                    'max_depth' : (2,6),
                                                    'min_split_gain' : (0, 0.5),
                                                    'min_child_weight' : (1,80),
                                                    'min_child_samples' : (5,80),
                                                    'subsample' : (0.001,1),
                                                    'colsample_bytree' : (0.2,1),
                                                    'reg_alpha' : (0.00001, 0.3),
                                                    'reg_lambda' : (0.00001, 0.3)},
                                 random_state = 6569)

bayesian_optimization = bopt_stacking.maximize(n_iter = 10, init_points = 10)
```

```
|   iter    |  target   | colsam... | max_depth | min_ch... | min_ch... | min_sp... | nu
m_le... | reg_alpha | reg_la... | subsample |
-------------------------------------------------------------------------------------------
---------------------------------------------
|  1        |  0.8062   |  0.7955   |  5.417    |  44.35    |  38.21    |  0.4826   |  1
9.51    |  0.06595  |  0.05161  |  0.5286   |
```

```
| 2         | 0.8063   | 0.7451   | 4.236     | 47.4      | 58.14     | 0.1048    |
 47.96    | 0.2365    | 0.249    | 0.7261    |
| 3         | 0.8065   | 0.6329   | 4.483     | 32.69     | 24.78     | 0.3669    |
 2.316    | 0.09315   | 0.09452  | 0.4668    |
| 4         | 0.8045   | 0.4877   | 4.83      | 70.16     | 64.09     | 0.2487    | 2
2.81     | 0.2672    | 0.01726  | 0.4483    |
| 5         | 0.7801   | 0.3603   | 5.12      | 68.84     | 64.73     | 0.1149    |
 9.906    | 0.1364    | 0.05073  | 0.1665    |
| 6         | 0.7778   | 0.273    | 2.604     | 26.55     | 36.91     | 0.2601    | 1
1.85     | 0.2722    | 0.1413   | 0.4826    |
| 7         | 0.8064   | 0.5581   | 3.506     | 67.74     | 38.22     | 0.06159   | 4
2.22     | 0.2114    | 0.2632   | 0.967     |
| 8         | 0.8016   | 0.2013   | 4.598     | 56.12     | 61.5      | 0.07268   | 4
8.74     | 0.01435   | 0.1516   | 0.9842    |
| 9         | 0.8059   | 0.3512   | 4.492     | 50.98     | 46.29     | 0.4965    |
 2.796    | 0.03877   | 0.01914  | 0.4598    |
| 10        | 0.8052   | 0.7035   | 3.293     | 62.59     | 77.48     | 0.2071    | 2
0.24     | 0.1691    | 0.06729  | 0.08457   |
| 11        | 0.7968   | 0.9518   | 5.961     | 49.74     | 60.77     | 0.438     | 4
6.64     | 0.2935    | 0.03713  | 0.4654    |
| 12        | 0.8067   | 0.777    | 3.466     | 42.65     | 9.224     | 0.07298   |
 29.92    | 0.01558   | 0.2188   | 0.2997    |
| 13        | 0.8047   | 0.3331   | 4.331     | 10.51     | 8.889     | 0.03475   | 3
1.18     | 0.03586   | 0.2845   | 0.5612    |
| 14        | 0.8048   | 0.4541   | 5.093     | 41.07     | 65.63     | 0.4961    | 2
1.89     | 0.003781  | 0.2952   | 0.7052    |
| 15        | 0.8062   | 0.9645   | 3.217     | 75.62     | 50.33     | 0.003747  | 2
8.0      | 0.2101    | 0.06944  | 0.183     |
| 16        | 0.8059   | 0.467    | 3.431     | 32.27     | 25.22     | 0.09926   |
 2.022    | 0.08982   | 0.1381   | 0.2243    |
| 17        | 0.8019   | 0.5934   | 4.195     | 43.37     | 51.37     | 0.3167    | 4
9.52     | 0.1234    | 0.1794   | 0.01152   |
| 18        | 0.7944   | 0.5235   | 4.574     | 72.82     | 43.34     | 0.2154    | 3
4.88     | 0.01741   | 0.169    | 0.3138    |
| 19        | 0.8023   | 0.2      | 2.0       | 41.84     | 59.69     | 0.0       | 5
0.0      | 0.1104    | 0.3      | 1.0       |
| 20        | 0.7911   | 0.6971   | 3.769     | 76.27     | 58.48     | 0.04643   | 2
7.71     | 0.1192    | 0.04012  | 0.009102  |
=========================================================================================
=============================================
```

In [22]:

```python
#extracting the best parameters
target_values = []
for result in bopt_stacking.res:
    target_values.append(result['target'])
    if result['target'] == max(target_values):
        best_params = result['params']

print("Best Hyperparameters obtained are:\n")
print(best_params)
```

```
Best Hyperparameters obtained are:

{'colsample_bytree': 0.777005917777368, 'max_depth': 3.4661509471225993, 'min_child_samp
les': 42.65047945139628, 'min_child_weight': 9.223836586454116, 'min_split_gain': 0.0729
7767759639462, 'num_leaves': 29.918608004586297, 'reg_alpha': 0.015583802542867243, 'reg
_lambda': 0.21883161475419755, 'subsample': 0.29967715403460243}
```

Training on Optimized Hyper-parameters

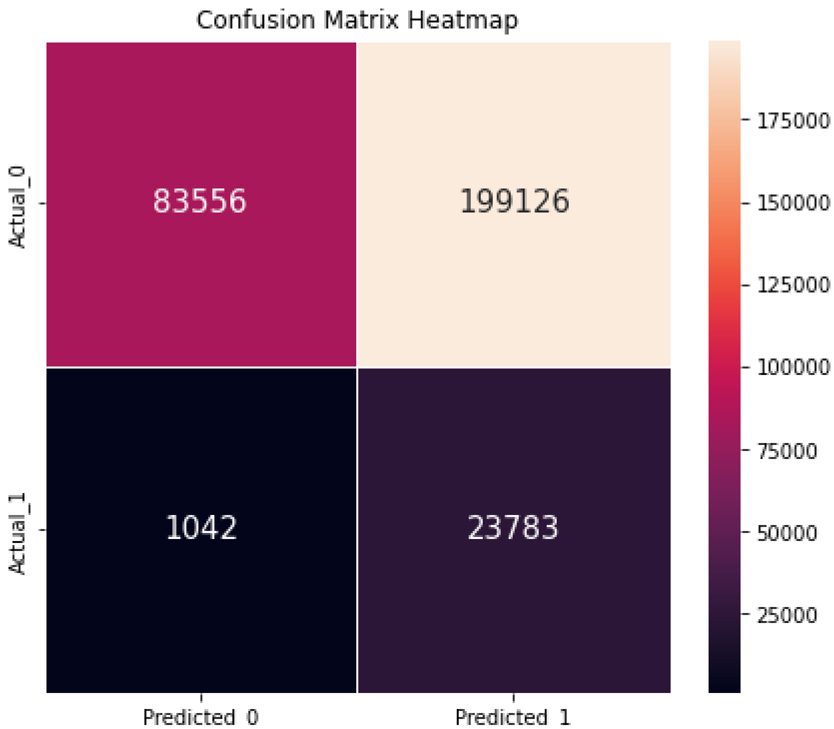
In [89]:
```
params = {
        'objective' : 'binary',
        'boosting_type' : 'gbdt',
        'learning_rate' : 0.005,
        'n_estimators' : 10000,
        'n_jobs' : -1,
        'num_leaves' : 30,
        'max_depth' : 4,
        'min_split_gain' : 0.07297767759639462,
        'min_child_weight' : 9.223836586454116,
        'min_child_samples' : 43,
        'subsample': 0.29967715403460243,
        'subsample_freq' : 1,
        'colsample_bytree' : 0.777005917777368,
        'reg_alpha' : 0.015583802542867243,
        'reg_lambda' : 0.21883161475419755,
        'verbosity' : -1,
        'seed' : 266
    }
stacker_boosting = Boosting(stack_train, stack_target, stack_test, params, random_state
stacker_boosting.train(booster = 'lightgbm')
```

```
Fitting the lightgbm on Training Data with 4 fold cross validation, and using Out-Of-Fol
ds Predictions for Cross-Validation

        Fold Number 1

[400]   training's auc: 0.808608        training's binary_logloss: 0.230945     valid_
1's auc: 0.804848       valid_1's binary_logloss: 0.231659
[800]   training's auc: 0.809359        training's binary_logloss: 0.229169     valid_
1's auc: 0.804915       valid_1's binary_logloss: 0.230363

        Fold Number 2

[400]   training's auc: 0.808453        training's binary_logloss: 0.230887     valid_
1's auc: 0.805412       valid_1's binary_logloss: 0.231861

        Fold Number 3

[400]   training's auc: 0.80782 training's binary_logloss: 0.231023     valid_1's auc:
0.807424        valid_1's binary_logloss: 0.231547
[800]   training's auc: 0.80851 training's binary_logloss: 0.229263     valid_1's auc:
0.807504        valid_1's binary_logloss: 0.230096

        Fold Number 4

[400]   training's auc: 0.807416        training's binary_logloss: 0.231241     valid_
1's auc: 0.808646       valid_1's binary_logloss: 0.231023
Done.
Time elapsed = 0:00:52.946849
```

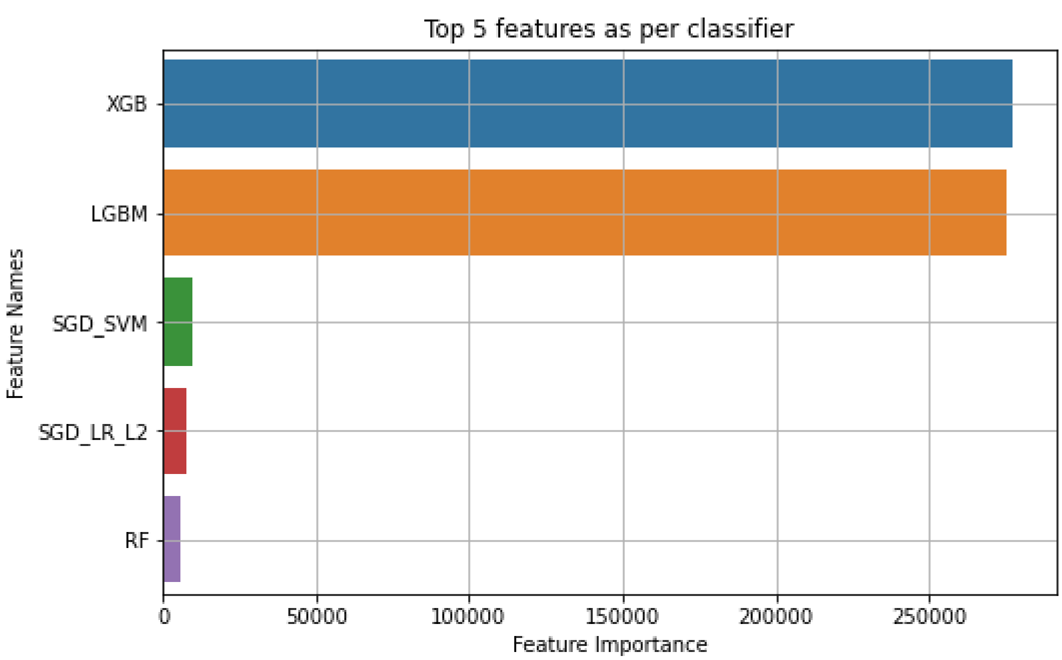
In [90]:
```
#displaying the results and metrics
stacker_boosting.results()
#displaying top 50 important features
stacker_boosting.feat_importances_show(5, figsize = (8,5))
```

```
==========================================================================================
============
Train Results:
```

```
The best selected Threshold as per the J-Statistic, which is J = TPR - FPR, is = 0.02726
2015818079285

        ROC-AUC Score = 0.8068788232594093
        Precision Score = 0.19050680825909755
        Recall Score = 0.7399798590130916
CV Results:
        ROC-AUC Score = 0.8049454509653551
        Precision Score = 0.1066937629256782
        Recall Score = 0.9580261832829808
==========================================================================================
============
Confusion, Precision and Recall Matrix on CV data:
```




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



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

In [91]:
```
stacking_model_df = pd.DataFrame({'SK_ID_CURR': skid_test, 'TARGET' : stacker_boosting.
stacking_model_df['SK_ID_CURR'] = stacking_model_df['SK_ID_CURR'].astype(int)
stacking_model_df.to_csv('./submissions/LGBM_Stacker_4_Fold.csv',index = False)
```

Model	Private Score	Public Score
LGBM_Stacker_4_Fold	0.79494	0.79702

title

Results Discussion

- From the stacking classifier, we see that the both the CV and Private Score obtained are the higher than all the base classifiers.
- We also observe that the recall value is very high, but the Precision value has gone down even further.
- This stacking model, however, performs worse than a single XGB model, this could be due to the low number of base models. Also the base classifiers might be correlated with each other too, so that may also have caused low performance with stacking.
- From the Feature Importance of LightGBM, we see that the LGBM base model's predictions score the highest among all the base classifiers, which is followed by XGB's predictions. The least importance comes out for the Random Forest Classifier's predictions.

2.11 Blending of Predictions

Based on the feature importances obtained from the Stacking Classifier, we will now try to blend the predictions of all classifiers, in a ratio proportion to their feature importances from above.

In [92]:
```
#getting the feature importance from stacking classifier
feat_imp_gain_stacking = stacker_boosting.feature_importance
#normalizing the gain values to lie between 0 to 1
feat_imp_gain_stacking.gain = feat_imp_gain_stacking.gain / sum(feat_imp_gain_stacking.
print("Normalized Gain Value as per Stacking Classifier for each base models are:")
display(feat_imp_gain_stacking)
```

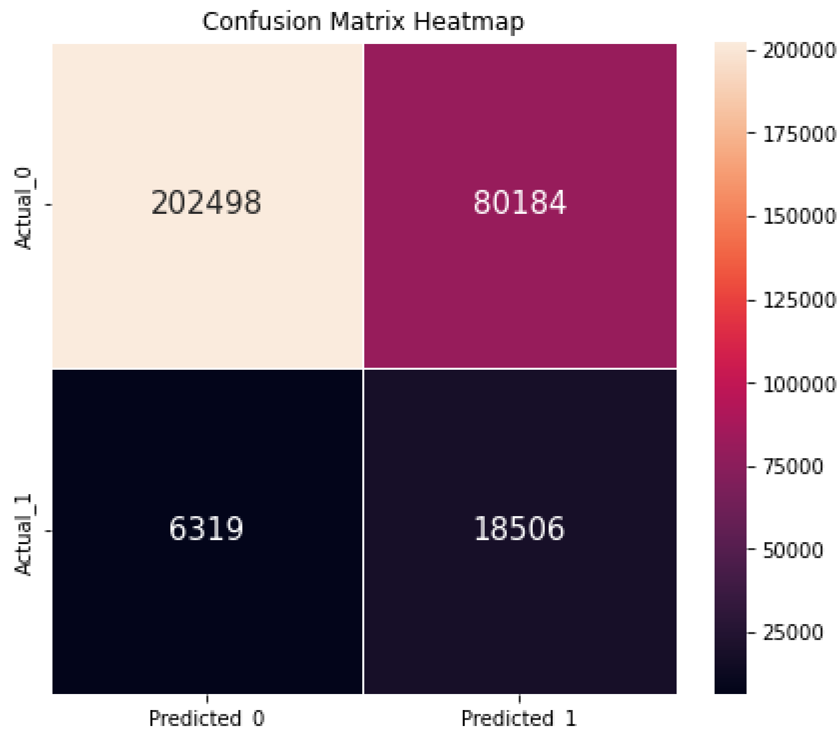
```
Normalized Gain Value as per Stacking Classifier for each base models are:
```

	features	gain
4	XGB	0.481779
0	LGBM	0.478200
3	SGD_SVM	0.016968
2	SGD_LR_L2	0.013509
1	RF	0.009545

In [93]:
```
blended_cv_preds = sgd_lr_l2_cv * 0.021874 + sgd_svm_cv * 0.024588 + rf_cv * 0.011525 +
```

```
blended_test_preds = sgd_lr_l2_test * 0.021874 + sgd_svm_test * 0.024588 + rf_test * 0.

#tuning the threshold again
fpr, tpr, threshold = roc_curve(target_train, blended_cv_preds)
j = tpr - fpr
best_threshold = threshold[np.argmax(j)]
#calculating the metrics values
print("Results on CV Dataset:")
print("="*100)
print(f"\nThe best selected Threshold as per the J-Statistic, which is J = TPR - FPR, i
print(f"ROC-AUC Score CV = {roc_auc_score(target_train, blended_cv_preds)}")
print(f"Precison Score CV = {precision_score(target_train, np.where(blended_cv_preds >=
print(f"Recall Score CV = {recall_score(target_train, np.where(blended_cv_preds >= best_

print('=' * 100)
print("Confusion Matrix of CV data:")
conf_mat = confusion_matrix(target_train, np.where(blended_cv_preds >= best_threshold,
conf_mat = pd.DataFrame(conf_mat, columns = ['Predicted_0','Predicted_1'], index = ['Ac
plt.figure(figsize = (7,6))
plt.title('Confusion Matrix Heatmap')
sns.heatmap(conf_mat, annot = True, fmt = 'g', linewidth = 0.5, annot_kws = {'size' : 1
plt.show()
print("="*100)
```

```
Results on CV Dataset:
====================================================================================================

The best selected Threshold as per the J-Statistic, which is J = TPR - FPR, is = 0.08695461862946152

ROC-AUC Score CV = 0.8063421157831652
Precison Score CV = 0.1875164657006789
Recall Score CV = 0.7454582074521652
====================================================================================================
Confusion Matrix of CV data:
```




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

In [94]:

```
blending_model_df = pd.DataFrame({'SK_ID_CURR': skid_test, 'TARGET' : blended_test_pred
blending_model_df['SK_ID_CURR'] = blending_model_df['SK_ID_CURR'].astype(int)
blending_model_df.to_csv('Blending_all.csv',index = False)
```

Model	Private Score	Public Score
Blending_all	0.79650	0.79910

title

Results Discussion

- We see that this blended model shows the second highest CV ROC-AUC among all models.
- It was a rather more heuristic process, as we just used the importance of each model obtained from Stacking Classifier as its weight to blend, and it turns out to give the best ROC-AUC among all the models so far, on Test Dataset.
- However, the Recall and Precision values are not that good. This might require rather more rigorous threshold tuning to improve the results a bit more.

3 Results Summarization and Conclusion

In [95]:

```
results_df = pd.DataFrame([['Random Model', np.round(0.502074521651561, 5), '-', np.rou
                          ['Dominant Class', 0.0, '-', np.round(0.5003946699208937, 5
                          ['SGD LogisticRegression L2', np.round(0.7383282980866063,5
                             np.round(0.7974418830512477,5), np.round(0.7908099381800
                          ['SGD Linear SVM', np.round(0.731238670694864, 5), np.round
                                    np.round(0.7982204969030176, 5), np.round(0.79170
                          ['RandomForestClassifier', np.round(0.7992749244712991, 5),
                                    np.round(0.8319575125937456, 5), np.round(0.77752
```

```
                                        ['XGBoostClassifier', np.round(0.7961329305135951, 5), np.r
                                                  np.round(0.868035603552344, 5), np.round(0.801593
                                        ['XGBoostClassifier - 600 Features', np.round(0.78296072507
                                                  np.round(0.8602796863474593, 5), np.round(0.80129
                                        ['LightGBMClassifier', np.round(0.8279556898288016, 5), np.
                                                  np.round(0.9165567214393182, 5), np.round(0.80591
                                        ['StackingClassifier', np.round(0.7245115810674723, 5), np.
                                                  np.round(0.8086869294476866, 5), np.round(0.80638
                                        ['Blending Results', '-', np.round(0.7270493454179254, 5),
                                                  0.79650, 0.79910]],
                                      columns = ['Model', 'Train Recall', 'CV Recall', 'Train ROC-A
                                                 'Private Score (ROC-AUC)', 'Public Score (ROC-AUC)

display(results_df)
```

	Model	Train Recall	CV Recall	Train ROC-AUC	CV ROC-AUC	Private Score (ROC-AUC)	Public Score (ROC-AUC)
1	Random Model	0.50207	-	0.50192	-	0.49735	0.48234
2	Dominant Class	0.0	-	0.50039	-	0.49054	0.50009
3	SGD LogisticRegression L2	0.73833	0.72032	0.79744	0.79081	0.77894	0.78355
4	SGD Linear SVM	0.73124	0.72342	0.79822	0.79171	0.77924	0.78397
5	RandomForestClassifier	0.79927	0.72036	0.83196	0.77752	0.76396	0.77103
6	XGBoostClassifier	0.79613	0.86606	0.86804	0.80159	0.79733	0.79921
7	XGBoostClassifier - 600 Features	0.78296	0.86453	0.86028	0.8013	0.79744	0.79768
8	LightGBMClassifier	0.82796	0.8224	0.91656	0.80591	0.79358	0.79470
9	StackingClassifier	0.72451	0.9395	0.80869	0.80639	0.79494	0.79702
10	Blending Results	-	0.72705	-	0.80602	0.79650	0.79910

From the above results, we observe that the:

- The highest Cross-Validation Recall value was obtained for Stacking Classifier Model, which is then followed by XGBoostClassifier Model.
- The Highest CV ROC-AUC was observed for StackingClassifier, followed by Blended Model, then LightGBM and XGBoost. They all are pretty close to each other.
- The best Private Score (ROC-AUC) was obtained for XGBoostClassifier-600 features Model
- Overall the Boosting Models out-performed the other models by a margin, as can be seen from both the CV and Private Score. However, the linear models like Logistic Regression and Linear SVM performed better thatn RandomForest Ensemble. This could be due to the high dimensionality of the dataset