

Analyzing default-ability of Home Loan Lenders using Machine Learning Models

SAKIB SADMAN SHAJIB

1731201042

North South University
sakib.shajib173@northsouth.edu

TAHMID HASIN NAFI

1321213042

North South University
tahmid.nafi@northsouth.edu

ILMIAT FARHANA

1712428042

North South University
ilmia.farhana@northsouth.edu

IFTEKHAR ALAM PIAL

1410213642

North South University
iftekharpial@northsouth.edu

April 27, 2021

Abstract

A more efficient way of reviewing loan applications which might reduce the number of loans defaulted and might make loan processing faster. The kaggle kernel we reviewed talks about using machine learning to analyze the patterns in the data collected in a loan application and tries to predict if the applicant will payback the loan in due time. This report will go through the pipelines of how the data was pre-processed, feature engineering, algorithms used, data validation process of a few kaggle kernels on our selected Dataset. The dataset was provided by Home Credit Group for a Kaggle Competition.

I. INTRODUCTION

i. Kernel List

This report will review the below mentioned Kaggle kernels:

1. LightGBM 7th place solution
2. LightGBM with Simple Features
3. Home Credit Default Risk: Visualization & Analysis
4. Start Here: A Gentle Introduction
5. Advanced Auto-Feature Engineering | Credit Default
6. Home Credit Default Report 2

ii. Points to discuss

Points we are looking into to study the kernels¹:

- Pre-processing
- Feature Engineering

¹ All points might not be accurately represented all the teammates reviews

- Algorithms Used and How
- Validation
- Accuracy

There might be a few good performing kernels which might have multiple other important points, we'll discuss them under a custom point as well.

II. KERNEL REVIEW

i. LightGBM 7th place solution

This kernel was one of the best performing ones submitted by kaggle users Aguiar and Abdelwahed Assklou. It scored 0.80028 AUC score. Let's discuss this kernel.

In short[1]:

- They feature engineered the dataset to have around 1200 features.
- They used 11 different LightGBM[2] models with multiple different set of features.
- They also tried out two different Neural Networks, they didn't perform better.
- But they got the best score by stacking 13 models of a LinearRegression[3] algorithm.
- They used StratifiedKFold[4] for Cross Validation

i.1 Pre-Processing

At the beginning they started out with pre-processing.

Application They removed redundant data like people with invalid gender, invalid income, invalid days employed, invalid days from last phone number change. create a categorical age feature.

Used label encoding for categorical data and dropped a few column based on permutation feature importance.

Bureau The categorical data is processed with one-hot encoder. and the data was merged with Bureau_balance.csv.

Previous Application Used One Hot Encoding

i.2 Feature Engineering

Application They created new features from documents using count and kurtosis[6]. And a few ratio based features, and groupbys.

Features created are:

- Credit ratios
- Income ratios
- Time ratios

Bureau, Previous Application, POS_CASH, Installments, Credit Card Created new features based on domain knowledge.

- Bureau
 - Credit duration and credit/account and date difference
 - Credit to debt ration and difference
 - Flag months with late payments (days past due)

- Aggregate by number of months in balance and merge with bureau (loan length agg)
 - General loans aggregations
 - Active and closed loans aggregations
 - Aggregations for the main loan types
 - Time based aggregations: last x months
 - Last loan max overdue
 - Ratios: total debt/total credit and active loans debt/ active loans credit
 - Calculate rate for each category with decay
 - Min, Max, Count and mean duration of payments (months)
- Previous Application
 - ratios and difference
 - Interest ratio on previous application (simplified)
 - Active loans - approved and not complete yet (last_due 365243)
 - Find how much was already payed in active loans (using installments csv)
 - Active loans: difference of what was payed and installments
 - Merge with active_df
 - Perform aggregations for active applications
 - Change 365.243 values to nan (missing)
 - Days last due difference (scheduled x done)
 - Categorical features Aggregations
 - Perform general aggregations
 - Merge active loans dataframe on agg_prev
 - Aggregations for approved and refused loans
 - Aggregations for Consumer loans and Cash loans
 - Get the SK_ID_PREV for loans with late payments (days past due)
 - Aggregations for loans with late payments
 - Aggregations for loans in the last x months
- POS Cash
 - Flag months with late payment
 - Aggregate by SK_ID_CURR
 - Sort and group by SK_ID_PREV
 - Percentage of previous loans completed and completed before initial term
 - Number of remaining installments (future installments) and percentage from total
 - Number of remaining installments (future installments) and percentage from total
 - Percentage of late payments for the 3 most recent applications
 - Last month of each application
 - Most recent applications (last 3)
 - Drop some useless categorical features
- Installment Payments
 - Group payments and get Payment difference
 - Payment Entry: Days past due and Days before due
 - Flag late payment
 - Percentage of payments that were late
 - Flag late payments that have a significant amount
 - Flag k threshold late payments
 - Aggregations by SK_ID_CURR
 - Installments in the last x months
 - Last x periods trend features
 - Last loan features
- Credit Card

- Amount used from limit
- Current payment / Min payment
- Late payment
- How much drawing of limit
- Aggregations by SK_ID_CURR
- Last month balance of each credit card application
- Aggregations for last x months

i.3 Algorithms used and How

Used LinearRegression for Feature Engineering in Installment Payment.

LightGBM for binary classification, with parameters such as goss[5] boosting type, ad other parameters.

```
LIGHTGBM_PARAMS = {  
    'boosting_type': 'goss',  
    'n_estimators': 10000,  
    'learning_rate': 0.005134,  
    'num_leaves': 54,  
    'max_depth': 10,  
    'subsample_for_bin': 240000,  
    'reg_alpha': 0.436193,  
    'reg_lambda': 0.479169,  
    'colsample_bytree': 0.508716,  
    'min_split_gain': 0.024766,  
    'subsample': 1,  
    'is_unbalance': False,  
    'silent': -1,  
    'verbose': -1  
}
```

i.4 Validation

Used KFold or Stratified KFold to Cross Validate the data. KFold is recommended.

i.5 Accuracy

Metrics used to calculate accuracy is ROC AUC. The accuracy of this kernel is 0.80028 in LeaderBoard, but in private Cross Validation it was 0.79851.

ii. LightGBM with Simple Features

This kernel was one of the best performing ones submitted by kaggle users MEHMET KELEŞ. Let's discuss this kernel.

Key Ideas:

- Divide or subtract important features to get rates (like annuity and income).
- In Bureau Data: create specific features for Active credits and Closed credits.
- In Previous Applications: create specific features for Approved and Refused applications.
- Modularity: one function for each table (except bureau_balance and application_test).
- One-hot encoding for categorical features.
- All tables are joined with the application DF using the SK_ID_CURR key (except bureau_balance).
- You can use LightGBM with KFold or Stratified KFold. KFold gives better results.

ii.1 Pre-Processing

Removed Gender with invalid input, and binary encoded gender, ownership of car, ownership of realestate. And one-hot encoding for rest of the categorical data. and removed garbage data.

ii.2 Feature Engineering

Created new features using domain knowledge such as payment rate, annuity income percentage and other percentage based features, Time ratios etc.

In Bureau balance, Perform aggregations and merge with bureau.csv. Bureau and bureau_balance numeric and categorical features are added. Active and closed Credit Scores are computed using the numerical aggregations

In Previous Application, Computed loan received percentage with amount application by amount credit. Also create new numeric and categorical features. Find Approved and rejected application from numerical feature.

In POS and Credit Card data, the data is grouped using SK_ID_CURR/. In case of Credit Cards, they created a new feature with the amount of limits used.

In Installments Payments data, they ran an aggregator to create new numerical feature and grouped the data by loan applicants.

ii.3 Algorithms used and How

LightGBM for binary classification, with a few parameters:

```
# LightGBM parameters found by Bayesian optimization
clf = LGBMClassifier(
    nthread=4,
    n_estimators=10000,
    learning_rate=0.02,
    num_leaves=34,
    colsample_bytree=0.9497036,
    subsample=0.8715623,
    max_depth=8,
    reg_alpha=0.041545473,
    reg_lambda=0.0735294,
    min_split_gain=0.0222415,
    min_child_weight=39.3259775,
    silent=-1,
    verbose=-1, )
```

ii.4 Validation

Used KFold or Stratified KFold to Cross Validate the data.

ii.5 Accuracy

Metrics used to calculate accuracy is ROC AUC.

iii. Home Credit Default Risk: Visualization & Analysis

iii.1 Introduction

Objective: Predict whether a loan applicant is capable of repaying the intended borrowed amount. We'll be predicting a category for an applicant

iii.2 Data Processing

There are 10 files in our library. 7 of them are data sources, and the remaining 3 are the train, test, and sample submission files. For the training file, we have a total of 307511 observations and 122 features to consider with integer, float and object datatypes. Also there are 67 features with null values..

iii.3 Observation

The test file is almost similar as the training file.having 48744 observations, 121 features (minus the predictor variable 'TARGET'), and 64 features having null values. The words being used as; Observations = rows, features= columns Based on what we have observed, Only around (8training set didn't repay the loan. So the dataset is imbalanced.

iii.4 Exploratory Data Analysis

Here They firstly done was drop 'Sk id curr" column which is just the loan ID that is unique for every individual. It will not contribute to the prediction algorithm. Also we are predicting target variable, so it will not in the prediction algorithm.The observation was: It's a small difference but it looks like people with no car and/or realty tend to default more than those who have. The decisions was I'll be converting these 2 categorical fields into one nominal field called 'assets'.

iii.5 Married Customer's Observations

We have a large number of married customers in our sample population. The married set also contains the most frequent defaulting individuals. 'CNT CHILDREN', 'NAME FAMILY STATUS', 'CNT FAM MEMBERS' are all related to family size. And if we only want to consider the family size for approximation of expenditures, we already have 'CNT FAM MEMBERS' wherein it accounts for the customer itself, plus spouse (if any), plus number of children (if any). Decisions: Use 'CNT FAM MEMBER' as a feature model. Fill in missing values. Drop 'CNT CHILDREN' and 'NAME FAMILY STATUS'. They have also used heatmap. This heatmap shows the correlation of those fields with null values: how strongly the presence or absence of one variable affects the presence of another. 1 : positive correlation (blue) » if a variable appears, the other variable definitely does too. 0 : zero correlation (white) » variables appearing or not appearing have no effect on one another. -1 : negative correlation (red) » if a variable appears, the other variable definitely does NOT. REGION POPULATION RELATIVE has min val = 0.00029 and max val = 0.072508. Normalized values usually range from 0 to 1, but for this sample, the maximum value is way too far from 1. Whatever values this may serve, there is a noticeable number of defaults as the value gets higher.

iii.6 Application Days

Majority of the customers apply during weekdays, with a few on weekends. The trend on customers who weren't able to repay the loan is similar with that of those who did. Suspiciously, there are people applying for a loan account as early as 3am, and it gets denser throughout the day. Do note that those who defaulted on their loan has a similar pattern with those having good records.

iii.7 Feature Model:Region

REGION POPULATION RELATIVE as a feature model. The larger the population on a certain region, the more chances of getting a customer with a bad record. Drop WEEKDAY APPR

PROCESS START. There's little chance of defaulting if the customer opens an account during Sundays, because the account opening overall on that day is relatively low compared to other days. Drop HOUR APPR PROCESS START. Similar reasoning with above. And for the final set of variables, I tagged it as 'trustworthiness' to account for trustworthy credentials, or environmental/ unconscious factors that may shape an individual's behavior. It is quite a lot, so let's break it down again per function.

iii.8 Feature model:Gender

Used CODE GENDER as a model feature. Convert the categorical text to numeric. Drop DAYS REGISTRATION, DAYS ID PUBLISH, NAME TYPE SUITE. I generated a KDE plot and I hardly see the difference. Create new feature 'age' from DAYS BIRTH. Compute for the age in years rather than using the day count.

iii.9 Feature Model: Flag Contain

Observations: All 26 'FLAG-' (flag for contacts and documents) variables have nominal categorical values: 1='YES', 0='NO'. Done: Create new feature 'FlagContact' to account for the total flag-contact variable recorded per individual. Create new feature 'FlagDocu' to account for the total flag-document variable recorded per individual

iii.10 Data Wrangling

iii.11 Feature Engineering

Feature Engineering Only few variables were processed at the data wrangling phase, mainly because most of it will be engineered to create new features from existing ones. Hopefully, this will improve the model performance.

iii.12 Model Training and Evaluation

iv. Start Here: A Gentle Introduction

iv.1 Objective

The objective of this competition is to use historical loan application data to predict whether or not an applicant will be able to repay a loan.

iv.2 Introduction

There are 7 different sources of data.

iv.3 Metric

They have used Receiver operating characteristic Curve area under the curve. According to the ROC AUC, we do not generate 0 or 1 predictions, but rather a probability between 0 and 1.

iv.4 Data Exploration

For import they have used a typical data science stack: numpy, pandas, sklearn, matplotlib. First they have read the data and then list the data. The training data has 307511 observations and 122 features (variables) including the TARGET. The test set is considerably smaller and lacks a TARGET column.

iv.5 Exploratory Data Analysis

It will tell us if it's 0= loan repaid and 1= didn't pay. After our observation, we can see that, it is an imbalanced class problem. There are far more loans that were repaid on time than loans that were not repaid.

iv.6 Missing Values

This dataframe has 122 columns. There are 67 columns that have missing values.

iv.7 Column Types

Let's look at the number of columns of each data type. float64= 65,int64=41,object= 16,dtype=int64

iv.8 Encoding

They have used label encoding and one hot encoding. For label encoding they have taken two unique categories and for one hot encoding they have taken more than two categories. After label encoding the result is : 3 columns were label encoded. After one hot encoding the result is : Training Features shape: (307511, 243) Testing Features shape: (48744, 239)

iv.9 Aligning Training and Testing Data

One-hot encoding has created more columns in the training data because there were some categorical variables with categories not represented in the testing data. To remove the columns in the training data that are not in the testing data, we need to align the dataframes. We have to remove the target column and keep it to axis=1 since we are aligning column not row. The result was Training Features shape: (307511, 240) Testing Features shape: (48744, 239)

iv.10 Anomalies

There is a problem doing EDA is anomalies within the data. These may be due to mis-typed numbers, errors in measuring equipment, or they could be valid but extreme measurements. Such as The numbers in the DAYS BIRTH column are negative because they are recorded relative to the current loan application. To see these stats in years, we can multiple by -1 and divide by the number of days in a year. It turns out that the anomalies have a lower rate of default. There are 9274 anomalies in the test data out of 48744 entries.

iv.11 Correlation

The correlation coefficient is not the greatest method to represent "relevance" of a feature, but it does give us an idea of possible relationships within the data. After observation we see that, the correlation is positive, but the value of this feature is actually negative, meaning that as the client gets older, they are less likely to default on their loan.

iv.12 Effect of age repayment

As the client gets older, there is a negative relationship with the target meaning that as clients get older, they are more likely to repay their loans on time.

iv.13 Feature Engineering

Polynomial Features Domain Knowledge Features

iv.14 Algorithm

Logistic Regression Random Forest

iv.15 Model Interpretation

iv.16 Conclusion

In this notebook, we have first understood the data, our task, and the metric that will be used to evaluate submissions. Then we ran a quick EDA to see if there were any relationships, patterns, or anomalies that could help our modeling. We had to do some preprocessing along the way, such as encoding categorical variables, impute missing values, and scale features to a range. We have followed this outline: i.Understand the problem and the data ii.Data cleaning and formatting (this was mostly done for us) iii.Exploratory Data Analysis iv.Baseline model v.Improved model vi.Model interpretation

v. Advanced Auto-Feature Engineering | Credit Default

Home Credit Default Risk competition. We will explore a few different methods for improving the set of features and incorporating domain knowledge into the final dataset. These methods include:

- Properly representing variable types
- Creating and using time variables
- Setting interesting values of variables
- Creating seed features
- Building custom primitives

Reading through the discussion around this competition and working through some of the top kernels, intricate feature engineering is a must. Using the default feature primitives in the basic notebook did improve our score, but to do better we will need some more advanced methods.

v.1 Loading Libraries

```
# pandas and numpy for data manipulation
import pandas as pd
import numpy as np

# automated feature engineering
import featuretools as ft

# Filter out pandas warnings
import warnings
warnings.filterwarnings('ignore')
```

v.2 Read in Data and Create Smaller Datasets

We will limit the data to 1000 rows because automated feature engineering is computationally intensive work. Later we can refactor this code into functions and put it in a script to run on a more powerful machine.

```
# Read in the datasets and limit to the first 1000 rows (sorted by SK_ID_CURR)
# This allows us to actually see the results in a reasonable amount of time!
app_train = pd.read_csv('../input/application_train.csv').sort_values('SK_ID_CURR').reset_index()
app_train.loc[:1000, :].drop(columns = ['index'])
app_test = pd.read_csv('../input/application_test.csv').sort_values('SK_ID_CURR').reset_index()
app_test.loc[:1000, :].drop(columns = ['index'])
bureau = pd.read_csv('../input/bureau.csv').sort_values(['SK_ID_CURR', 'SK_ID_BUREAU']).reset_index()
bureau.loc[:1000, :].drop(columns = ['index'])
bureau_balance = pd.read_csv('../input/bureau_balance.csv').sort_values('SK_ID_BUREAU').reset_index()
bureau_balance.loc[:1000, :].drop(columns = ['index'])
cash = pd.read_csv('../input/POS_CASH_balance.csv').sort_values(['SK_ID_CURR', 'SK_ID_PREV']).reset_index()
cash.loc[:1000, :].drop(columns = ['index'])
credit = pd.read_csv('../input/credit_card_balance.csv').sort_values(['SK_ID_CURR', 'SK_ID_PREV']).reset_index()
credit.loc[:1000, :].drop(columns = ['index'])
previous = pd.read_csv('../input/previous_application.csv').sort_values(['SK_ID_CURR', 'SK_ID_PREV']).reset_index()
previous.loc[:1000, :].drop(columns = ['index'])
installments = pd.read_csv('../input/installments_payments.csv').sort_values(['SK_ID_CURR', 'SK_ID_PREV']).reset_index()
installments.loc[:1000, :].drop(columns = ['index'])
```

v.3 Properly Representing Variable Types

There are a number of columns in the app dataframe that are represented as integers but are really discrete variables that can only take on a limited number of features. Some of these are Boolean flags (only 1 or 0) and two columns are ordinal (ordered discrete). To tell featurer to treat these as Boolean variables, we need to pass in the correct datatype using a dictionary mapping variable_name: variable_type.

```
app_types = {}

# Iterate through the columns and record the Boolean columns
for col in app_train:
    # If column is a number with only two values, encode it as a Boolean
    if (app_train[col].dtype != 'object') and (len(app_train[col].unique()) <= 2):
        app_types[col] = ft.variable_types.Boolean

print('Number of boolean variables: ', len(app_types))
```

```
Number of boolean variables: 33
```

There are also two ordinal variables in the app data: the rating of the region with and without the city.

```
# Record ordinal variables
app_types['REGION_RATING_CLIENT'] = ft.variable_types.Ordinal
app_types['REGION_RATING_CLIENT_W_CITY'] = ft.variable_types.Ordinal

app_test_types = app_types.copy()
del app_test_types['TARGET']
```

The previous data also has two Boolean variables.

```
# Record boolean variables in the previous data
previous_types= {'NFLAG_LAST_APPL_IN_DAY': ft.variable_types.Boolean,
                 'NFLAG_INSURED_ON_APPROVAL': ft.variable_types.Boolean}
```

v.4 Time Variables

Time can be a crucial factor in many datasets because behaviors change over time and therefore we want to make features to reflect this. For example, a client might be taking out larger and larger loans over time which could be an indicator that they are about to default or they could have a run of missed payments but then get back on track.

There are no explicit datetimes in the data, but there are relative time offsets. All the time offset are measured from the current application at Home Credit and are measured in months or days. For example, in bureau, the DAYS_CREDIT column represents "How many days before current application did client apply for Credit Bureau credit". (Credit Bureau refers to any other credit organization besides Home Credit). Although we do not know the actual application date, if we assume a starting application date that is the same for all clients, then we can convert the MONTHS_BALANCE into a datetime. This can then be treated as a relative time that we can use to find trends or identify the most recent value of a variable.

Replace Outliers There are a number of day offsets that are recorded as 365243. Reading through discussions, others replaced this number with np.nan. If we don't do this, Pandas will not be able to convert into a timedelta and throws an error that the number is too large.

```
import re

def replace_day_outliers(df):
    """Replace 365243 with np.nan in any columns with DAYS"""
    for col in df.columns:
        if "DAYS" in col:
            df[col] = df[col].replace({365243: np.nan})

    return df

# Replace all the day outliers
app_train = replace_day_outliers(app_train)
app_test = replace_day_outliers(app_test)
bureau = replace_day_outliers(bureau)
bureau_balance = replace_day_outliers(bureau_balance)
credit = replace_day_outliers(credit)
cash = replace_day_outliers(cash)
previous = replace_day_outliers(previous)
installments = replace_day_outliers(installments)
```

First we can establish an arbitrary date and then convert the time offset in months into a Pandas timedelta object.

```
# Establish a starting date for all applications at Home Credit
start_date = pd.Timestamp("2016-01-01")
start_date
```

```
Timestamp('2016-01-01 00:00:00')
```

```
# Convert to timedelta in days
for col in ['DAYS_CREDIT', 'DAYS_CREDIT_ENDDATE', 'DAYS_ENDDATE_FACT', 'DAYS_CREDIT_UPDATE']:
    bureau[col] = pd.to_timedelta(bureau[col], 'D')

bureau[['DAYS_CREDIT', 'DAYS_CREDIT_ENDDATE', 'DAYS_ENDDATE_FACT', 'DAYS_CREDIT_UPDATE']].head()
```

	DAYS_CREDIT	DAYS_CREDIT_ENDDATE	DAYS_ENDDATE_FACT	DAYS_CREDIT_UPDATE
0	-857 days	-492 days	-553 days	-155 days
1	-909 days	-179 days	-877 days	-155 days
2	-879 days	-514 days	-544 days	-155 days
3	-1572 days	-1329 days	-1328 days	-155 days
4	-559 days	902 days	NaT	-6 days

These four columns represent different offsets:

- **DAYS_CREDIT**: Number of days before current application at Home Credit client applied for loan at other financial institution. We will call this the application date, `bureau_credit_application_date` and make it the `time_index` of the entity.
- **DAYS_CREDIT_ENDDATE**: Number of days of credit remaining at time of client's application at Home Credit. We will call this the ending date, `bureau_credit_end_date`
- **DAYS_ENDDATE_FACT**: For closed credits, the number of days before current application at Home Credit that credit at other financial institution ended. We will call this the closing date, `bureau_credit_close_date`.
- **DAYS_CREDIT_UPDATE**: Number of days before current application at Home Credit that the most recent information about the previous credit arrived. We will call this the update date, `bureau_credit_update_date`.

If we were doing manual feature engineering, we might want to create new columns such as by subtracting `DAYS_CREDIT_ENDDATE` from `DAYS_CREDIT` to get the planned length of the loan in days, or subtracting `DAYS_CREDIT_ENDDATE` from `DAYS_ENDDATE_FACT` to find the number of days the client paid off the loan early. However, in this notebook we will not make any features by hand, but rather let featurer tools develop useful features for us.

To make date columns from the `timedelta`, we simply add the offset to the start date
Create the date columns

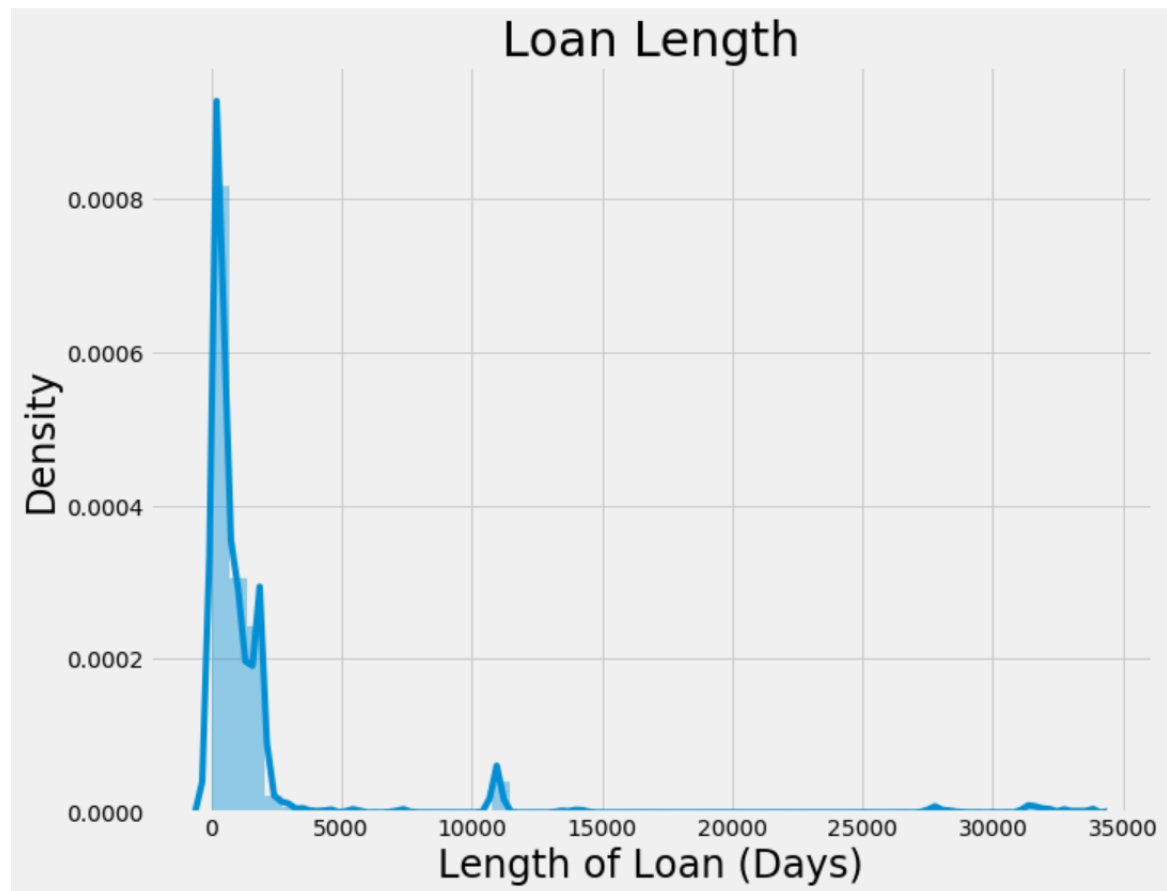
```
bureau['bureau_credit_application_date'] = start_date + bureau['DAYS_CREDIT']
bureau['bureau_credit_end_date'] = start_date + bureau['DAYS_CREDIT_ENDDATE']
bureau['bureau_credit_close_date'] = start_date + bureau['DAYS_ENDDATE_FACT']
bureau['bureau_credit_update_date'] = start_date + bureau['DAYS_CREDIT_UPDATE']
```

Plot for a sanity check To make sure the conversion went as planned, let's make a plot showing the distribution of loan lengths

```
import matplotlib.pyplot as plt
import seaborn as sns
# Set up default plot styles
plt.rcParams['font.size'] = 26
plt.style.use('fivethirtyeight')

# Drop the time offset columns
bureau = bureau.drop(columns = ['DAYS_CREDIT', 'DAYS_CREDIT_ENDDATE', 'DAYS_ENDDATE_FACT', 'DAYS_CREDIT_UPDATE'])

plt.figure(figsize = (10, 8))
sns.distplot((bureau['bureau_credit_end_date'] - bureau['bureau_credit_application_date']).dropna().dt.days);
plt.xlabel('Length of Loan (Days)', size = 24); plt.ylabel('Density', size = 24); plt.title('Loan Length', size = 30);
```



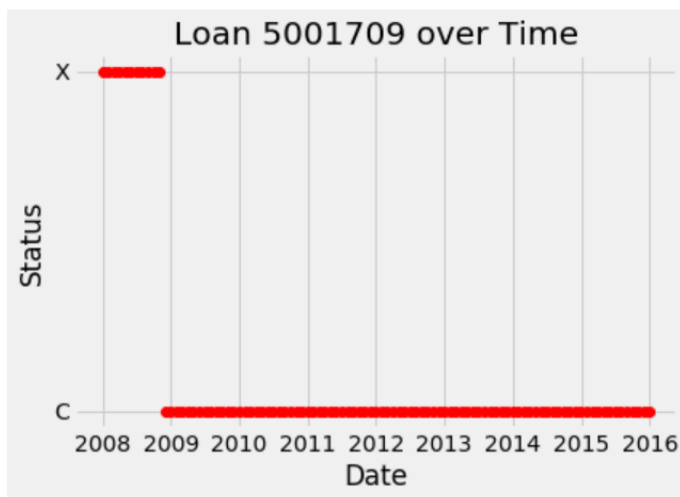
It looks as if there are a number of loans that are unreasonably long. Reading through the discussions, other people had noticed this as well. At this point, we will just leave in the outliers. We also will drop the time offset columns.

Bureau Balance The bureau balance dataframe has a MONTHS_BALANCE column that we can use as a months offset. The resulting column of dates can be used as a time_index.

```
# Convert to timedelta
bureau_balance['MONTHS_BALANCE'] = pd.to_timedelta(bureau_balance['MONTHS_BALANCE'], 'M')

# Make a date column
bureau_balance['bureau_balance_date'] = start_date + bureau_balance['MONTHS_BALANCE']
bureau_balance = bureau_balance.drop(columns = ['MONTHS_BALANCE'])

# Select one loan and plot
example_credit = bureau_balance[bureau_balance['SK_ID_BUREAU'] == 5001709]
plt.plot(example_credit['bureau_balance_date'], example_credit['STATUS'], 'ro');
plt.title('Loan 5001709 over Time'); plt.xlabel('Date'); plt.ylabel('Status');
```



Previous Applications The previous dataframe holds previous applications at Home Credit. There are a number of time offset columns in this dataset:

- **DAYS_DECISION**: number of days before current application at Home Credit that decision was made about previous application. This will be the `time_index` of the data.
- **DAYS_FIRST_DRAWING**: number of days before current application at Home Credit that first disbursement was made
- **DAYS_FIRST_DUE**: number of days before current application at Home Credit that first due was supposed to be
- **DAYS_LAST_DUE_1ST_VERSION**: number of days before current application at Home Credit that first was??
- **DAYS_LAST_DUE**: number of days before current application at Home Credit of last due date of previous application
- **DAYS_TERMINATION**: number of days before current application at Home Credit of expected termination

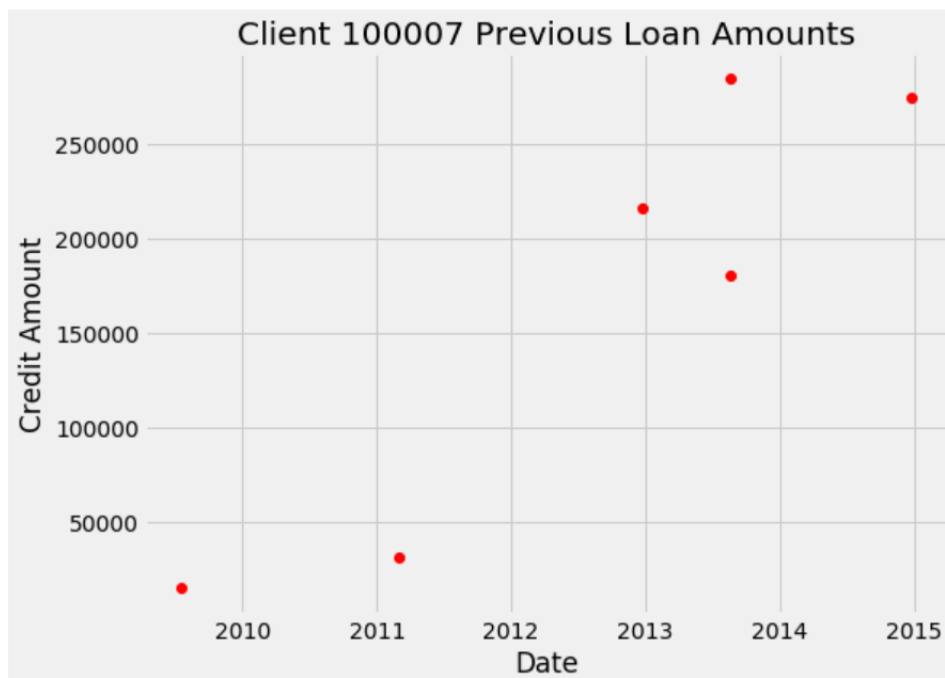
Let's convert all these into timedeltas in a loop and then make time columns.

```
# Convert to timedeltas in days
for col in ['DAYS_DECISION', 'DAYS_FIRST_DRAWING', 'DAYS_FIRST_DUE', 'DAYS_LAST_DUE_1ST_VERSION', 'DAYS_LAST_DUE', 'DAYS_TERMINATION']:
    previous[col] = pd.to_timedelta(previous[col], 'D')

# Make date columns
previous['previous_decision_date'] = start_date + previous['DAYS_DECISION']
previous['previous_drawing_date'] = start_date + previous['DAYS_FIRST_DRAWING']
previous['previous_first_due_date'] = start_date + previous['DAYS_FIRST_DUE']
previous['previous_last_duefirst_date'] = start_date + previous['DAYS_LAST_DUE_1ST_VERSION']
previous['previous_last_due_date'] = start_date + previous['DAYS_LAST_DUE']
previous['previous_termination_date'] = start_date + previous['DAYS_TERMINATION']

# Drop the time offset columns
previous = previous.drop(columns = ['DAYS_DECISION', 'DAYS_FIRST_DRAWING', 'DAYS_FIRST_DUE', 'DAYS_LAST_DUE_1ST_VERSION', 'DAYS_LAST_DUE', 'DAYS_TERMINATION'])

plt.figure(figsize = (8, 6))
example_client = previous[previous['SK_ID_CURR'] == 100007]
plt.plot(example_client['previous_decision_date'], example_client['AMT_CREDIT'], 'ro')
plt.title('Client 100007 Previous Loan Amounts'); plt.xlabel('Date'); plt.ylabel('Credit Amount');
```



Previous Credit and Cash The `credit_card_balance` and `POS_CASH_balance` each have a `MONTHS_BALANCE` column with the month offset. This is the number of months before the current application at Home Credit of the previous application record. These will represent the

time_index of the data.

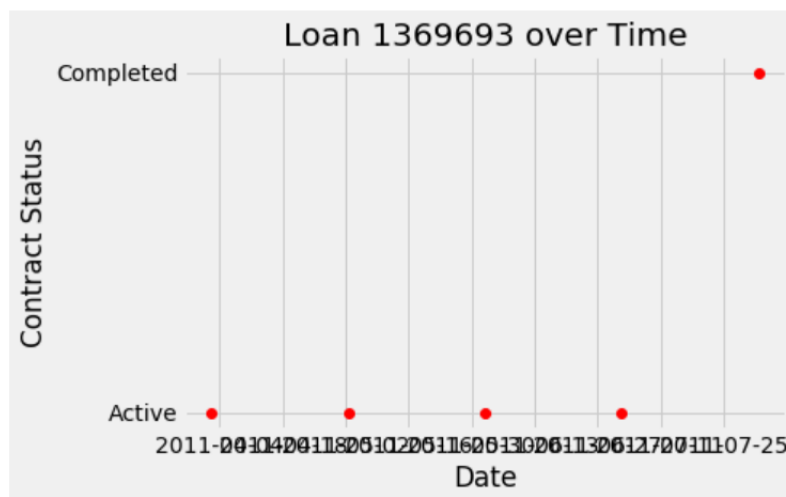
```
# Convert to timedelta objects
credit['MONTHS_BALANCE'] = pd.to_timedelta(credit['MONTHS_BALANCE'], 'M')
cash['MONTHS_BALANCE'] = pd.to_timedelta(cash['MONTHS_BALANCE'], 'M')

# Make a date column
credit['credit_balance_date'] = start_date + credit['MONTHS_BALANCE']
credit = credit.drop(columns = ['MONTHS_BALANCE'])

# Make a date column
cash['cash_balance_date'] = start_date + cash['MONTHS_BALANCE']
cash = cash.drop(columns = ['MONTHS_BALANCE'])

# Select on loan and plot
example_credit = cash[cash['SK_ID_PREV'] == 1369693]

plt.plot(example_credit['cash_balance_date'], example_credit['NAME_CONTRACT_STATUS'], 'ro');
plt.title('Loan 1369693 over Time'); plt.xlabel('Date'); plt.ylabel('Contract Status');
```



Installments Payments The installments_payments data contains information on each payment made on the previous loans at Home Credit. It has two date offset columns:

- DAYS_INSTALLMENT: number of days before current application at Home Credit that previous installment was supposed to be paid
- DAYS_ENTRY_PAYMENT: number of days before current application at Home Credit that previous installment was actually paid

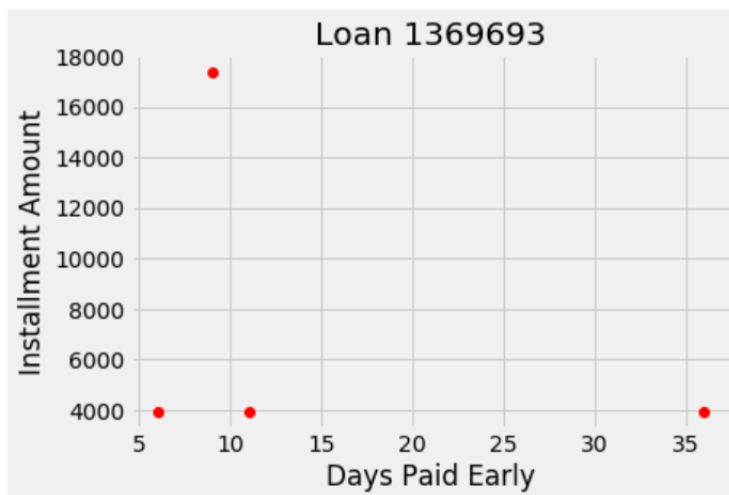
By now the process should be familiar: convert to timedeltas and then make time columns. The DAYS_INSTALLMENT will serve as the time_index.

```
# Conver to time delta object
installments['DAYS_INSTALMENT'] = pd.to_timedelta(installments['DAYS_INSTALMENT'], 'D')
installments['DAYS_ENTRY_PAYMENT'] = pd.to_timedelta(installments['DAYS_ENTRY_PAYMENT'], 'D')

# Create time column and drop
installments['installments_due_date'] = start_date + installments['DAYS_INSTALMENT']
installments = installments.drop(columns = ['DAYS_INSTALMENT'])

installments['installments_paid_date'] = start_date + installments['DAYS_ENTRY_PAYMENT']
installments = installments.drop(columns = ['DAYS_ENTRY_PAYMENT'])

# Select one loan and plot
example_credit = installments[installments['SK_ID_PREV'] == 1369693]
plt.plot((example_credit['installments_due_date'] - example_credit['installments_paid_date']).dt.days, example_credit['AMT_INSTALMENT'], 'ro');
plt.title('Loan 1369693'); plt.xlabel('Days Paid Early'); plt.ylabel('Installment Amount');
```



v.5 Applying Featuretools

We can now start making features using the time columns. We will create an entityset named clients much as before, but now we have time variables that we can use.

```
#Make an entityset
es = ft.EntitySet(id = 'clients')
```

Entities When creating the entities, we specify the index, the time_index (if present), and the variable_types (if they need to be specified).

```
# Entities with a unique index
es = es.entity_from_dataframe(entity_id = 'app_train', dataframe = app_train,
                             index = 'SK_ID_CURR', variable_types = app_types)

es = es.entity_from_dataframe(entity_id = 'app_test', dataframe = app_test,
                             index = 'SK_ID_CURR', variable_types = app_test_types)

es = es.entity_from_dataframe(entity_id = 'bureau', dataframe = bureau,
                             index = 'SK_ID_BUREAU', time_index='bureau_credit_application_date')

es = es.entity_from_dataframe(entity_id = 'previous', dataframe = previous,
                             index = 'SK_ID_PREV', time_index = 'previous_decision_date',
                             variable_types = previous_types)

# Entities that do not have a unique index
es = es.entity_from_dataframe(entity_id = 'bureau_balance', dataframe = bureau_balance,
                             make_index = True, index = 'bb_index',
                             time_index = 'bureau_balance_date')

es = es.entity_from_dataframe(entity_id = 'cash', dataframe = cash,
                             make_index = True, index = 'cash_index',
                             time_index = 'cash_balance_date')

es = es.entity_from_dataframe(entity_id = 'installments', dataframe = installments,
                             make_index = True, index = 'installments_index',
                             time_index = 'installments_paid_date')

es = es.entity_from_dataframe(entity_id = 'credit', dataframe = credit,
                             make_index = True, index = 'credit_index',
                             time_index = 'credit_balance_date')
```

v.6 Relationships

Not surprisingly, the relationships between tables has not changed since the previous implementation.

```
# Relationship between app and bureau
r_app_bureau = ft.Relationship(es['app_train']['SK_ID_CURR'], es['bureau']['SK_ID_CURR'])

# Test Relationship between app and bureau
r_test_app_bureau = ft.Relationship(es['app_test']['SK_ID_CURR'], es['bureau']['SK_ID_CURR'])

# Relationship between bureau and bureau balance
r_bureau_balance = ft.Relationship(es['bureau']['SK_ID_BUREAU'], es['bureau_balance']['SK_ID_BUREAU'])

# Relationship between current app and previous apps
r_app_previous = ft.Relationship(es['app_train']['SK_ID_CURR'], es['previous']['SK_ID_CURR'])

# Test Relationship between current app and previous apps
r_test_app_previous = ft.Relationship(es['app_test']['SK_ID_CURR'], es['previous']['SK_ID_CURR'])

# Relationships between previous apps and cash, installments, and credit
r_previous_cash = ft.Relationship(es['previous']['SK_ID_PREV'], es['cash']['SK_ID_PREV'])
r_previous_installments = ft.Relationship(es['previous']['SK_ID_PREV'], es['installments']['SK_ID_PREV'])
r_previous_credit = ft.Relationship(es['previous']['SK_ID_PREV'], es['credit']['SK_ID_PREV'])

# Add in the defined relationships
es = es.add_relationships([r_app_bureau, r_test_app_bureau, r_bureau_balance, r_app_previous, r_test_app_previous,
                           r_previous_cash, r_previous_installments, r_previous_credit])

# Print out the EntitySet
es
```

```
Entityset: clients
Entities:
  app_train [Rows: 1001, Columns: 122]
  app_test [Rows: 1001, Columns: 121]
  bureau [Rows: 1001, Columns: 17]
  previous [Rows: 1001, Columns: 37]
  bureau_balance [Rows: 1001, Columns: 4]
  cash [Rows: 1001, Columns: 9]
  installments [Rows: 1001, Columns: 9]
  credit [Rows: 1001, Columns: 24]
Relationships:
  bureau.SK_ID_CURR -> app_train.SK_ID_CURR
  bureau.SK_ID_CURR -> app_test.SK_ID_CURR
  bureau_balance.SK_ID_BUREAU -> bureau.SK_ID_BUREAU
  previous.SK_ID_CURR -> app_train.SK_ID_CURR
  previous.SK_ID_CURR -> app_test.SK_ID_CURR
  cash.SK_ID_PREV -> previous.SK_ID_PREV
  installments.SK_ID_PREV -> previous.SK_ID_PREV
  credit.SK_ID_PREV -> previous.SK_ID_PREV
```

v.7 Time Features

Let's look at some of the time features we can make from the new time variables. Because these times are relative and not absolute, we are only interested in values that show change over time, such as trend or cumulative sum. We would not want to calculate values like the year or month

since we choose an arbitrary starting date.

Throughout this notebook, we will pass in a `chunk_size` to the `dfs` call which specifies the number of rows (if an integer) or the fraction of rows to use in each chunk (if a float). This can help to optimize the `dfs` procedure, and the `chunk_size` can have a significant effect on the run time. Here we will use a chunk size equal to the number of rows in the data so all the results will be calculated in one pass. We also want to avoid making any features with the testing data, so we pass in `ignore_entities = [app_test]`.

```
time_features, time_feature_names = ft.dfs(entityset = es, target_entity = 'app_train',
                                           trans_primitives = ['cum_sum', 'time_since_previous'], max_depth = 2,
                                           agg_primitives = ['trend'],
                                           features_only = False, verbose = True,
                                           chunk_size = len(app_train),
                                           ignore_entities = ['app_test'])
```

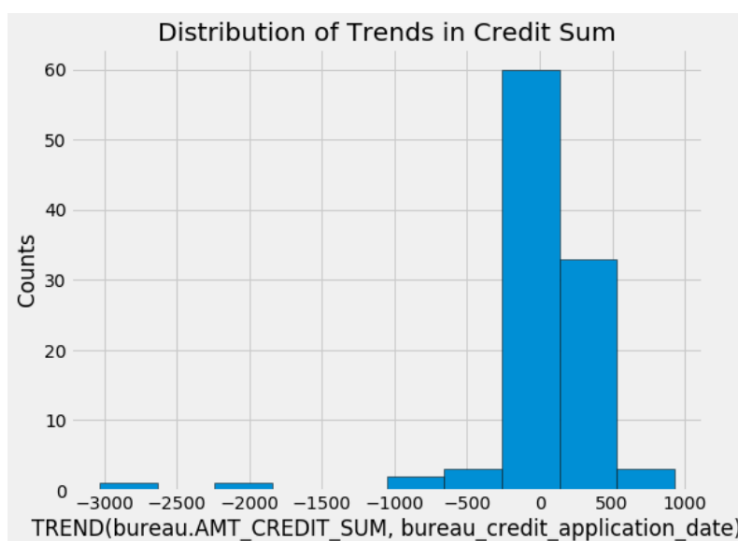
Built 221 features

Elapsed: 00:35 | Remaining: 00:00 | Progress: 100% | Calculated: 1/1 chunks

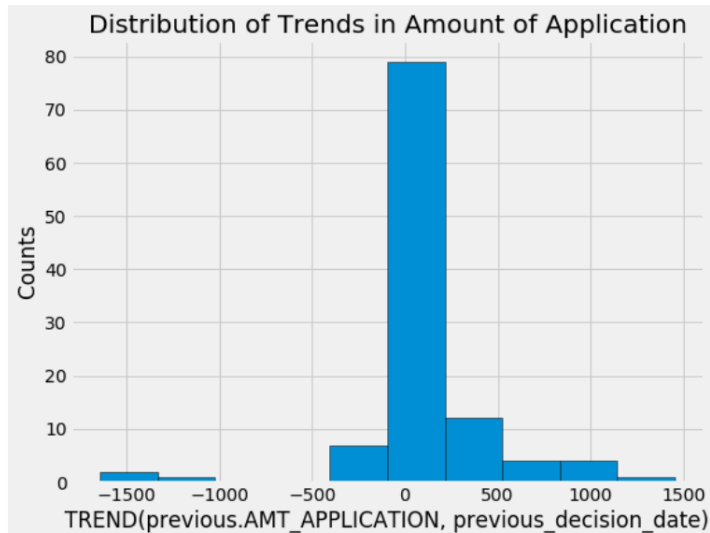
```
time_features.iloc[:, -10:].head()
```

Let's visualize one of these new variables. We can look at the trend in credit size over time. A positive value indicates that the loan size for the client is increasing over time

```
plt.figure(figsize = (8, 6))
plt.hist(time_features['TREND(bureau.AMT_CREDIT_SUM, bureau_credit_application_date)'].dropna(), edgecolor = 'k');
plt.xlabel('TREND(bureau.AMT_CREDIT_SUM, bureau_credit_application_date)'); plt.ylabel('Counts');
plt.title('Distribution of Trends in Credit Sum');
```



```
plt.figure(figsize = (8, 6))
plt.hist(time_features['TREND(previous.AMT_APPLICATION, previous_decision_date)'].dropna(), edgecolor = 'k');
plt.xlabel('TREND(previous.AMT_APPLICATION, previous_decision_date)'); plt.ylabel('Counts'); plt.title('Distribution of Trends in Amount of Application');
```



```
time_feature_names[-10:]
```

```
[<Feature: TREND(previous.TREND(credit.AMT_DRAWINGS_POS_CURRENT, credit_balance_date), previous_decision_date)>,
 <Feature: TREND(bureau.CUM_SUM(AMT_ANNUITY by SK_ID_CURR), bureau_credit_application_date)>,
 <Feature: TREND(bureau.CUM_SUM(CREDIT_DAY_OVERDUE by SK_ID_CURR), bureau_credit_application_date)>,
 <Feature: TREND(bureau.CUM_SUM(AMT_CREDIT_SUM_DEBT by SK_ID_CURR), bureau_credit_application_date)>,
 <Feature: TREND(bureau.CUM_SUM(AMT_CREDIT_MAX_OVERDUE by SK_ID_CURR), bureau_credit_application_date)>,
 <Feature: TREND(bureau.time_since_previous_by_SK_ID_CURR, bureau_credit_application_date)>,
 <Feature: TREND(bureau.CUM_SUM(AMT_CREDIT_SUM_LIMIT by SK_ID_CURR), bureau_credit_application_date)>,
 <Feature: TREND(bureau.CUM_SUM(AMT_CREDIT_SUM_OVERDUE by SK_ID_CURR), bureau_credit_application_date)>,
 <Feature: TREND(bureau.CUM_SUM(CNT_CREDIT_PROLONG by SK_ID_CURR), bureau_credit_application_date)>,
 <Feature: TREND(bureau.CUM_SUM(AMT_CREDIT_SUM by SK_ID_CURR), bureau_credit_application_date)>]
```

v.8 Interesting Values

Another method we can use in featuretools is "interesting values." Specifying interesting values will calculate new features conditioned on values of existing features. For example, we can create new features that are conditioned on the value of NAME_CONTRACT_STATUS in the previous dataframe. Each stat will be calculated for the specified interesting values which can be useful when we know that there are certain indicators that are of greater importance in the

data.

```
previous['NAME_CONTRACT_STATUS'].value_counts()
```

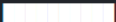
```
Approved      594
Refused       206
Canceled      186
Unused offer   15
Name: NAME_CONTRACT_STATUS, dtype: int64
```

To use interesting values, we assign them to the variable and then specify the `where_primitives` in the `dfs` call.

```
# Assign interesting values
es['previous']['NAME_CONTRACT_STATUS'].interesting_values = ['Approved', 'Refused', 'Canceled']

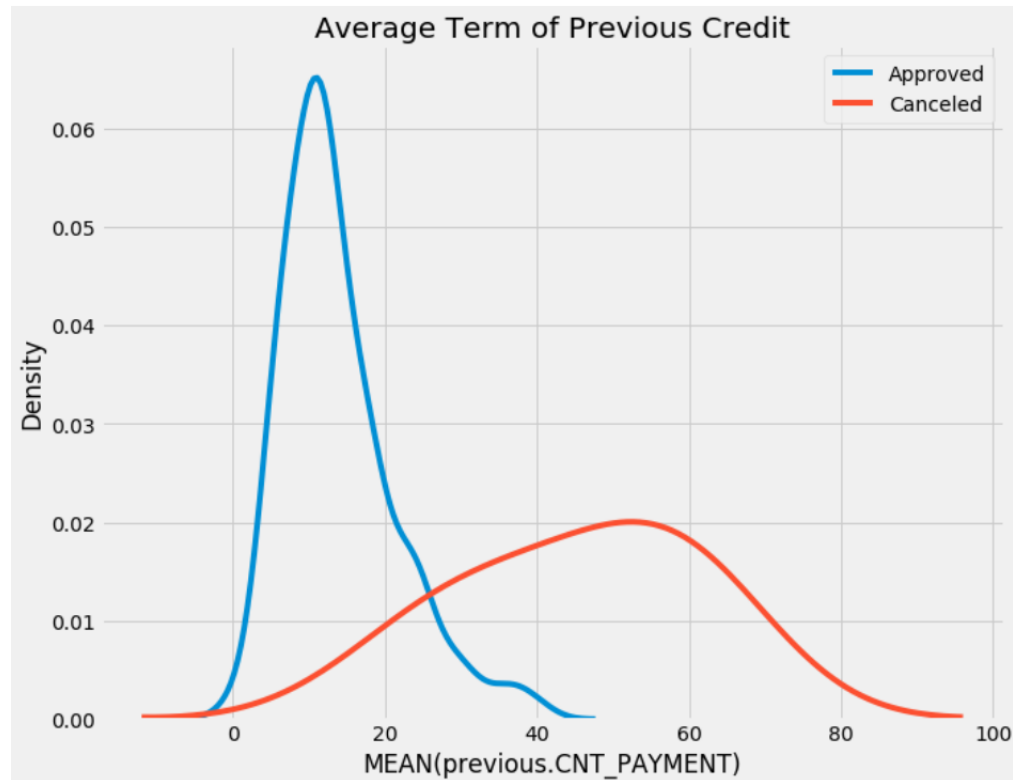
# Calculate the features with interesting values
interesting_features, interesting_feature_names = ft.dfs(entityset=es, target_entity='app_train',
max_depth = 1,
where_primitives = ['mean', 'mode'],
trans_primitives=[], features_only = False,
verbose = True,
chunk_size = len(app_train),
ignore_entities = ['app_test'])
```

Built 355 features

Elapsed: 00:06 | Remaining: 00:00 | Progress: 100%|  | Calculated: 1/1 chunks

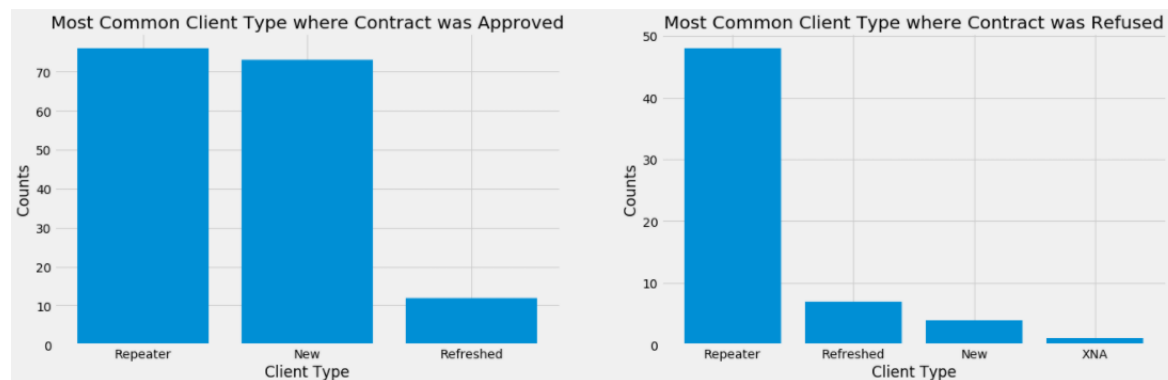
One of the features is `MEAN(previous.CNT_PAYMENT WHERE NAME_CONTRACT_STATUS = Approved)`. This shows the average "term of previous credit" on previous loans conditioned on the previous loan being approved. We can compare the distribution of this feature to the `MEAN(previous.CNT_PAYMENT WHERE NAME_CONTRACT_STATUS = Canceled)` to see how these loans differ.

```
plt.figure(figsize = (10, 8))
sns.kdeplot(interesting_features['MEAN(previous.CNT_PAYMENT WHERE NAME_CONTRACT_STATUS = Approved)'].dropna(), label = 'Approved')
sns.kdeplot(interesting_features['MEAN(previous.CNT_PAYMENT WHERE NAME_CONTRACT_STATUS = Canceled)'].dropna(), label = 'Canceled')
plt.xlabel('MEAN(previous.CNT_PAYMENT)'); plt.ylabel('Density'); plt.title('Average Term of Previous Credit');
```




```
# Plot of client type when contract was approved
plt.figure(figsize = (20, 6))
plt.subplot(1, 2, 1)
plt.bar(list(range(3)), interesting_features['MODE(previous.NAME_CLIENT_TYPE WHERE NAME_CONTRACT_STATUS = Approved)'].value_counts())
plt.xticks(list(range(3)), interesting_features['MODE(previous.NAME_CLIENT_TYPE WHERE NAME_CONTRACT_STATUS = Approved)'].value_counts().index);
plt.xlabel("Client Type"); plt.ylabel("Counts");
plt.title("Most Common Client Type where Contract was Approved");

# Plot of client type where contract was refused
plt.subplot(1, 2, 2)
plt.bar(list(range(4)), interesting_features['MODE(previous.NAME_CLIENT_TYPE WHERE NAME_CONTRACT_STATUS = Refused)'].value_counts())
plt.xticks(list(range(4)), interesting_features['MODE(previous.NAME_CLIENT_TYPE WHERE NAME_CONTRACT_STATUS = Refused)'].value_counts().index);
plt.xlabel("Client Type"); plt.ylabel("Counts");
plt.title("Most Common Client Type where Contract was Refused");
```



Based on the most important features returned by a model, we can create new interesting features. This is one area where we can apply domain knowledge to feature creation.

v.9 Seed Features

An additional extension to the default aggregations and transformations is to use seed features. These are user defined features that we provide to deep feature synthesis that can then be built on top of where possible.

As an example, we can create a seed feature that determines whether or not a payment was late. This time when we make the dfs function call, we need to pass in the seed_features argument.

Review of Relevant Work for Analyzing default-ability of Home Loan Lenders using Machine Learning Models

```
# Late Payment seed feature
late_payment = ft.Feature(es['installments']['installments_due_date']) < ft.Feature(es['installments']['installments_paid_date'])

# Rename the feature
late_payment = late_payment.rename("late_payment")

# DFS with seed features
seed_features, seed_feature_names = ft.dfs(entityset = es,
                                           target_entity = 'app_train',
                                           agg_primitives = ['percent_true', 'mean'],
                                           trans_primitives = [],
                                           seed_features = [late_payment],
                                           features_only = False, verbose = True,
                                           chunk_size = len(app_train),
                                           ignore_entities = ['app_test'])
```

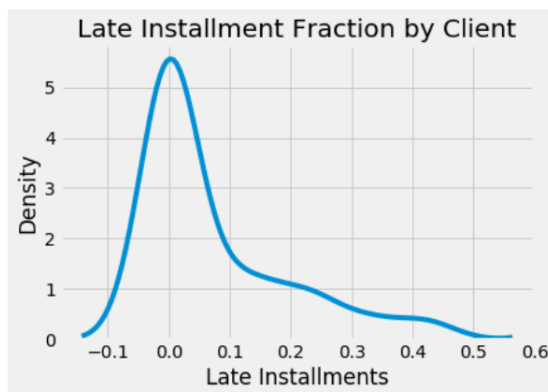
Built 204 features

Elapsed: 00:01 | Remaining: 00:00 | Progress: 100% | Calculated: 1/1 chunks

```
seed_features.iloc[:, -2:].head(10)
```

	PERCENT_TRUE(installments.late_payment)	MEAN(previous.PERCENT_TRUE(installments.late_payment))
SK_ID_CURR		
100002	0.000000	0.000000
100003	0.000000	0.000000
100004	0.000000	0.000000
100006	0.000000	0.000000
100007	0.242424	0.287912
100008	0.028571	0.022727
100009	0.000000	0.000000
100010	0.000000	0.000000
100011	0.146154	0.277160
100012	0.104167	0.118687

```
sns.kdeplot(seed_features['PERCENT_TRUE(installments.late_payment)'].dropna(), label = '')
plt.xlabel('Late Installments'); plt.ylabel('Density'); plt.title('Late Installment Fraction by Client');
```



v.10 Putting it all Together

Finally, we can run deep feature synthesis with the time variables, with the correct specified categorical variables, with the interesting features, with the seed features, and with the custom features. To actually run this on the entire dataset, we can take the code here, put it in a script, and then use more computational resources.

```
# Run and create the features
feature_matrix, feature_names = ft.dfs(entityset = es, target_entity = 'app_train',
                                     agg_primitives = ['mean', 'max', 'min', 'trend', 'mode',
                                     'count',
                                     'sum', 'percent_true', NormalizedModeC
count, MostRecent, LongestSeq],
                                     trans_primitives = ['diff', 'cum_sum', 'cum_mean', 'perc
entile'],
                                     where_primitives = ['mean', 'sum'],
                                     seed_features = [late_payment, past_due],
                                     max_depth = 2, features_only = False, verbose = True,
                                     chunk_size = len(app_train),
                                     ignore_entities = ['app_test'])
```

```
Built 3223 features
Elapsed: 03:52 | Remaining: 00:00 | Progress: 100% | Calculated: 1/1 chunks
```

We will now do the same operation applied to the test set. Doing the calculations separately should prevent leakage from the testing data into the training data.

```
# Run and create the features
feature_matrix_test, feature_names_test = ft.dfs(entityset = es, target_entity = 'app_test',
                                                agg_primitives = ['mean', 'max', 'min', 'tre
nd', 'mode', 'count',
                                                'sum', 'percent_true', Nor
malizedModeCount, MostRecent, LongestSeq],
                                                trans_primitives = ['diff', 'cum_sum', 'cum_
mean', 'percentile'],
                                                where_primitives = ['mean', 'sum'],
                                                seed_features = [late_payment, past_due],
                                                max_depth = 2, features_only = False, verbos
e = True,
                                                chunk_size = len(app_test),
                                                ignore_entities = ['app_train'])
```

```
Built 3222 features
Elapsed: 00:50 | Remaining: 00:00 | Progress: 100% | Calculated: 1/1 chunks
```

```
import random
random.sample(feature_names, 10)
```

```
[<Feature: MEAN(previous.CUM_SUM(AMT_DOWN_PAYMENT by SK_ID_CURR) WHERE NAME_CONTRACT_STATUS = Approved)>,
 <Feature: SUM(previous.TREND(credit.AMT_BALANCE, credit_balance_date) WHERE NAME_CONTRACT_STATUS = Refused)>,
 <Feature: MEAN(credit.CNT_DRAWINGS_CURRENT)>,
 <Feature: TREND(credit.SK_ID_CURR, credit_balance_date)>,
 <Feature: MAX(previous.PERCENT_TRUE(installments.late_payment))>,
 <Feature: MEAN(previous.TREND(credit.AMT_RECEIVABLE_PRINCIPAL, credit_balance_date) WHERE NAME_CONTRACT_STATUS = Canceled)>,
 <Feature: NONLIVINGAPARTMENTS_MEDI>,
 <Feature: MIN(previous.CUM_MEAN(AMT_GOODS_PRICE by SK_ID_CURR))>,
 <Feature: MEAN(previous.MAX(installments.NUM_INSTALLMENT_NUMBER) WHERE NAME_CONTRACT_STATUS = Canceled)>,
 <Feature: MEAN(previous.SUM(credit.CNT_INSTALLMENT_MATURE_CUM))>]
```

v.11 Conclusion

we explored some of the advanced functionality in featuretools including:

Time Variables: allow us to track trends over time
Interesting Variables: condition new features on values of existing features
Seed Features: define new features manually that featuretools will then build on top of
Custom feature primitives: design any transformation or aggregation feature that can incorporate domain knowledge
We can use these methods to encode domain knowledge about a problem into our features or create features based on what others have found useful. The next step from here would be to run the script on the entire dataset, then use the features for modeling. We could use the feature importances from the model to determine the most relevant features, perform feature selection, and then go through another round of feature synthesis with a new set of primitives, seed features, and interesting features. As with many aspects of machine learning, feature creation is largely an empirical and iterative procedure.

vi. Home Credit Default Report 2

vi.1 Objective

The main objective is to identify the potential Defaulters based on the given data about the applicants.

The probability of classification is essential because we want to be very sure when we classify someone as a Non-Defaulter, as the cost of making a mistake can be very high to the company.

vi.2 Metric

ROC-AUC Score: This metric is insensitive to class imbalance. It works by ranking the probabilities of prediction of the positive class label and calculating the Area under the ROC Curve which is plotted between True Positive Rates and False Positive Rates for each threshold value.

Recall Score: It is the ratio of the True Positives predicted by the model and the total number of Actual Positives. It is also known as True Positive Rate.

Precision Score: It is the ratio of True Positives and the Total Positives predicted by the model.

Confusion Matrix : The confusion matrix helps us to visualize the mistakes made by the model on each of the classes, be it positive or negative. Hence, it tells us about misclassifications for both classes.

vi.3 Dataset Description

The dataset provided contains lots of details about the borrower. It is segregated into multiple relational tables, which contain applicants' static data such as their gender, age, number of family members, occupation, and other related fields, applicant's previous credit history obtained from the credit bureau department, and the applicant's past credit history within the Home Credit Group itself. The dataset is an imbalanced dataset, where the Negative class dominates the Positive class, as there are only a few number of defaulters among all the applicants.

vi.4 Exploratory Data Analysis

We will first start by checking the shapes of the tables in hand. Since there are 8 tables in total, we will load each table, and print their shapes and a few of their rows.

We also found that all the tables contain large number of missing values, and thus it is important to address the counts of the missing values. We have plotted Bar Plots for the missing values of each column for each table.

vi.5 Distribution of Target Variable

Now, let us have a look at the distribution of the Target Variable in our Training Dataset. We see that there are only 8% (24.8k) Defaulters, and 91.9% (282.6k) Non-Defaulters in the train dataset. This shows that the Positive class is a minority class in our dataset. It also implies that it is an Imbalanced Dataset, and we need to come up with adequate ways to handle it.

vi.6 Column Types

Let's look at the number of columns of each data type. float64= 65,int64=41,object= 16,dtype=int64

vi.7 Analysis of Continuous Variables

For continuous variables, we have used Box-Plots, Violin-Plots, and PDFs extensively. We have plotted the distribution of data-points for both Defaulters and Non-Defaulters together, and try to see if these distributions are similar or distinguishable. Similar to code for plotting categorical variables, we have generalized the code for continuous variables as well.

vi.8 Algorithm

Logistic Regression
Random Forest

vi.9 Model Interpretation

vi.10 Conclusion

We also noticed some correlated features from the correlation analysis, which would be increasing the dimensionality of data without adding much value. We would ideally want to remove such features, provided they don't degrade the performance of the model. The dataset is imbalanced, and we would need to come up with techniques to handle it. For Default Risk prediction, the Defaulters usually tend to have some behavior which deviate from the normal, and thus, we cannot remove outliers or far-off points, as they may suggest some important Defaulting tendency. With all these observations and insights in mind, we will move to the Data Cleaning and Feature Engineering task.

REFERENCES

- [1] Abdelwahed Assklou and Aguiar. 7th solution - **Not great things but small things in a great way.** *Kaggle*.
- [2] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, Tie-Yan Liu (2017). LightGBM: A Highly Efficient Gradient Boosting Decision Tree. *Academia.edu*.
- [3] DC Montgomery, EA Peck and GG Vining (2021). Introduction to linear regression analysis. *John Wiley & Sons*.
- [4] N.A.Diamantidis, D.Karlis and E.A.Giakoumakis, (2000). Unsupervised stratification of cross-validation for accuracy estimation. *Elsevier, Artificial Intelligence*, Volume 116, Issues 1-2, January 2000, Pages 1-16.
- [5] Andreea Anghel, Nikolaos Papandreou, Thomas Parnell, Alessandro De Palma, Haralampos Pozidis (2019) Benchmarking and Optimization of Gradient Boosting Decision Tree Algorithms *arXiv* arXiv:1809.04559 .
- [6] KP Balanda, HL MacGillivray (1988). Kurtosis: a critical review *The American Statistician*.