Student ID: 251108648

Couse Code: FINMOD 9528A

Assignment Title: Coursework2 – Credit Risk Analytics

Word Count: 1999

For LGD modeling, since decision tree and xgboost would cost a great amount of time, it's better to tune the model with part of the dataset first. Therefore, I cut the dataset to one with only 5000 rows used as cleaning data and tuning models. After I got a satisfied accuracy from the model, I would apply the complete dataset then. However, for credit scoring application model, I would use the full dataset to do the dataset cleaning and model tuning from the beginning since logistic regression converged within a shorter time and data cleaning itself wouldn't take up too much RAM.

1. Data Cleaning:

Drop sparse columns:

Before correlation and outlier analysis, I roughly checked the given dataset, and drop columns with over 95% of null values. Since null values will cause error during correlation analysis, and model building, and we don't have robust estimate for too many null values within one column, so I decided to drop those columns, such as "member_id", "url", "desc" etc. The function that I use is "df.drop(columns = ["variable names"], axis=1, inplace=True)".

Imputation:

For columns with less than 5% of missing values, I used median of the rest of values in that column to replace the null values. After imputation, I dropped all the rows still containing null values.

Categorical variables:

For credit scoring application model, we will use weight of evidence transform, so it is not necessary to use dummy coding. However, for categorical variables with too many categories, I decided to change them into numerical variables to decrease complexity, such as emp_title. Also, categorical variables that had categories obviously could be divided into different levels could be converted into numerical data as well, like "emp_length", which represents the employment length in years. In common sense, longer employment length favors borrower to pay the loan with more savings.

On the other hand, there were some categorical variables could be simplified by only acquiring part of its information to decrease complexity of the modelling in later procedures. For example, categorical variable "earliest_cr_line" contains both months and years in it, so I cut the string to make the variable only store the year when the borrower's earliest reported credit line was opened. Similarly, I cut "zip_code" to contain only the first digit, which is the national code. In this way, it will also be easier for me to find pattern from those variables and make adjustment such as grouping some of categories later.

Specifically, for Loss Given Default modeling, I used function “crosstab” to check the occurrence of different LGD value corresponding to each category, and by observation, I could decide if several categories should be group into one new category.

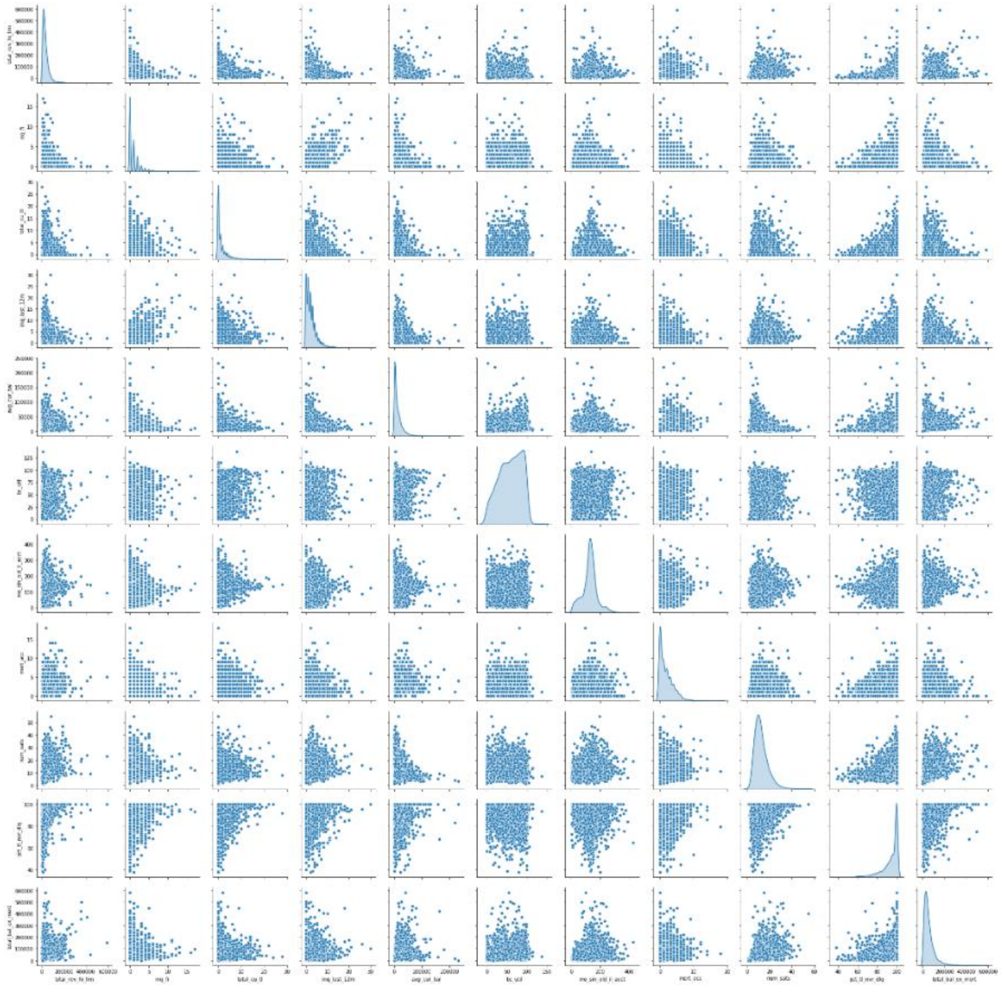
Finally, “loan_status”, as the respond variable in credit scoring application model, must be converted into binary number. I converted all category of “Charge off” into 1 and “Fully paid” into 0, and I dropped all the other categories since they could not be determined.

Initial variable selection:

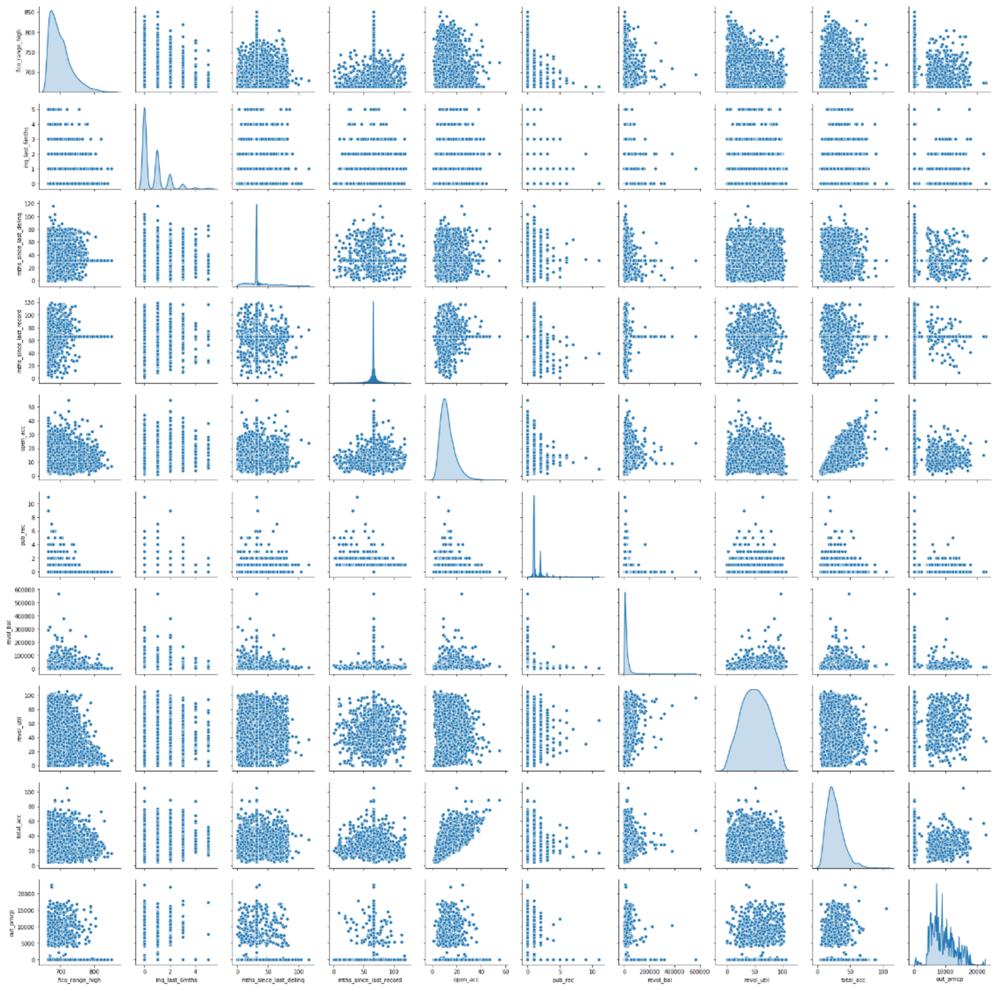
In the original dataset, we had 148 columns. After dropping off sparse columns, there were 100 columns left, still too many before correlation analysis. Therefore, I used expert judgement to roughly decide which ones were apparently redundant or useless for future modeling such as “Loan Title” and “title” and “zip_code” and “addr_state”. After doing this, I reduced the number of predictor variables to about 45, which was acceptable for me.

Correlation analysis:

Since the number of columns left was still very large in my current dataset, around 45, I thought it more convenient to use visualization with graphs. I used function “pairplots()” to plot scatterplots related each pair of categories as shown below.



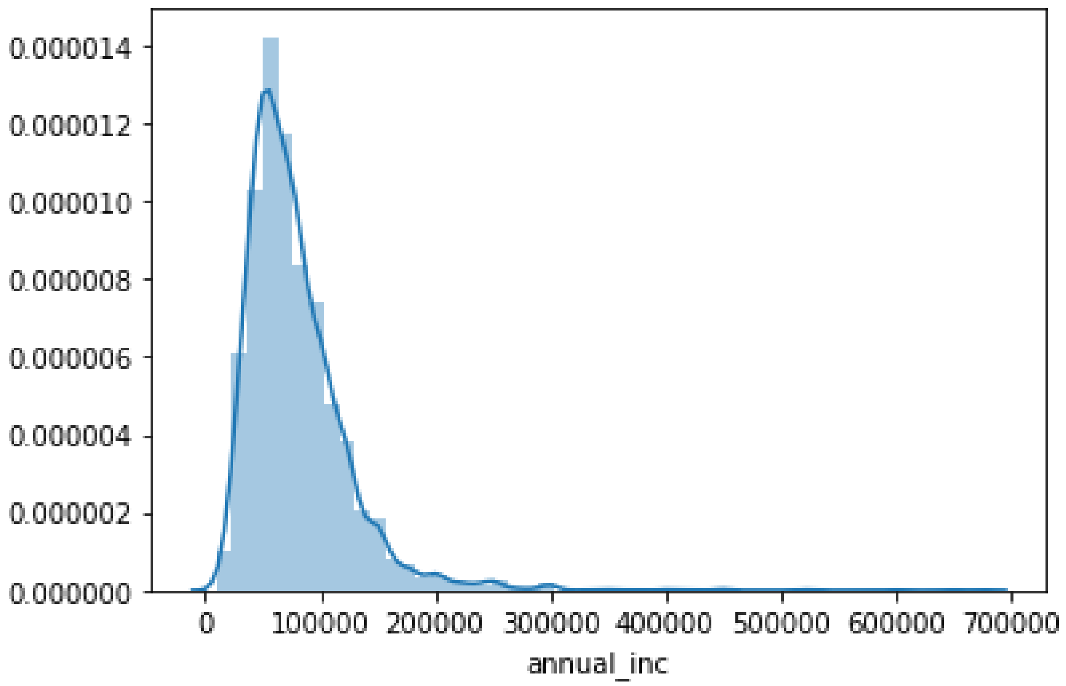
Since the number of variables is too large to plot all of them together, I plotted ten variables for each time at first, and after I dropped some of variables that had large correlation with others, I plotted rest of variables together to get an overall view. With small scale, I could quickly find the graphs in which points seemed in linear patterns which mean that pair of variables were probably highly correlated, then I zoomed in to find which variables were in suspension, and use function “corr()” to verify my assumptions.



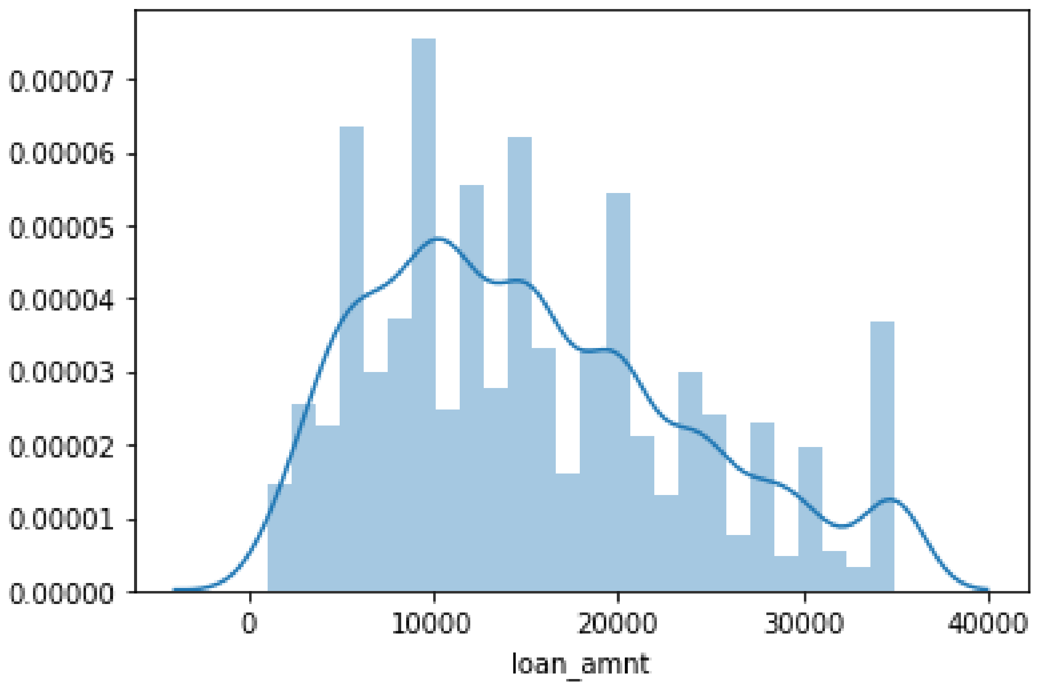
When I found one pair of variables highly correlated, I would observe the same row and same column where the plot located at to decide which variable had more correlation with others. The one with more correlation with other variables should be dropped off.

Outliers:

By drawing histogram for each left variable, I could observe the empirical distribution of one variable and if it had outliers that need to be dropped off. For example, look at the graphs shown below.



The distribution of “annual_inc” was positive skewed with long right tail, where outliers could be found.

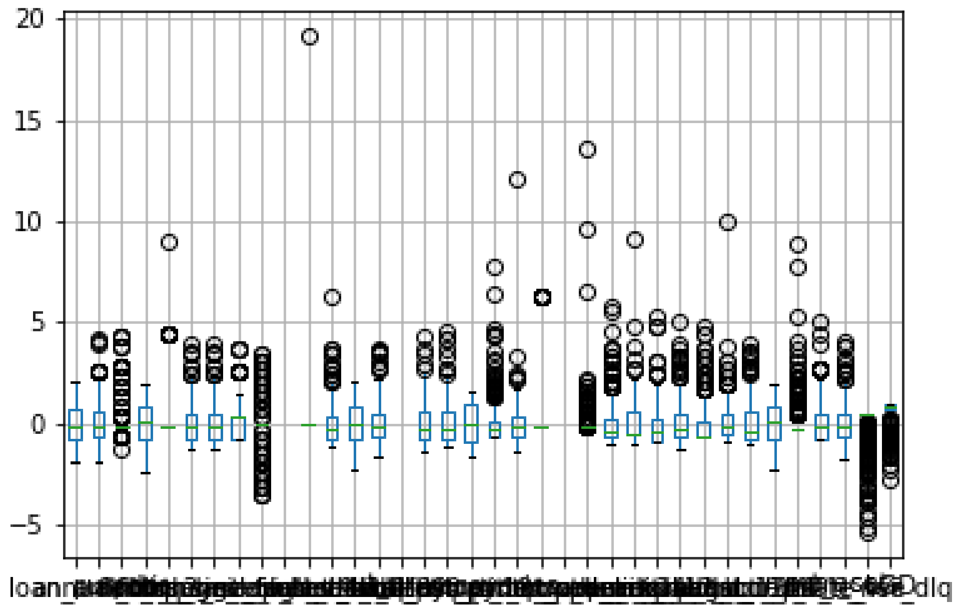


Similarly, the distribution of loan_amnt probably had outliers in its right tail.

In order to increase the accuracy of prediction, in most cases, we need drop off the outliers to make the model more robust. For those variables with long tails in its distribution plot, I used "[median – 3 * IQR, median + 3 * IQR]" as the outer fence, which would help me filter the outliers that were outside of the range. Note that median was the third quantile, and interquartile range was the difference of the fourth quantile and the second quantile.

Normalization:

For most model, it is better to normalize the dataset to reduce data redundancy and improve data integrity. Since we have weight of evidence transform, I only need normalize dataset when comes to model for LGD. In detail, I identified the numerical columns first and map function “zscore” to those columns. After normalization, the datapoints are more centered and in the same scale.



Dummy coding:

For LGD modeling, we need preprocess categorical data by ourselves instead of relying on WOE transform. First, I applied “crosstab” function and regroup some categorical variable like I mentioned above in part of “categorical variables”. Next, I used function from pandas “get_dummies(df)” to convert all the categorical variable into the form that could be provided to machine learning algorithms to do a better job in prediction.

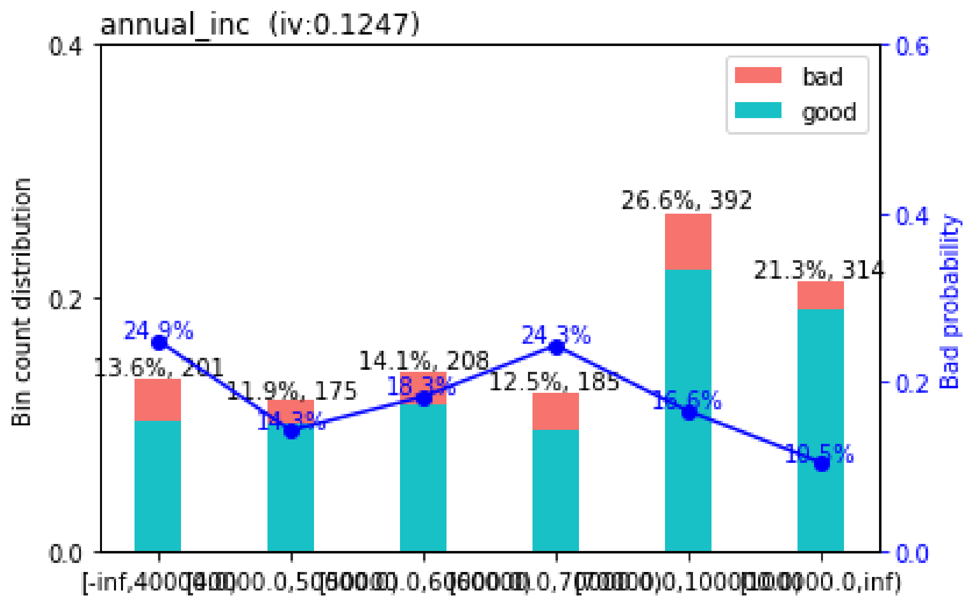
Since the dataset was very large, I assumed each group in categorical variables had enough number of occurrences after combining some of them, following the 5% rule.

2. Scorecard construction:

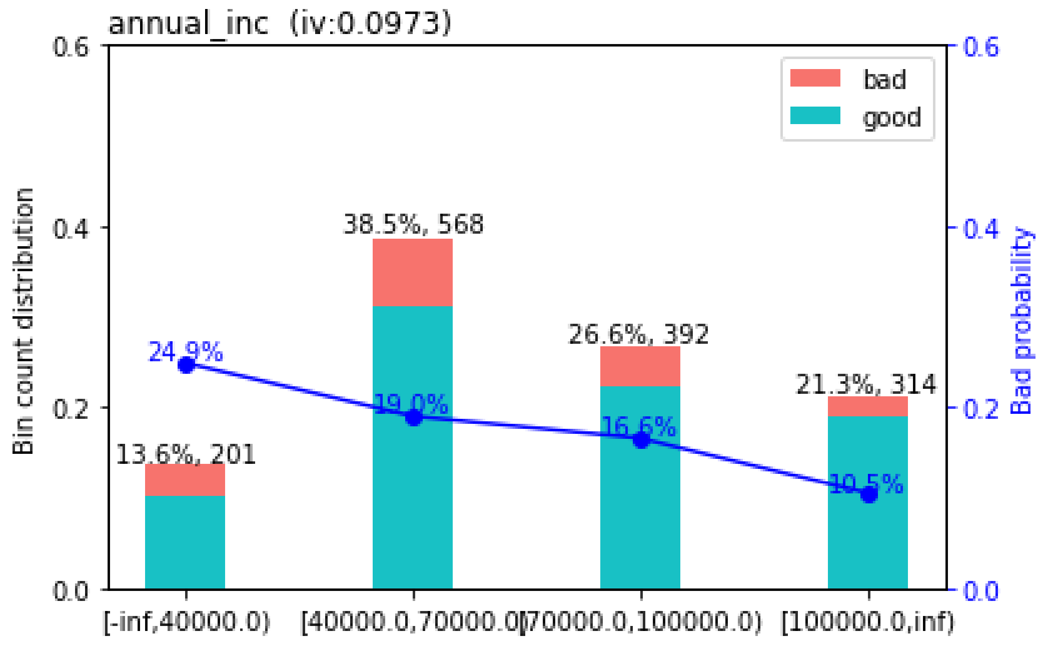
Weight of Evidence

Before putting the cleaned data into logistic regression model, I used weight of evidence transformation to select variables I needed. First, I use "scorecardpy.split_df" to split the dataset into train dataset and test dataset with ratio of 0.7. After splitting data, I binned the data in train dataset and plot bins with function "scorecardpy.woebin_plot()", by which I observed that the binning for variable "annual_Inc" could not make sense due to inconsistent trend. Therefore, I manually adjust the break to fix this problem.

Here is the binning plot before the adjustment:



Below is the binning plot after the adjustment:



As shown in the graph, we could understand that higher annual income had a negative impact on the probability of default, which was the same as our intuition. Note that the other variables in the dataset were binned with reasonable patterns, so I didn't adjust them. To get the woe transformation for each variable, I applied the woe bins on the original train dataset and test dataset.

Information value filter:

I could checked the information value by call the function "scorecardpy.iv(train_woe, 'loan_status')", and a list of iv along with the name of variables would be printed. By information value filter, I got five significant variables that had iv of larger than 0.1: "total_pymnt", "last_pymnt_amnt", "int_rate", "term", "last_fico_range_high". Moreover, both "total_pymnt" and "last_pymnt_amnt had iv of higher than 1. However, I then realized that the reason was both those variables contained post information that Lending Clube didn't know when they lent the loan, so I dropped them.

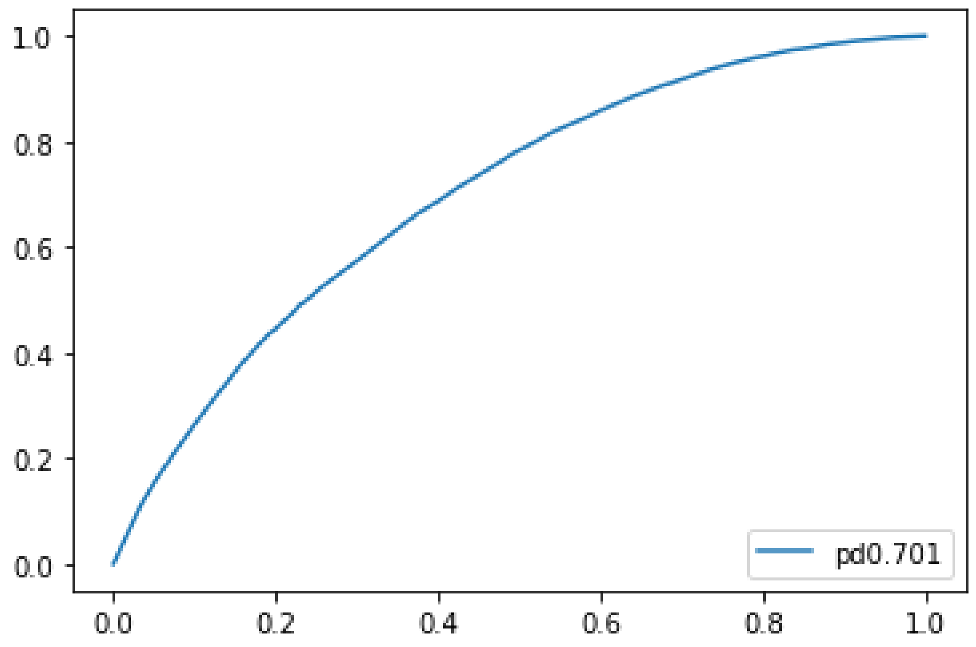
Initially, I only include the left three variables above in the logistic regression, and then I thought of that I could also include some variables with little lower iv than 0.1 since the sample size was quite large in the full dataset. Therefore, I also include revol_util and bc_util whose iv were around 0.77

Model tuning:

The major parameters that could be tuned in logistic regression were "max_iter" and "C", where "max_iter" represented maximum number of iterations taken for the solvers to converge, and "C" was the inverse of regularization strength. Moreover, I set "n_jobs"=-1 such that I could use all the processor and run the model faster. Since the speed of fitting model was not bad, I directly used train dataset from full dataset to tune the model. My optimal parameter was that "C" = 0.7 and "max_iter" = 300.

Result:

I applied the fitted model on test dataset, and got y score through function "pd_logreg.predict_proba". AUC was calculated by function "roc_auc_score" to be 0.701. Here was my ROC curve:



3. Loss Given Default model:

Before starting to build the models, I calculated LGD for each record of borrower using the follow formula:

$\frac{loan\ amount - total\ payment - recoveries - total\ rec\ late\ fee}{loan\ amount - total\ payment}$, where the numerator was the total exposure if default happens (loss), and the numerator was the default.

Random forest:

While tuning, I used a sample dataset with only 10% of the original one. Also, the ratio between train dataset and test dataset was 0.7. Following was my procedure of tuning:

1. I created a pipeline using function “Pipeline ()”, in which I putted random forest regressor with a initial set of parameters given my experience
2. I created three more similar pipeline, but each differed from the first one on one or two sub-parameters, such as “n_estimators”, “min_sample_leaf” and “max_features”
3. I inputted each pipeline in cross validation score function in fold of 5 with mean square error as the scoring method
4. Printed the mean of cross-validation score for each model and compared their score.
5. Repeated the steps above until I couldn’t find models with better cv-score.

Finally, my best result comes with “n_estimator” = 15000, “min_samples_leaf” = 0.003, and “max_features” = ‘sqrt’.

Xgboost:

The process of tuning for xgboost was similar to what I did on random forest regressor, but parameters were slightly different. I used “sklearn.GradientBoostingRegressor()” in cross validation in fold of 5, and tried to find the optimal set of parameters by tuning on “learning_rate”, “n_estimator”, “min_samples_leaf”. Note that as I increased “n_estimator”, I needed to decrease the learning rate.

At last, my best result comes with “learning_rate” =0.07, “min_samples_leaf” =20 and “n_estimator” =15000.

Comparison:

Using mean square error, and applying the tuned model on full dataset, I got 217.19313 of MSE for random forest, and 337.5984 of that for xgboost. By comparison, the performance of random forest was superior to that of xgboost. Since xgboost was more suitable for small dataset, the result just met my expectation.

What’s more, the variables of importance were different between these two models. There was only one variable with lager than 0.1 of feature importance xgboost, which was “total_pymnt”. In contrast there were three important features in total for random forest, which were “total_pymnt”, “recoveries” and “total_rec_late_fee”. The reason causing this difference, from my view, was due to that xgboost was more sensitive to overfitting if the data is noisy, so it reduces the feature importance for almost all the variables.

Link: https://colab.research.google.com/drive/1aEaUM_4u6POnMpq9dPHLPpgPq02NeY3x

Appendix:

```
# -*- coding: utf-8 -*-

"""

Created on Fri Nov  1 21:08:40 2019



@author: Yitia

"""

import pandas as pd

import numpy as np

import scorecardpy as sc

import seaborn as sns

from numpy import cov

from sklearn.ensemble import RandomForestRegressor

from sklearn.linear_model import LogisticRegression

import copy
```

```
from sklearn.metrics import confusion_matrix

import matplotlib.pyplot as plt

from sklearn.metrics import roc_auc_score, confusion_matrix, roc_curve

from sklearn.ensemble import RandomForestRegressor

from sklearn.model_selection import cross_val_score

from sklearn.pipeline import Pipeline

from sklearn.metrics import mean_squared_error, mean_absolute_error,
make_scorer

from sklearn.ensemble import GradientBoostingRegressor

%matplotlib inline



# a)

LC_part = pd.read_csv('LCFinal.csv')



# remove loans that hadn't have enough time to default

LC_part = LC_part.loc[LC_part['loan_status'] != 'Current' ]
LC_part = LC_part.loc[LC_part['loan_status'] != 'In Grace Period' ]

LC_part = LC_part.loc[LC_part['loan_status'] != 'Late (16-30 days)' ]

LC_part = LC_part.loc[LC_part['loan_status'] != 'Late (31-120 days)' ]

LC_part = LC_part.loc[LC_part['loan_status'] != 'Default' ]

LC_part.loc[LC_part['loan_status'] == 'Fully Paid', 'loan_status'] = 0

LC_part.loc[LC_part['loan_status'] == 'Charged Off', 'loan_status'] = 1




# drop varible by common sense and observation

LC_part.drop(LC_part.iloc[:,129:142],axis=1,inplace=True) # too sparsed

LC_part.drop(LC_part.iloc[:,144:],axis=1,inplace=True) #too sparsed

LC_part = LC_part.drop(columns=['member_id','pymnt_plan','url',

                   'desc','annual_inc_joint','dti_joint',

                   'verification_status_joint','revol_bal_joint',

                   'sec_app_fico_range_low','sec_app_fico_range_high',
                   'sec_app_earliest_cr_line','sec_app_inq_last_6mths',

                   'sec_app_mort_acc','sec_app_open_acc',

                   'sec_app_revol_util','sec_app_open_act_il',

                   'sec_app_num_rev_accts','sec_app_chargeoff_within_12_mths',


'sec_app_collections_12_mths_ex_med','sec_app_mths_since_last_major_derog'

,

                   'hardship_type','hardship_reason',

                   'hardship_status', 'debt_settlement_flag_date',

                   'settlement_status', 'settlement_date',

                   'settlement_amount', 'settlement_percentage',

                   'settlement_term', 'next_pymnt_d',

                   'mths_since_recent_revol_delinq', 'mths_since_recent_inq',

                   'mths_since_recent_bc_dlq', 'issue_d'])

      # issue_d is all same for each loan
# deal with missing value


# replace missing value with median if the median value only accounts for less
than 5%
mths_since_last_delinq_median =
np.nanmedian(LC_part["mths_since_last_delinq"])
LC_part['mths_since_last_delinq'].fillna(value=mths_since_last_delinq_median,
inplace=True)


mths_since_last_record_median =
np.nanmedian(LC_part["mths_since_last_record"])
LC_part['mths_since_last_record'].fillna(value=mths_since_last_record_median,
inplace=True)


mths_since_recent_bc_median =
np.nanmedian(LC_part["mths_since_recent_bc"])
```

```
LC_part['mths_since_recent_bc'].fillna(value=mths_since_recent_bc_median,
inplace=True)


il_util_median = np.nanmedian(LC_part["il_util"])
LC_part['il_util'].fillna(value=il_util_median, inplace=True)



mo_sin_old_il_acct_median = np.nanmedian(LC_part["mo_sin_old_il_acct"])
LC_part['mo_sin_old_il_acct'].fillna(value=mo_sin_old_il_acct_median,
inplace=True)


num_tl_120dpd_2m_median = np.nanmedian(LC_part["num_tl_120dpd_2m"])
LC_part['num_tl_120dpd_2m'].fillna(value=num_tl_120dpd_2m_median,
inplace=True)


percent_bc_gt_75_median = np.nanmedian(LC_part["percent_bc_gt_75"])
```

```
LC_part['percent_bc_gt_75'].fillna(value=percent_bc_gt_75_median,

inplace=True)




# drop rows containing null value since number of nan is small

#LC_part = LC_part.dropna()




#deal with categorical variable



#emp_title

LC_part['emp_title'] = LC_part.emp_title.astype("category").cat.codes



# mths_since_last_major_derog (too sparsed)

LC_part = LC_part.drop(columns=["mths_since_last_major_derog"])
```

```
#term

#LC_part['term'] = LC_part.term.astype("category").cat.codes



#emp_length

#LC_part.emp_length = LC_part.emp_length.astype("category").cat.codes

LC_part.loc[LC_part['emp_length'] == '< 1 year', 'emp_length'] = 1

LC_part.loc[LC_part['emp_length'] == '1 year', 'emp_length'] = 2

LC_part.loc[LC_part['emp_length'] == '2 years', 'emp_length'] = 3

LC_part.loc[LC_part['emp_length'] == '3 years', 'emp_length'] = 4

LC_part.loc[LC_part['emp_length'] == '4 years', 'emp_length'] = 5

LC_part.loc[LC_part['emp_length'] == '5 years', 'emp_length'] = 6

LC_part.loc[LC_part['emp_length'] == '6 years', 'emp_length'] = 7

LC_part.loc[LC_part['emp_length'] == '7 years', 'emp_length'] = 8

LC_part.loc[LC_part['emp_length'] == '8 years', 'emp_length'] = 9

LC_part.loc[LC_part['emp_length'] == '9 years', 'emp_length'] = 10
LC_part.loc[LC_part['emp_length'] == '10+ years', 'emp_length'] = 11



#home_ownership (three levels)



#issue_d (number of levels is small)



# loan_status

#LC_part.loan_status = LC_part.loan_status.astype("category").cat.codes



# purpose

LC_part.purpose = LC_part.purpose.astype("category").cat.codes



# title

#LC_part.title = LC_part.title.astype("category").cat.codes
```

```
# zip_code

LC_part.zip_code = LC_part.zip_code.str[:1]



# addr_state

LC_part.addr_state = LC_part.addr_state.astype("category").cat.codes



# earliest_cr_line (only accounts for years)

LC_part.earliest_cr_line = LC_part.earliest_cr_line.str[4:]



# initial_list_status (two levels)

#LC_part.initial_list_status =

LC_part.initial_list_status.astype("category").cat.codes



LC_part = LC_part.dropna()
```

```
LC_part['loan_status'] = LC_part['loan_status'].astype('int')

#pick variables that related to modeling probability of default

# loan_amnt, funded_amnt, funded_amnt_inv, home_ownership, annual_inc

# loan_status, purpose, title, dti, delinq_2yrs,

# earliest_cr_line, fico_range_low, fico_range_high,
inq_last_6mths,mths_since_last_delinq,

# mths_since_last_record, open_acc, pub_rec, revol_bal, revol_util, total_acc

# out_prncp, out_prncp_inv, hardship_flag, application_type,

# total_pymnt, total_pymnt_inv, total_rec_prncp, total_rec_prncp

# recoveries, last_pymnt_amnt, collections_12_mths_ex_med,

# acc_now_delinq, tot_coll_amt, open_acc_6m, open_act_il, open_il_24m,

# il_util, open_rv_12m, max_bal_bc, total_rev_hi_lim, inq_fi

# total_cu_tl, inq_last_12m, avg_cur_bal, bc_open_to_buy, bc_util,

# mo_sin_old_il_acct, mort_acc, num_sats, pct_tl_nvr_dlq, total_bal_ex_mort
# correlation analysis

LC_pd = LC_part[['loan_amnt','funded_amnt_inv','home_ownership','annual_inc',

            'loan_status','purpose','title','dti','delinq_2yrs','term',



'earliest_cr_line','fico_range_high','inq_last_6mths','mths_since_last_delinq',



'mths_since_last_record','open_acc','pub_rec','revol_bal','revol_util','total_acc',

            'out_prncp','out_prncp_inv','application_type','int_rate',

            'total_pymnt','total_rec_prncp','zip_code','emp_title',

            'recoveries','last_pymnt_amnt','collections_12_mths_ex_med',

            'acc_now_delinq','tot_coll_amt','open_act_il','open_il_24m',

            'il_util','open_rv_12m','max_bal_bc','total_rev_hi_lim','inq_fi',

            'total_cu_tl','inq_last_12m','avg_cur_bal','bc_util',



'mo_sin_old_il_acct','mort_acc','num_sats','pct_tl_nvr_dlq','total_bal_ex_mort']]
```

```
   # temporarily change categorical variable to numeric

#LC_pd_temp = LC_pd

#LC_pd_temp['home_ownership'] =

LC_pd_temp.home_ownership.astype("category").cat.codes

#LC_pd_temp['application_type'] =

LC_pd_temp.application_type.astype("category").cat.codes



   # draw plots

   # first five variables

sns.pairplot(LC_pd_temp.iloc[:,:5], diag_kind="kde")

   # will remove funded_amount_inv (correlated with loan_amnt)

   # will remove title (highly correlated with purpose)

   # next five variables

sns.pairplot(LC_pd_temp.iloc[:,5:11], diag_kind="kde")

   # next ten avaribles

sns.pairplot(LC_pd_temp.iloc[:,11:21], diag_kind="kde")
```

```
    # will remove open_acc (correlated with total_acc)

    # next five avariables

sns.pairplot(LC_pd_temp.iloc[:,21:26], diag_kind="kde")

    # will remove out_prncp_inv (highly correlated with total_pymnt)

    # next ten avariables

sns.pairplot(LC_pd_temp.iloc[:,26:36], diag_kind="kde")

    # will remove il_util (highly correlated with open_act_il and open_il_24m)

    # rest variables

sns.pairplot(LC_pd_temp.iloc[:,36:], diag_kind="kde")

    # will remove inq_fi(highly correlated with inq_last_12m and pct_ti_nvr_diq

    # and total_bar_ex_mort)

    # will remove total_bal_ex_mort (highly correlated with many other variables)

    # will remove total_rev_hi_lim (highly correlated with many other variables)

    # will remove recoveries (highly correlated with many other variables)

    # will remove mo_sin_old_il_acct
```

```
LC_pd.drop(columns=['funded_amnt_inv','open_acc','out_prncp_inv',
                    'il_util','inq_fi','total_bal_ex_mort','total_rev_hi_lim',
                    'mo_sin_old_il_acct', 'title'], inplace=True)



    # plot pairplots for each of rest variables
LC_pd_temp = LC_pd
sns.pairplot(LC_pd_temp.drop(LC_pd_temp.columns[[24,25]],axis=1),
diag_kind="kde")
    # remove out_prncp (highly correlated with total_pymnt)
    # remove total_rec_prncp (highly correlated with many other variables)
    # remove pct_tl_nvr_dlq
    # remove max_bal_bc
LC_pd.drop(columns=['out_prncp','total_rec_prncp','pct_tl_nvr_dlq','max_bal_bc'
],inplace=True)
LC_pd_temp = LC_pd
```

```
# check outliers

sns.distplot(LC_pd['loan_amnt'])

sns.distplot(LC_pd['annual_inc'])

sns.distplot(LC_pd['dti'])

sns.distplot(LC_pd['mths_since_last_delinq'])

sns.distplot(LC_pd['mths_since_last_record'])

sns.distplot(LC_pd['avg_cur_bal'])
```

```
    # find interquantiles for annual_inc

    annual_inc_q1 = np.percentile(LC_pd.annual_inc,25,interpolation='midpoint')

    annual_inc_q2 = np.percentile(LC_pd.annual_inc,75,interpolation='midpoint')

    IQR_annual_inc = annual_inc_q2 - annual_inc_q1
    # strim

    lower_annual_inc = np.median(LC_pd.annual_inc)-3*IQR_annual_inc

    upper_annual_inc = np.median(LC_pd.annual_inc)+3*IQR_annual_inc

    LC_pd = LC_pd.loc[(LC_pd['annual_inc'] >= lower_annual_inc) &

                 (LC_pd['annual_inc'] <= upper_annual_inc)]




    # find interquantiles for mths_since_last_delinq

    mths_since_last_delinq_q1 =

np.percentile(LC_pd.mths_since_last_delinq,25,interpolation='midpoint')

    mths_since_last_delinq_q2 =

np.percentile(LC_pd.mths_since_last_delinq,75,interpolation='midpoint')

    IQR_mths_since_last_delinq = mths_since_last_delinq_q2 -

mths_since_last_delinq_q1

    # strim
    lower_mths_since_last_delinq = np.median(LC_pd.mths_since_last_delinq)-
3*IQR_mths_since_last_delinq

    upper_mths_since_last_delinq =
np.median(LC_pd.mths_since_last_delinq)+3*IQR_mths_since_last_delinq

    LC_pd = LC_pd.loc[(LC_pd['mths_since_last_delinq'] >=
lower_mths_since_last_delinq) &

                      (LC_pd['mths_since_last_delinq'] <= upper_mths_since_last_delinq)]



    # find interquantiles for mths_since_last_record

    mths_since_last_record_q1 =
np.percentile(LC_pd.mths_since_last_record,25,interpolation='midpoint')

    mths_since_last_record_q2 =
np.percentile(LC_pd.mths_since_last_record,75,interpolation='midpoint')

    IQR_mths_since_last_record = mths_since_last_record_q2 -
mths_since_last_record_q1

    # strim
    lower_mths_since_last_record = np.median(LC_pd.mths_since_last_record)-
3*IQR_mths_since_last_record
    upper_mths_since_last_record =
np.median(LC_pd.mths_since_last_record)+3*IQR_mths_since_last_record
    LC_pd = LC_pd.loc[(LC_pd['mths_since_last_record'] >=
lower_mths_since_last_record) &
                   (LC_pd['mths_since_last_record'] <= upper_mths_since_last_record)]


    # find interquantiles for avg_cur_bal
    avg_cur_bal_q1 =
np.percentile(LC_pd.avg_cur_bal,25,interpolation='midpoint')
    avg_cur_bal_q2 =
np.percentile(LC_pd.avg_cur_bal,75,interpolation='midpoint')
    IQR_avg_cur_bal = avg_cur_bal_q2 - avg_cur_bal_q1
    # strim
    lower_avg_cur_bal = np.median(LC_pd.avg_cur_bal)-3*IQR_avg_cur_bal
    upper_avg_cur_bal = np.median(LC_pd.avg_cur_bal)+3*IQR_avg_cur_bal

    LC_pd = LC_pd.loc[(LC_pd['avg_cur_bal'] >= lower_avg_cur_bal) &

                      (LC_pd['avg_cur_bal'] <= upper_avg_cur_bal)]




# find interquantile for int_rate

int_rate_q1 = np.percentile(LC_pd.int_rate,25,interpolation='midpoint')

int_rate_q2 = np.percentile(LC_pd.int_rate,75,interpolation='midpoint')

IQR_int_rate = int_rate_q2 - int_rate_q1

# strim

lower_int_rate = np.median(LC_pd.int_rate)-3*IQR_int_rate

upper_int_rate = np.median(LC_pd.int_rate)+3*IQR_int_rate

LC_pd = LC_pd.loc[(LC_pd['int_rate'] >= lower_int_rate) &

                      (LC_pd['int_rate'] <= upper_int_rate)]
```

```
# data cleaning for LGD

# filter rows


#LC = LC_part.iloc[:50000,:]

LC = LC_part



LC_afterfilter =

LC[['loan_amnt','funded_amnt_inv','home_ownership','annual_inc',

               'loan_status','purpose','title','dti','delinq_2yrs', 'emp_length',


'earliest_cr_line','fico_range_low','fico_range_high','inq_last_6mths','mths_since

_last_delinq',
'mths_since_last_record','open_acc','pub_rec','revol_bal','revol_util','total_acc',

                'out_prncp','out_prncp_inv','application_type', 'int_rate',

                'total_pymnt','total_rec_prncp','zip_code','emp_title',

                'recoveries','last_pymnt_amnt','collections_12_mths_ex_med',

                'acc_now_delinq','tot_coll_amt','open_act_il','open_il_24m',

                'il_util','open_rv_12m','max_bal_bc','total_rev_hi_lim','inq_fi',

                'total_cu_tl','inq_last_12m','avg_cur_bal','bc_util', 'total_rec_late_fee',


'mo_sin_old_il_acct','mort_acc','num_sats','pct_tl_nvr_dlq','total_bal_ex_mort']]



LC_lgd = LC_afterfilter.loc[LC_part['loan_status'] == 1]



# calculate LGD

LC_lgd['LGD'] = (LC_lgd.loan_amnt-LC_lgd.total_pymnt-LC_lgd.recoveries-

LC_lgd.total_rec_late_fee)/(
    LC_lgd.loan_amnt-LC_lgd.total_pymnt)



LC_lgd.loc[LC_lgd['LGD'] > 1,'LGD'] = 1

# Normalization for LC_lgd

from scipy.stats import zscore



#correlation

LC_lgd.drop(columns=['funded_amnt_inv','open_acc','out_prncp_inv',

                'il_util','inq_fi','total_bal_ex_mort','total_rev_hi_lim',

                'mo_sin_old_il_acct', 'title', 'loan_status','fico_range_low'],

inplace=True)

LC_lgd.drop(columns=['out_prncp','total_rec_prncp','pct_tl_nvr_dlq','max_bal_bc

'],inplace=True)

LC_lgd.dropna(axis=0,inplace=True)
```

```
# outliers

LC_lgd = LC_lgd.loc[(LC_lgd['annual_inc'] >= lower_annual_inc) &

                    (LC_lgd['annual_inc'] <= upper_annual_inc)]




LC_lgd = LC_lgd.loc[(LC_lgd['mths_since_last_delinq'] >=

lower_mths_since_last_delinq) &

                    (LC_lgd['mths_since_last_delinq'] <= upper_mths_since_last_delinq)]




LC_lgd = LC_lgd.loc[(LC_lgd['mths_since_last_record'] >=

lower_mths_since_last_record) &

                    (LC_lgd['mths_since_last_record'] <= upper_mths_since_last_record)]




LC_lgd = LC_lgd.loc[(LC_lgd['avg_cur_bal'] >= lower_avg_cur_bal) &
```

```
                    (LC_lgd['avg_cur_bal'] <= upper_avg_cur_bal)]




LC_lgd = LC_lgd.loc[(LC_lgd['int_rate'] >= lower_int_rate) &

                    (LC_lgd['int_rate'] <= upper_int_rate)]



# select all numeric columns

numeric_cols = LC_lgd.select_dtypes(include=[np.number]).columns

numeric_cols = numeric_cols[:-1]



#apply the zscore function to all data

LC_lgd[numeric_cols] = LC_lgd[numeric_cols].apply(zscore)



LC_lgd.boxplot()
```

```
# dummy coding

#home_ownership



pd.crosstab(LC_lgd['home_ownership'], LC_lgd['LGD'])



pd.crosstab(LC_lgd['application_type'], LC_lgd['LGD'])



pd.crosstab(LC_lgd['earliest_cr_line'], LC_lgd['LGD'])



LC_lgd_temp = copy.copy(LC_lgd)

LC_lgd.loc[LC_lgd_temp['earliest_cr_line'].astype(int) < 1990, 'earliest_cr_line'] =
'early'

LC_lgd.loc[LC_lgd_temp['earliest_cr_line'].astype(int) >= 1990, 'earliest_cr_line'] =
'late'
```

```
pd.crosstab(LC_lgd['zip_code'], LC_lgd['LGD'])

LC_lgd.loc[(LC_lgd_temp['zip_code'].astype(int) < 3) |

(LC_lgd_temp['zip_code'].astype(int) == 4) |

        ((LC_lgd_temp['zip_code'].astype(int) >= 6) &

(LC_lgd_temp['zip_code'].astype(int) <= 8)), 'zip_code'] = 'zip_mid'

LC_lgd.loc[(LC_lgd_temp['zip_code'].astype(int) == 5), 'zip_code'] = 'zip_low'

LC_lgd.loc[(LC_lgd_temp['zip_code'].astype(int) == 3) |

(LC_lgd_temp['zip_code'].astype(int) == 9), 'zip_code'] = 'zip_high'



pd.crosstab(LC_lgd['emp_length'], LC_lgd['LGD'])

LC_lgd.loc[LC_lgd_temp['emp_length'] <= 7, 'emp_length'] = 'emp_length_mid'

LC_lgd.loc[(LC_lgd_temp['emp_length'] > 7) & (LC_lgd_temp['emp_length'] <=

10), 'emp_length'] = 'emp_length_low'

LC_lgd.loc[LC_lgd_temp['emp_length'] == 11, 'emp_length'] = 'emp_length_hi'
```

```
# dummy_variable

LC_lgd = pd.get_dummies(LC_lgd)

LC_lgd.describe()
```

```
#Q3

LC_lgd.dropna(axis=1,inplace=True)

train_lgd, test_lgd = sc.split_df(LC_lgd, y = 'LGD', ratio = 0.7, seed =

251108648).values()
```

```
# cv

cv_model1 = Pipeline([('rf', RandomForestRegressor(n_estimators=10000, #
Number of trees to train
                    criterion='mse', # How to train the trees. Also supports entropy.
                    max_depth=None, # Max depth of the trees. Not necessary to
change.
                    min_samples_split=2, # Minimum samples to create a split.
                    min_samples_leaf=0.001, # Minimum samples in a leaf. Accepts
fractions for %. This is 0.1% of sample.
                    min_weight_fraction_leaf=0.0, # Same as above, but uses the class
weights.
                    max_features='auto', # Maximum number of features per split (not
tree!) by default is sqrt(vars)
```

```
                    max_leaf_nodes=None, # Maximum number of nodes.

                    min_impurity_decrease=0.0001, # Minimum impurity decrease. This
is 10^-3.

                    bootstrap=True, # If sample with repetition. For large samples
(>100.000) set to false.

                    oob_score=True,  # If report accuracy with non-selected cases.

                    n_jobs=-1, # Parallel processing. Set to -1 for all cores. Watch your
RAM!!

                    random_state=251108648, # Seed

                    verbose=1, # If to give info during training. Set to 0 for silent training.

                    warm_start=False, # If train over previously trained tree.

                    ))])


cv_model2 = Pipeline([('rf', RandomForestRegressor(n_estimators=10000, #
Number of trees to train

                    criterion='mse', # How to train the trees. Also supports entropy.
```

```
                  max_depth=None, # Max depth of the trees. Not necessary to
change.
                  min_samples_split=2, # Minimum samples to create a split.
                  min_samples_leaf=0.002, # Minimum samples in a leaf. Accepts
fractions for %. This is 0.1% of sample.
                  min_weight_fraction_leaf=0.0, # Same as above, but uses the class
weights.
                  max_features='auto', # Maximum number of features per split (not
tree!) by default is sqrt(vars)
                  max_leaf_nodes=None, # Maximum number of nodes.
                  min_impurity_decrease=0.0001, # Minimum impurity decrease. This
is 10^-3.
                  bootstrap=True, # If sample with repetition. For large samples
(>100.000) set to false.
                  oob_score=True,  # If report accuracy with non-selected cases.
                  n_jobs=-1, # Parallel processing. Set to -1 for all cores. Watch your
RAM!!
```

```
          random_state=251108648, # Seed

          verbose=1, # If to give info during training. Set to 0 for silent training.

          warm_start=False, # If train over previously trained tree.

          ))])


cv_model3 = Pipeline([('rf', RandomForestRegressor(n_estimators=15000, #
Number of trees to train

          criterion='mse', # How to train the trees. Also supports entropy.

          max_depth=None, # Max depth of the trees. Not necessary to
change.

          min_samples_split=2, # Minimum samples to create a split.

          min_samples_leaf=0.001, # Minimum samples in a leaf. Accepts
fractions for %. This is 0.1% of sample.

          min_weight_fraction_leaf=0.0, # Same as above, but uses the class
weights.
```

```
            max_features='auto', # Maximum number of features per split (not tree!) by default is sqrt(vars)
            max_leaf_nodes=None, # Maximum number of nodes.
            min_impurity_decrease=0.0001, # Minimum impurity decrease. This is 10^-3.
            bootstrap=True, # If sample with repetition. For large samples (>100.000) set to false.
            oob_score=True,  # If report accuracy with non-selected cases.
            n_jobs=-1, # Parallel processing. Set to -1 for all cores. Watch your RAM!!
            random_state=251108648, # Seed
            verbose=1, # If to give info during training. Set to 0 for silent training.
            warm_start=False, # If train over previously trained tree.
            ))])
```

```
cv_model4 = Pipeline([('rf', RandomForestRegressor(n_estimators=15000, #
Number of trees to train
                         criterion='mse', # How to train the trees. Also supports entropy.
                         max_depth=None, # Max depth of the trees. Not necessary to
change.
                         min_samples_split=2, # Minimum samples to create a split.
                         min_samples_leaf=0.001, # Minimum samples in a leaf. Accepts
fractions for %. This is 0.1% of sample.
                         min_weight_fraction_leaf=0.0, # Same as above, but uses the class
weights.
                         max_features='sqrt', # Maximum number of features per split (not
tree!) by default is sqrt(vars)
                         max_leaf_nodes=None, # Maximum number of nodes.
                         min_impurity_decrease=0.0001, # Minimum impurity decrease. This
is 10^-3.
                         bootstrap=True, # If sample with repetition. For large samples
(>100.000) set to false.
```

```
                oob_score=True,  # If report accuracy with non-selected cases.

                n_jobs=-1, # Parallel processing. Set to -1 for all cores. Watch your

RAM!!

                random_state=251108648, # Seed

                verbose=1, # If to give info during training. Set to 0 for silent training.

                warm_start=False, # If train over previously trained tree.

                ))])



# check

cv_score1 = cross_val_score(cv_model1, train_lgd.drop(columns=['LGD']),

train_lgd['LGD'],cv=5,

                                    scoring = make_scorer(mean_squared_error))



cv_score2 = cross_val_score(cv_model2, train_lgd.drop(columns=['LGD']),

train_lgd['LGD'],cv=5,
                                        scoring = make_scorer(mean_squared_error))


cv_score3 = cross_val_score(cv_model3, train_lgd.drop(columns=['LGD']),
train_lgd['LGD'],cv=5,
                                        scoring = make_scorer(mean_squared_error))


cv_score4 = cross_val_score(cv_model4, train_lgd.drop(columns=['LGD']),
train_lgd['LGD'],cv=5,
                                        scoring = make_scorer(mean_squared_error))


print(cv_score1.mean(),cv_score2.mean(),cv_score3.mean(),cv_score4.mean())


cv_model5 = Pipeline([('rf', RandomForestRegressor(n_estimators=15000, #
Number of trees to train
```

```
                    criterion='mse', # How to train the trees. Also supports entropy.
                    max_depth=None, # Max depth of the trees. Not necessary to
change.
                    min_samples_split=2, # Minimum samples to create a split.
                    min_samples_leaf=0.002, # Minimum samples in a leaf. Accepts
fractions for %. This is 0.1% of sample.
                    min_weight_fraction_leaf=0.0, # Same as above, but uses the class
weights.
                    max_features='sqrt', # Maximum number of features per split (not
tree!) by default is sqrt(vars)
                    max_leaf_nodes=None, # Maximum number of nodes.
                    min_impurity_decrease=0.0001, # Minimum impurity decrease. This
is 10^-3.
                    bootstrap=True, # If sample with repetition. For large samples
(>100.000) set to false.
                    oob_score=True,  # If report accuracy with non-selected cases.
                    n_jobs=-1, # Parallel processing. Set to -1 for all cores. Watch your
RAM!!
                    random_state=251108648, # Seed
                    verbose=1, # If to give info during training. Set to 0 for silent training.
                    warm_start=False, # If train over previously trained tree.
                    ))])

cv_score5 = cross_val_score(cv_model5, train_lgd.drop(columns=['LGD']),
train_lgd['LGD'],cv=5,
                                   scoring = make_scorer(mean_squared_error))


print(cv_score5.mean())


cv_model6 = Pipeline([('rf', RandomForestRegressor(n_estimators=15000, #
Number of trees to train
                    criterion='mse', # How to train the trees. Also supports entropy.
                    max_depth=None, # Max depth of the trees. Not necessary to
change.
                    min_samples_split=2, # Minimum samples to create a split.
                    min_samples_leaf=0.002, # Minimum samples in a leaf. Accepts
fractions for %. This is 0.1% of sample.
                    min_weight_fraction_leaf=0.0, # Same as above, but uses the class
weights.
                    max_features='log2', # Maximum number of features per split (not
tree!) by default is sqrt(vars)
                    max_leaf_nodes=None, # Maximum number of nodes.
                    min_impurity_decrease=0.0001, # Minimum impurity decrease. This
is 10^-3.
                    bootstrap=True, # If sample with repetition. For large samples
(>100.000) set to false.
                    oob_score=True,  # If report accuracy with non-selected cases.
                    n_jobs=-1, # Parallel processing. Set to -1 for all cores. Watch your
RAM!!
```

```
          random_state=251108648, # Seed

          verbose=1, # If to give info during training. Set to 0 for silent training.

          warm_start=False, # If train over previously trained tree.

          ))])



cv_model7 = Pipeline([('rf', RandomForestRegressor(n_estimators=15000, #
Number of trees to train

          criterion='mse', # How to train the trees. Also supports entropy.

          max_depth=None, # Max depth of the trees. Not necessary to
change.

          min_samples_split=2, # Minimum samples to create a split.

          min_samples_leaf=0.003, # Minimum samples in a leaf. Accepts
fractions for %. This is 0.1% of sample.

          min_weight_fraction_leaf=0.0, # Same as above, but uses the class
weights.
```

```
          max_features='sqrt', # Maximum number of features per split (not tree!) by default is sqrt(vars)
          max_leaf_nodes=None, # Maximum number of nodes.
          min_impurity_decrease=0.0001, # Minimum impurity decrease. This is 10^-3.
          bootstrap=True, # If sample with repetition. For large samples (>100.000) set to false.
          oob_score=True,  # If report accuracy with non-selected cases.
          n_jobs=-1, # Parallel processing. Set to -1 for all cores. Watch your RAM!!
          random_state=251108648, # Seed
          verbose=1, # If to give info during training. Set to 0 for silent training.
          warm_start=False, # If train over previously trained tree.
          ))])
```

```
cv_model8 = Pipeline([('rf', RandomForestRegressor(n_estimators=15000, #
Number of trees to train

                    criterion='mse', # How to train the trees. Also supports entropy.

                    max_depth=None, # Max depth of the trees. Not necessary to
change.

                    min_samples_split=2, # Minimum samples to create a split.

                    min_samples_leaf=0.003, # Minimum samples in a leaf. Accepts
fractions for %. This is 0.1% of sample.

                    min_weight_fraction_leaf=0.0, # Same as above, but uses the class
weights.

                    max_features='log2', # Maximum number of features per split (not
tree!) by default is sqrt(vars)

                    max_leaf_nodes=None, # Maximum number of nodes.

                    min_impurity_decrease=0.0001, # Minimum impurity decrease. This
is 10^-3.

                    bootstrap=True, # If sample with repetition. For large samples
(>100.000) set to false.
```

```
                    oob_score=True,  # If report accuracy with non-selected cases.
                    n_jobs=-1, # Parallel processing. Set to -1 for all cores. Watch your
RAM!!
                    random_state=251108648, # Seed
                    verbose=1, # If to give info during training. Set to 0 for silent training.
                    warm_start=False, # If train over previously trained tree.
                    ))])



cv_score6 = cross_val_score(cv_model6, train_lgd.drop(columns=['LGD']),
train_lgd['LGD'],cv=5,
                                               scoring = make_scorer(mean_squared_error))


cv_score7 = cross_val_score(cv_model7, train_lgd.drop(columns=['LGD']),
train_lgd['LGD'],cv=5,
                                               scoring = make_scorer(mean_squared_error))
```

```
cv_score8 = cross_val_score(cv_model8, train_lgd.drop(columns=['LGD']),
train_lgd['LGD'],cv=5,
                                            scoring = make_scorer(mean_squared_error))


print(cv_score6.mean(),cv_score7.mean(),cv_score8.mean())



cv_model9 = Pipeline([('rf', RandomForestRegressor(n_estimators=15000, #
Number of trees to train
                  criterion='mse', # How to train the trees. Also supports entropy.
                  max_depth=None, # Max depth of the trees. Not necessary to
change.
                  min_samples_split=2, # Minimum samples to create a split.
                  min_samples_leaf=0.004, # Minimum samples in a leaf. Accepts
fractions for %. This is 0.1% of sample.
```

```
                    min_weight_fraction_leaf=0.0, # Same as above, but uses the class
weights.
                    max_features='sqrt', # Maximum number of features per split (not
tree!) by default is sqrt(vars)
                    max_leaf_nodes=None, # Maximum number of nodes.
                    min_impurity_decrease=0.0001, # Minimum impurity decrease. This
is 10^-3.
                    bootstrap=True, # If sample with repetition. For large samples
(>100.000) set to false.
                    oob_score=True,  # If report accuracy with non-selected cases.
                    n_jobs=-1, # Parallel processing. Set to -1 for all cores. Watch your
RAM!!
                    random_state=251108648, # Seed
                    verbose=1, # If to give info during training. Set to 0 for silent training.
                    warm_start=False, # If train over previously trained tree.
                    ))])
```

```
cv_model10 = Pipeline([('rf', RandomForestRegressor(n_estimators=15000, #
Number of trees to train

                    criterion='mse', # How to train the trees. Also supports entropy.

                    max_depth=None, # Max depth of the trees. Not necessary to
change.

                    min_samples_split=2, # Minimum samples to create a split.

                    min_samples_leaf=0.005, # Minimum samples in a leaf. Accepts
fractions for %. This is 0.1% of sample.

                    min_weight_fraction_leaf=0.0, # Same as above, but uses the class
weights.

                    max_features='sqrt', # Maximum number of features per split (not
tree!) by default is sqrt(vars)

                    max_leaf_nodes=None, # Maximum number of nodes.

                    min_impurity_decrease=0.0001, # Minimum impurity decrease. This
is 10^-3.
```

```
                  bootstrap=True, # If sample with repetition. For large samples
(>100.000) set to false.
                  oob_score=True,  # If report accuracy with non-selected cases.
                  n_jobs=-1, # Parallel processing. Set to -1 for all cores. Watch your
RAM!!
                  random_state=251108648, # Seed
                  verbose=1, # If to give info during training. Set to 0 for silent training.
                  warm_start=False, # If train over previously trained tree.
                  ))])


cv_model11 = Pipeline([('rf', RandomForestRegressor(n_estimators=15000, #
Number of trees to train
                  criterion='mse', # How to train the trees. Also supports entropy.
                  max_depth=None, # Max depth of the trees. Not necessary to
change.
                  min_samples_split=2, # Minimum samples to create a split.
```

```
                    min_samples_leaf=0.006, # Minimum samples in a leaf. Accepts
fractions for %. This is 0.1% of sample.
                    min_weight_fraction_leaf=0.0, # Same as above, but uses the class
weights.
                    max_features='sqrt', # Maximum number of features per split (not
tree!) by default is sqrt(vars)
                    max_leaf_nodes=None, # Maximum number of nodes.
                    min_impurity_decrease=0.0001, # Minimum impurity decrease. This
is 10^-3.
                    bootstrap=True, # If sample with repetition. For large samples
(>100.000) set to false.
                    oob_score=True,  # If report accuracy with non-selected cases.
                    n_jobs=-1, # Parallel processing. Set to -1 for all cores. Watch your
RAM!!
                    random_state=251108648, # Seed
                    verbose=1, # If to give info during training. Set to 0 for silent training.
                    warm_start=False, # If train over previously trained tree.
                    ))])


cv_score9 = cross_val_score(cv_model9, train_lgd.drop(columns=['LGD']),

train_lgd['LGD'],cv=5,

                                               scoring = make_scorer(mean_squared_error))



cv_score10 = cross_val_score(cv_model10, train_lgd.drop(columns=['LGD']),

train_lgd['LGD'],cv=5,

                                               scoring = make_scorer(mean_squared_error))



cv_score11 = cross_val_score(cv_model11, train_lgd.drop(columns=['LGD']),

train_lgd['LGD'],cv=5,

                                               scoring = make_scorer(mean_squared_error))



print(cv_score9.mean(),cv_score10.mean(),cv_score11.mean())
```

```
# cv_score7 get best performance


# Apply the model on the full dataset

cv_score = cross_val_score(cv_model7, train_lgd.drop(columns=['LGD']),

train_lgd['LGD'],cv=5,

                                   scoring = make_scorer(mean_squared_error))

print(cv_score.mean())



rf_importance =

cv_model7[0].fit(train_lgd.drop(columns=['LGD']),train_lgd['LGD']).feature_impor

tances_
```

```
# xgboost
LC = LC_part.iloc[:50000,:]



cv_model_xg1 = Pipeline([('xgboost', GradientBoostingRegressor(loss='ls', # How
to calculate losses. Deviance is for probabilistic outputs. Alternative exponential
for AdaBoost.
                    learning_rate=0.1, # How much to shrink error in each subsequent
training. Trade-off with no. estimators.
                    n_estimators=10000, # How many trees to use, the more the
better, but decrease learning rate if many used.
                    subsample=0.632, # Subsampling to use. 63.2% of data is standard
for XGBoost.
                    criterion='friedman_mse', # Error to use for each split. Good idea
to leave it as is.
                    min_samples_split=2, # Minimum samples for a split.
                    min_samples_leaf=1, # Minimum samples in a leaf.
```

```
                min_weight_fraction_leaf=0.0, # Minimum fraction of samples in a
leaf. Consider increasing if first few trees too good.

                max_depth=3, # Maximum depth. Keep it small!

                min_impurity_decrease=0.01, # Minimum impurity decrease.
Might want to use 1% or so.

                init=None, # How to make first prediction (it needs one). Can give
model that supports fit and predict.

                random_state=251108648, # Seed

                max_features='auto', # Same as RF.

                verbose=1,  # Same as RF.

                max_leaf_nodes=None,  # Same as RF.

                warm_start=False,  # Same as RF.

                presort='auto', # Whether to presort the data to speed up training.

                validation_fraction=0.3, # XGBoost CAN overfit, so control this just
in case. Uses 30% validation in this case.
```

```
                    n_iter_no_change=None, # Iters to stop training if no change
occurs between one tree and the next.
                    tol=0.0001 # Tolerance. Means maximum change of 10^-4
                   ))])



cv_score_xg1 = cross_val_score(cv_model_xg1, train_lgd.drop(columns=['LGD']),
train_lgd['LGD'],cv=5,
                                        scoring = make_scorer(mean_squared_error))


print(cv_score_xg1.mean())


cv_model_xg2 = Pipeline([('xgboost', GradientBoostingRegressor(loss='ls', # How
to calculate losses. Deviance is for probabilistic outputs. Alternative exponential
for AdaBoost.
```

```
                    learning_rate=0.1, # How much to shrink error in each subsequent
training. Trade-off with no. estimators.
                    n_estimators=15000, # How many trees to use, the more the
better, but decrease learning rate if many used.
                    subsample=0.632, # Subsampling to use. 63.2% of data is standard
for XGBoost.
                    criterion='friedman_mse', # Error to use for each split. Good idea
to leave it as is.
                    min_samples_split=2, # Minimum samples for a split.
                    min_samples_leaf=1, # Minimum samples in a leaf.
                    min_weight_fraction_leaf=0.0, # Minimum fraction of samples in a
leaf. Consider increasing if first few trees too good.
                    max_depth=3, # Maximum depth. Keep it small!
                    min_impurity_decrease=0.01, # Minimum impurity decrease.
Might want to use 1% or so.
                    init=None, # How to make first prediction (it needs one). Can give
model that supports fit and predict.
```

```
                    random_state=251108648, # Seed

                    max_features='auto', # Same as RF.

                    verbose=1,  # Same as RF.

                    max_leaf_nodes=None,  # Same as RF.

                    warm_start=False,  # Same as RF.

                    presort='auto', # Whether to presort the data to speed up training.

                    validation_fraction=0.3, # XGBoost CAN overfit, so control this just
in case. Uses 30% validation in this case.

                    n_iter_no_change=None, # Iters to stop training if no change
occurs between one tree and the next.

                    tol=0.0001 # Tolerance. Means maximum change of 10^-4

                   ))])



cv_model_xg3 = Pipeline([('xgboost', GradientBoostingRegressor(loss='ls', # How
to calculate losses. Deviance is for probabilistic outputs. Alternative exponential
for AdaBoost.
```

learning_rate=0.1, # How much to shrink error in each subsequent training. Trade-off with no. estimators.

n_estimators=10000, # How many trees to use, the more the better, but decrease learning rate if many used.

subsample=0.632, # Subsampling to use. 63.2% of data is standard for XGBoost.

criterion='friedman_mse', # Error to use for each split. Good idea to leave it as is.

min_samples_split=2, # Minimum samples for a split.

min_samples_leaf=50, # Minimum samples in a leaf.

min_weight_fraction_leaf=0.0, # Minimum fraction of samples in a leaf. Consider increasing if first few trees too good.

max_depth=3, # Maximum depth. Keep it small!

min_impurity_decrease=0.01, # Minimum impurity decrease. Might want to use 1% or so.

init=None, # How to make first prediction (it needs one). Can give model that supports fit and predict.

```
                    random_state=251108648, # Seed

                    max_features='auto', # Same as RF.

                    verbose=1,  # Same as RF.

                    max_leaf_nodes=None,  # Same as RF.

                    warm_start=False,  # Same as RF.

                    presort='auto', # Whether to presort the data to speed up training.

                    validation_fraction=0.3, # XGBoost CAN overfit, so control this just

in case. Uses 30% validation in this case.

                    n_iter_no_change=None, # Iters to stop training if no change

occurs between one tree and the next.

                    tol=0.0001 # Tolerance. Means maximum change of 10^-4

                   ))])
```

```
cv_model_xg4 = Pipeline([('xgboost', GradientBoostingRegressor(loss='ls', # How
to calculate losses. Deviance is for probabilistic outputs. Alternative exponential
for AdaBoost.
                learning_rate=0.05, # How much to shrink error in each
subsequent training. Trade-off with no. estimators.
                n_estimators=10000, # How many trees to use, the more the
better, but decrease learning rate if many used.
                subsample=0.632, # Subsampling to use. 63.2% of data is standard
for XGBoost.
                criterion='friedman_mse', # Error to use for each split. Good idea
to leave it as is.
                min_samples_split=2, # Minimum samples for a split.
                min_samples_leaf=1, # Minimum samples in a leaf.
                min_weight_fraction_leaf=0.0, # Minimum fraction of samples in a
leaf. Consider increasing if first few trees too good.
                max_depth=3, # Maximum depth. Keep it small!
```

```
                    min_impurity_decrease=0.01, # Minimum impurity decrease.
Might want to use 1% or so.
                    init=None, # How to make first prediction (it needs one). Can give
model that supports fit and predict.
                    random_state=251108648, # Seed
                    max_features='auto', # Same as RF.
                    verbose=1,  # Same as RF.
                    max_leaf_nodes=None,  # Same as RF.
                    warm_start=False,  # Same as RF.
                    presort='auto', # Whether to presort the data to speed up training.
                    validation_fraction=0.3, # XGBoost CAN overfit, so control this just
in case. Uses 30% validation in this case.
                    n_iter_no_change=None, # Iters to stop training if no change
occurs between one tree and the next.
                    tol=0.0001 # Tolerance. Means maximum change of 10^-4
                   ))])
```

```
cv_score_xg2 = cross_val_score(cv_model_xg2, train_lgd.drop(columns=['LGD']),
train_lgd['LGD'],cv=5,
                                        scoring = make_scorer(mean_squared_error))

cv_score_xg3 = cross_val_score(cv_model_xg3, train_lgd.drop(columns=['LGD']),
train_lgd['LGD'],cv=5,
                                        scoring = make_scorer(mean_squared_error))

cv_score_xg4 = cross_val_score(cv_model_xg4, train_lgd.drop(columns=['LGD']),
train_lgd['LGD'],cv=5,
                                        scoring = make_scorer(mean_squared_error))
```

```
print(cv_score_xg1.mean(),cv_score_xg2.mean(),cv_score_xg3.mean(),cv_score_x
g4.mean())



cv_model_xg5 = Pipeline([('xgboost', GradientBoostingRegressor(loss='ls', # How
to calculate losses. Deviance is for probabilistic outputs. Alternative exponential
for AdaBoost.
                    learning_rate=0.1, # How much to shrink error in each subsequent
training. Trade-off with no. estimators.
                    n_estimators=10000, # How many trees to use, the more the
better, but decrease learning rate if many used.
                    subsample=0.632, # Subsampling to use. 63.2% of data is standard
for XGBoost.
                    criterion='friedman_mse', # Error to use for each split. Good idea
to leave it as is.
                    min_samples_split=2, # Minimum samples for a split.
                    min_samples_leaf=60, # Minimum samples in a leaf.
                    min_weight_fraction_leaf=0.0, # Minimum fraction of samples in a
leaf. Consider increasing if first few trees too good.
                    max_depth=3, # Maximum depth. Keep it small!
                    min_impurity_decrease=0.01, # Minimum impurity decrease.
Might want to use 1% or so.
                    init=None, # How to make first prediction (it needs one). Can give
model that supports fit and predict.
                    random_state=251108648, # Seed
                    max_features='auto', # Same as RF.
                    verbose=1,  # Same as RF.
                    max_leaf_nodes=None,  # Same as RF.
                    warm_start=False,  # Same as RF.
                    presort='auto', # Whether to presort the data to speed up training.
                    validation_fraction=0.3, # XGBoost CAN overfit, so control this just
in case. Uses 30% validation in this case.
```

```
                        n_iter_no_change=None, # Iters to stop training if no change
occurs between one tree and the next.
                        tol=0.0001 # Tolerance. Means maximum change of 10^-4
                       ))])


cv_model_xg6 = Pipeline([('xgboost', GradientBoostingRegressor(loss='ls', # How
to calculate losses. Deviance is for probabilistic outputs. Alternative exponential
for AdaBoost.
                        learning_rate=0.2, # How much to shrink error in each subsequent
training. Trade-off with no. estimators.
                        n_estimators=10000, # How many trees to use, the more the
better, but decrease learning rate if many used.
                        subsample=0.632, # Subsampling to use. 63.2% of data is standard
for XGBoost.
                        criterion='friedman_mse', # Error to use for each split. Good idea
to leave it as is.
                        min_samples_split=2, # Minimum samples for a split.
```

```
                    min_samples_leaf=50, # Minimum samples in a leaf.
                    min_weight_fraction_leaf=0.0, # Minimum fraction of samples in a leaf. Consider increasing if first few trees too good.
                    max_depth=3, # Maximum depth. Keep it small!
                    min_impurity_decrease=0.01, # Minimum impurity decrease. Might want to use 1% or so.
                    init=None, # How to make first prediction (it needs one). Can give model that supports fit and predict.
                    random_state=251108648, # Seed
                    max_features='auto', # Same as RF.
                    verbose=1,  # Same as RF.
                    max_leaf_nodes=None,  # Same as RF.
                    warm_start=False,  # Same as RF.
                    presort='auto', # Whether to presort the data to speed up training.
                    validation_fraction=0.3, # XGBoost CAN overfit, so control this just in case. Uses 30% validation in this case.
```

n_iter_no_change=None, # Iters to stop training if no change occurs between one tree and the next.

tol=0.0001 # Tolerance. Means maximum change of 10^-4

))])

cv_model_xg7 = Pipeline([('xgboost', GradientBoostingRegressor(loss='ls', # How to calculate losses. Deviance is for probabilistic outputs. Alternative exponential for AdaBoost.

learning_rate=0.2, # How much to shrink error in each subsequent training. Trade-off with no. estimators.

n_estimators=10000, # How many trees to use, the more the better, but decrease learning rate if many used.

subsample=0.632, # Subsampling to use. 63.2% of data is standard for XGBoost.

criterion='friedman_mse', # Error to use for each split. Good idea to leave it as is.

min_samples_split=2, # Minimum samples for a split.

```
            min_samples_leaf=60, # Minimum samples in a leaf.
            min_weight_fraction_leaf=0.0, # Minimum fraction of samples in a leaf. Consider increasing if first few trees too good.
            max_depth=3, # Maximum depth. Keep it small!
            min_impurity_decrease=0.01, # Minimum impurity decrease. Might want to use 1% or so.
            init=None, # How to make first prediction (it needs one). Can give model that supports fit and predict.
            random_state=251108648, # Seed
            max_features='auto', # Same as RF.
            verbose=1,  # Same as RF.
            max_leaf_nodes=None,  # Same as RF.
            warm_start=False,  # Same as RF.
            presort='auto', # Whether to presort the data to speed up training.
            validation_fraction=0.3, # XGBoost CAN overfit, so control this just in case. Uses 30% validation in this case.
```

```
                    n_iter_no_change=None, # Iters to stop training if no change
occurs between one tree and the next.
                    tol=0.0001 # Tolerance. Means maximum change of 10^-4
                    ))])



cv_score_xg5 = cross_val_score(cv_model_xg5, train_lgd.drop(columns=['LGD']),
train_lgd['LGD'],cv=5,
                                        scoring = make_scorer(mean_squared_error))



cv_score_xg6 = cross_val_score(cv_model_xg6, train_lgd.drop(columns=['LGD']),
train_lgd['LGD'],cv=5,
                                        scoring = make_scorer(mean_squared_error))
```

```
cv_score_xg7 = cross_val_score(cv_model_xg7, train_lgd.drop(columns=['LGD']),
train_lgd['LGD'],cv=5,
                                        scoring = make_scorer(mean_squared_error))



print(cv_score_xg5.mean(),cv_score_xg6.mean(),cv_score_xg7.mean())



cv_model_xg8 = Pipeline([('xgboost', GradientBoostingRegressor(loss='ls', # How
to calculate losses. Deviance is for probabilistic outputs. Alternative exponential
for AdaBoost.
                    learning_rate=0.1, # How much to shrink error in each subsequent
training. Trade-off with no. estimators.
                    n_estimators=10000, # How many trees to use, the more the
better, but decrease learning rate if many used.
                    subsample=0.632, # Subsampling to use. 63.2% of data is standard
for XGBoost.
                    criterion='friedman_mse', # Error to use for each split. Good idea
to leave it as is.
```

```
            min_samples_split=2, # Minimum samples for a split.
            min_samples_leaf=40, # Minimum samples in a leaf.
            min_weight_fraction_leaf=0.0, # Minimum fraction of samples in a
leaf. Consider increasing if first few trees too good.
            max_depth=3, # Maximum depth. Keep it small!
            min_impurity_decrease=0.01, # Minimum impurity decrease.
Might want to use 1% or so.
            init=None, # How to make first prediction (it needs one). Can give
model that supports fit and predict.
            random_state=251108648, # Seed
            max_features='auto', # Same as RF.
            verbose=1,  # Same as RF.
            max_leaf_nodes=None,  # Same as RF.
            warm_start=False,  # Same as RF.
            presort='auto', # Whether to presort the data to speed up training.
```

```
                    validation_fraction=0.3, # XGBoost CAN overfit, so control this just
in case. Uses 30% validation in this case.

                    n_iter_no_change=None, # Iters to stop training if no change
occurs between one tree and the next.

                    tol=0.0001 # Tolerance. Means maximum change of 10^-4

                    ))])



cv_model_xg9 = Pipeline([('xgboost', GradientBoostingRegressor(loss='ls', # How
to calculate losses. Deviance is for probabilistic outputs. Alternative exponential
for AdaBoost.

                    learning_rate=0.1, # How much to shrink error in each subsequent
training. Trade-off with no. estimators.

                    n_estimators=10000, # How many trees to use, the more the
better, but decrease learning rate if many used.

                    subsample=0.632, # Subsampling to use. 63.2% of data is standard
for XGBoost.
```

```
                    criterion='friedman_mse', # Error to use for each split. Good idea
to leave it as is.
                    min_samples_split=2, # Minimum samples for a split.
                    min_samples_leaf=30, # Minimum samples in a leaf.
                    min_weight_fraction_leaf=0.0, # Minimum fraction of samples in a
leaf. Consider increasing if first few trees too good.
                    max_depth=3, # Maximum depth. Keep it small!
                    min_impurity_decrease=0.01, # Minimum impurity decrease.
Might want to use 1% or so.
                    init=None, # How to make first prediction (it needs one). Can give
model that supports fit and predict.
                    random_state=251108648, # Seed
                    max_features='auto', # Same as RF.
                    verbose=1,  # Same as RF.
                    max_leaf_nodes=None,  # Same as RF.
                    warm_start=False,  # Same as RF.
                    presort='auto', # Whether to presort the data to speed up training.
                    validation_fraction=0.3, # XGBoost CAN overfit, so control this just
in case. Uses 30% validation in this case.
                    n_iter_no_change=None, # Iters to stop training if no change
occurs between one tree and the next.
                    tol=0.0001 # Tolerance. Means maximum change of 10^-4
                   ))])


cv_model_xg10 = Pipeline([('xgboost', GradientBoostingRegressor(loss='ls', # How
to calculate losses. Deviance is for probabilistic outputs. Alternative exponential
for AdaBoost.
                    learning_rate=0.1, # How much to shrink error in each subsequent
training. Trade-off with no. estimators.
                    n_estimators=10000, # How many trees to use, the more the
better, but decrease learning rate if many used.
                    subsample=0.632, # Subsampling to use. 63.2% of data is standard
for XGBoost.
```

```
                    criterion='friedman_mse', # Error to use for each split. Good idea
to leave it as is.
                    min_samples_split=2, # Minimum samples for a split.
                    min_samples_leaf=20, # Minimum samples in a leaf.
                    min_weight_fraction_leaf=0.0, # Minimum fraction of samples in a
leaf. Consider increasing if first few trees too good.
                    max_depth=3, # Maximum depth. Keep it small!
                    min_impurity_decrease=0.01, # Minimum impurity decrease.
Might want to use 1% or so.
                    init=None, # How to make first prediction (it needs one). Can give
model that supports fit and predict.
                    random_state=251108648, # Seed
                    max_features='auto', # Same as RF.
                    verbose=1,  # Same as RF.
                    max_leaf_nodes=None,  # Same as RF.
                    warm_start=False,  # Same as RF.
                    presort='auto', # Whether to presort the data to speed up training.
                    validation_fraction=0.3, # XGBoost CAN overfit, so control this just
in case. Uses 30% validation in this case.
                    n_iter_no_change=None, # Iters to stop training if no change
occurs between one tree and the next.
                    tol=0.0001 # Tolerance. Means maximum change of 10^-4
                    ))])


cv_model_xg11 = Pipeline([('xgboost', GradientBoostingRegressor(loss='ls', # How
to calculate losses. Deviance is for probabilistic outputs. Alternative exponential
for AdaBoost.
                    learning_rate=0.07, # How much to shrink error in each
subsequent training. Trade-off with no. estimators.
                    n_estimators=15000, # How many trees to use, the more the
better, but decrease learning rate if many used.
                    subsample=0.632, # Subsampling to use. 63.2% of data is standard
for XGBoost.
```

```
                    criterion='friedman_mse', # Error to use for each split. Good idea to leave it as is.
                    min_samples_split=2, # Minimum samples for a split.
                    min_samples_leaf=50, # Minimum samples in a leaf.
                    min_weight_fraction_leaf=0.0, # Minimum fraction of samples in a leaf. Consider increasing if first few trees too good.
                    max_depth=3, # Maximum depth. Keep it small!
                    min_impurity_decrease=0.01, # Minimum impurity decrease. Might want to use 1% or so.
                    init=None, # How to make first prediction (it needs one). Can give model that supports fit and predict.
                    random_state=251108648, # Seed
                    max_features='auto', # Same as RF.
                    verbose=1,  # Same as RF.
                    max_leaf_nodes=None,  # Same as RF.
                    warm_start=False,  # Same as RF.
```

```
                        presort='auto', # Whether to presort the data to speed up training.

                        validation_fraction=0.3, # XGBoost CAN overfit, so control this just
in case. Uses 30% validation in this case.

                        n_iter_no_change=None, # Iters to stop training if no change
occurs between one tree and the next.

                        tol=0.0001 # Tolerance. Means maximum change of 10^-4

                       ))])



cv_score_xg8 = cross_val_score(cv_model_xg8, train_lgd.drop(columns=['LGD']),
train_lgd['LGD'],cv=5,

                                        scoring = make_scorer(mean_squared_error))



cv_score_xg9 = cross_val_score(cv_model_xg9, train_lgd.drop(columns=['LGD']),
train_lgd['LGD'],cv=5,
                                          scoring = make_scorer(mean_squared_error))



cv_score_xg10 = cross_val_score(cv_model_xg10,

train_lgd.drop(columns=['LGD']), train_lgd['LGD'],cv=5,

                                          scoring = make_scorer(mean_squared_error))


cv_score_xg11 = cross_val_score(cv_model_xg11,

train_lgd.drop(columns=['LGD']), train_lgd['LGD'],cv=5,

                                          scoring = make_scorer(mean_squared_error))

print(cv_score_xg8.mean(),cv_score_xg9.mean(),cv_score_xg10.mean(),cv_score

_xg11.mean())
```

```
cv_model_xg12 = Pipeline([('xgboost', GradientBoostingRegressor(loss='ls', # How
to calculate losses. Deviance is for probabilistic outputs. Alternative exponential
for AdaBoost.
                    learning_rate=0.07, # How much to shrink error in each
subsequent training. Trade-off with no. estimators.
                    n_estimators=15000, # How many trees to use, the more the
better, but decrease learning rate if many used.
                    subsample=0.632, # Subsampling to use. 63.2% of data is standard
for XGBoost.
                    criterion='friedman_mse', # Error to use for each split. Good idea
to leave it as is.
                    min_samples_split=2, # Minimum samples for a split.
                    min_samples_leaf=20, # Minimum samples in a leaf.
                    min_weight_fraction_leaf=0.0, # Minimum fraction of samples in a
leaf. Consider increasing if first few trees too good.
                    max_depth=3, # Maximum depth. Keep it small!
```

```
            min_impurity_decrease=0.01, # Minimum impurity decrease.
Might want to use 1% or so.
            init=None, # How to make first prediction (it needs one). Can give
model that supports fit and predict.
            random_state=251108648, # Seed
            max_features='auto', # Same as RF.
            verbose=1,  # Same as RF.
            max_leaf_nodes=None,  # Same as RF.
            warm_start=False,  # Same as RF.
            presort='auto', # Whether to presort the data to speed up training.
            validation_fraction=0.3, # XGBoost CAN overfit, so control this just
in case. Uses 30% validation in this case.
            n_iter_no_change=None, # Iters to stop training if no change
occurs between one tree and the next.
            tol=0.0001 # Tolerance. Means maximum change of 10^-4
           ))])
```

```
cv_score_xg12 = cross_val_score(cv_model_xg12,

train_lgd.drop(columns=['LGD']), train_lgd['LGD'],cv=5,

                                             scoring = make_scorer(mean_squared_error))

print(cv_score_xg12.mean())



xg_importance =

cv_model_xg12[0].fit(train_lgd.drop(columns=['LGD']),train_lgd['LGD']).feature_i

mportances_



# Q2

# weight of evidence for pd

train_pd, test_pd = sc.split_df(LC_pd, y='loan_status', ratio=0.7, seed =

251108648).values()
```

```
bins = sc.woebin(train_pd, y='loan_status',

              min_perc_fine_bin=0.05,

              min_perc_coarse_bin=0.05,

              stop_limit=0.1,

              max_num_bin=8,

              method='tree')
```

```
sc.woebin_plot(bins)
```

```
# We have fico_range_high, dti, last_pymnt_amnt (containing future infomation),

#annual_inc, earliest_cr_line,
# total_pymnt, fico_range_low

# we need mannually adjust annual_inc



break_adj = sc.woebin_adj(train_pd, 'loan_status', bins)






# apply new cutts

bins_adj = sc.woebin(train_pd, y='loan_status', breaks_list=break_adj) #Apply
new cuts

train_woe = sc.woebin_ply(train_pd,bins_adj) #calculate WOE dataset (train)

test_woe = sc.woebin_ply(test_pd, bins_adj) #calculate WOE dataset (test)




# list iv

sc.iv(train_woe, 'loan_status')
```

```
# build Lasso regression (training)

pd_logreg = LogisticRegression(penalty='l1',

                    tol=0.001,

                    C=0.7,

                    fit_intercept=True,

                    class_weight='balanced',

                    random_state=251108648,

                    max_iter=300,

                    verbose=1,

                    solver='saga',

                    warm_start=False)



# create range of accepted variables

train_woe = train_woe.loc[:, ['loan_status','term_woe',

                    'fico_range_high_woe',
                        'int_rate_woe', 'revol_util_woe',

                        'bc_util_woe']]

test_woe = test_woe.loc[:, ['loan_status','term_woe',

                        'fico_range_high_woe',

                        'int_rate_woe','revol_util_woe',

                        'bc_util_woe']]




#test_woe = test_woe.fillna(np.nanmedian(test_woe["earliest_cr_line_woe"]))

#train_woe.head()


# training for pd

train_woe['loan_status'] = train_woe['loan_status'].astype('int')

test_woe['loan_status'] = test_woe['loan_status'].astype('int')

pd_logreg.fit(X=train_woe.iloc[:,1:], y=train_woe['loan_status'].astype('int'))
```

```
# prediction

pred_pd_test = pd_logreg.predict(test_woe.iloc[:, 1:])

probs_test = pd_logreg.predict_proba(test_woe.iloc[:, 1:])

print(probs_test[0:5], pred_pd_test[0:5])




# confusion matrix



confusion_matrix(y_true=test_woe['loan_status'], y_pred=pred_pd_test)



# Calculate the ROC curve points

fpr, tpr, threshold = roc_curve(test_pd['loan_status'].astype('int'),probs_test[:,1])



plt.plot(fpr,tpr,label="pd"+str(auc))
plt.legend(loc=4)

plt.show()



#save the AUC in a variable to display it, round it first

auc = np.round(roc_auc_score(y_true = test_pd['loan_status'].astype('int'),

                          y_score=probs_test[:,1]), decimals=3)




#array([[331, 187],

#       [ 41,  69]], dtype=int64)



# scorecards

pd_sc = sc.scorecard(bins_adj, pd_logreg,

                  train_woe.columns[1:],

                  points0=600,
                    odds0=0.01,

                    pdo=50)



# applying the credit score. Applies over the original data!

train_score = sc.scorecard_ply(train_pd, pd_sc, print_step=0)



test_score = sc.scorecard_ply(test_pd, pd_sc, print_step=0)
```