CREDIT RISK MODELING

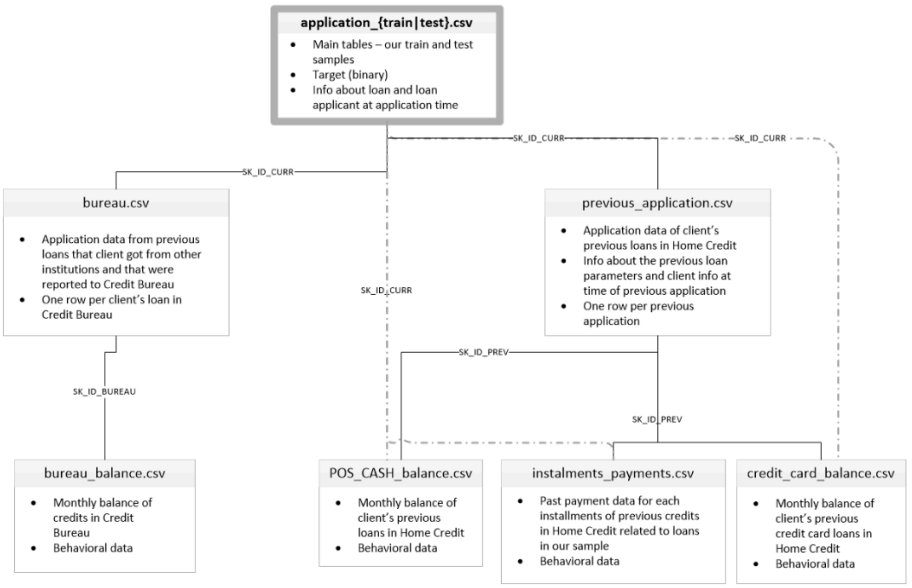
Group 5

1. Business/Real-world Problem

Nowadays, the demand for loans in banks and financial institutions is rising. The pandemic has just ended, and one after another, outbreaks of wars destabilize the global economy; it is of no surprise that more and more people are starting to resort to credit. However, the number of accepted applications for loans is much less compared to the number of applicants, and this is primarily caused because of insufficient information about borrowers. Most of them either have non-existent credit histories, or their current histories are too little to predict. This situation is both unpleasant for lenders and borrowers.

To ensure a positive loan taking experience for applicants, we developed a model that can assist lenders with this problem. With a small amount of data, we train a model to predict borrowers' repayment abilities. This solution will reduce the turn down rate of applications, while keeping the credit risk rate low for lenders.

2. Modeling Problem



Schema for the dataset

3. Data Analysis

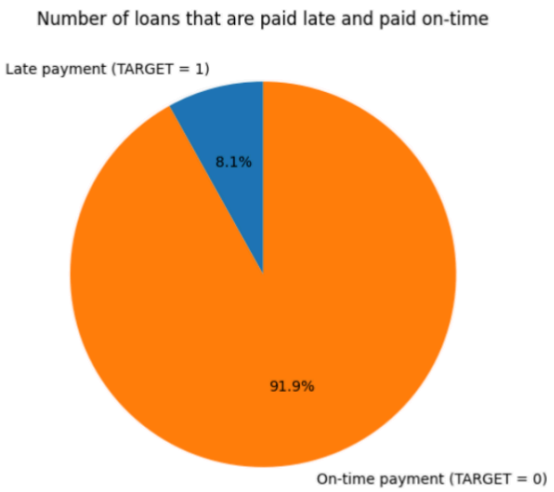
3.1. Check for missing data

We have checked for duplicate rows and null values across all tables and found that only the ***bureau_balance*** table has no null values. Additionally, none of the seven tables contain duplicate rows.

3.2. Data exploration

- application_{train|test}

There are 123 and 122 columns in table ***train*** and ***test*** respectively. So we just show the columns that we customized in the following steps.



The pie chart indicates that most loans are repaid on time. The table below shows the value count of *CODE_GENDER,* as there are a very low number of XNA values, we will replace all of XNA by mode value.

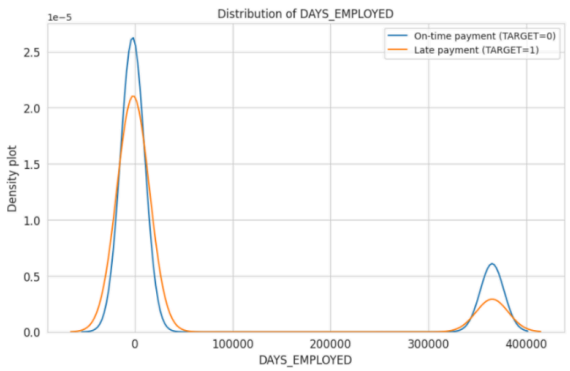
	CODE_GENDER	COUNT
0	F	161856
1	M	84150
2	XNA	3

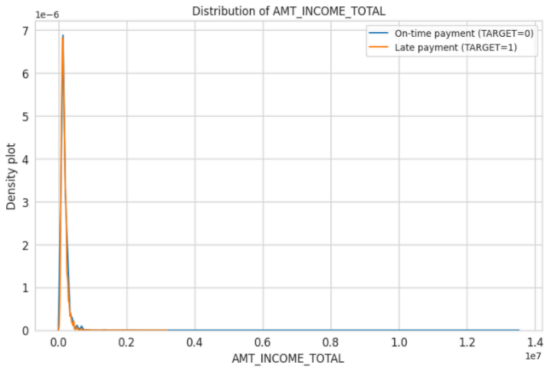
The next table demonstrates the values in the *CNT_FAM_MEMBERS* column. We also replace NaN value by the mode value, which is 2.0, because there is only one NaN.

	Unique Value	Value Count
0	11.0	1
1	NaN	1
2	13.0	1
3	15.0	1
4	16.0	2
5	20.0	2
6	12.0	2
7	10.0	2
8	9.0	4

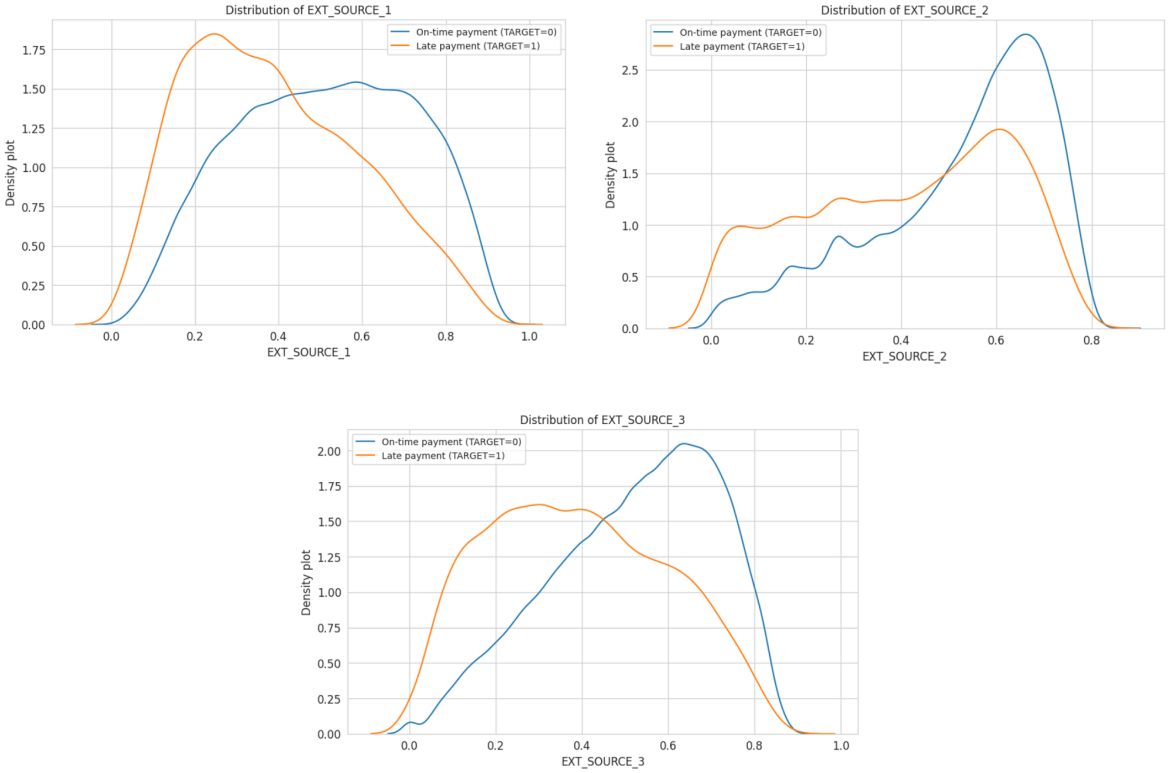
	Unique Value	Value Count
9	8.0	18
10	7.0	63
11	6.0	310
12	5.0	2733
13	4.0	19752
14	3.0	42092
15	1.0	54069
16	2.0	126956

Next, the first chart shows the distribution of *DAYS_EMPLOYED* which have been normalized in y axis and we can see that there is an outlier between 300,000 and 400,000, which is approximately 1000 years (really unreasonable). As same as the left chart, this chart shows an outlier of *AMT_INCOME_TOTAL*. So in the following steps we have to replace the outliers by null values.





In these 3 below charts, the clear separation in the density plots between on-time and late payments indicates that higher values of these features are strongly associated with on-time repayment. This makes *EXT_SOURCE_1, 2,* and *3* valuable features in credit risk modeling.



- In other tables, we analyzed and created some features that we think are useful in evaluating whether a client pays late or not. This will be indicated in the following part.

4. Feature Engineering

4.1. Preprocessing

- application_{train|test}

In our credit risk model, we identified four critical columns (*AMT_INCOME_TOTAL, AMT_CREDIT, AMT_ANNUITY, AMT_GOODS_PRICE*) and a few calculated ratios that significantly influence the model's ability to predict whether a customer will repay their loan on time. Firstly, in column *AMT_INCOME_TOTAL*, higher income usually indicates a better repayment ability but must be considered alongside the loan amount. Moreover, larger loans require larger monthly payments (*AMT_CREDIT*), influencing repayment ability. For *AMT_ANNUITY*, higher annuity amounts can increase default risk and the price of goods financed by the loan (*AMT_GOODS_PRICE*) helps in understanding the purpose and repayment capacity related to the loan.

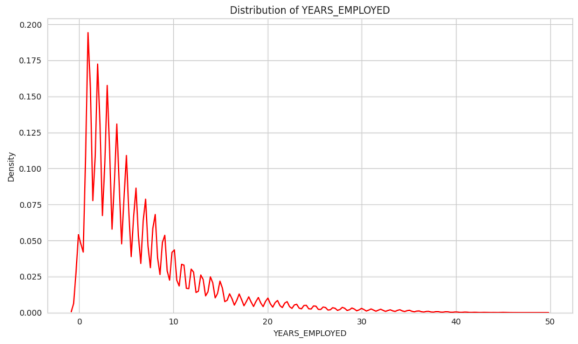
Beside these columns, we also aggregate a few columns that may contribute to the prediction of the repayment ability: *ANNUITY_INCOME_RATIO, CREDIT_GOODSPRICE_RATIO, CREDIT_INCOME_RATIO, CREDIT_ANNUITY_RATIO*. *ANNUITY_INCOME_RATIO* measures the proportion of income allocated to loan payments. *CREDIT_GOODSPRICE_RATIO* indicates how much of the loan is spent on goods. *CREDIT_INCOME_RATIO* estimates the loan amount relative to the borrower's income and *CREDIT_ANNUITY_RATIO* approximates the loan term duration. Because there are some elements equal 0 so we add 0.000001 to the denominator to make sure the ratio will not be infinitive.

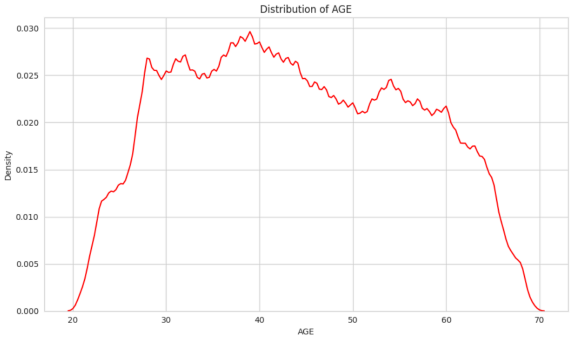
```
def get_thresh(feature):
    """ Outliers are usually > 3 standard deviations away from the mean. """
    ave=np.mean(train[feature])
    sdev=np.std(train[feature])
    threshold=round(ave+(3*sdev),2)
    print('Threshold for',feature,':',threshold)
    return threshold
```

```
thresh_income_train = get_thresh('AMT_INCOME_TOTAL')
thresh_employment_train = get_thresh('DAYS_EMPLOYED')

thresh_income_test = get_thresh('AMT_INCOME_TOTAL')
thresh_employment_test = get_thresh('DAYS_EMPLOYED')

anomalous_employment_train = train[train['DAYS_EMPLOYED'] > 0]
normal_employment_train = train[train['DAYS_EMPLOYED'] < 0]

anomalous_employment_test = test[test['DAYS_EMPLOYED'] > 0]
normal_employment_test = test[test['DAYS_EMPLOYED'] < 0]
```

```
train['DAYS_EMPLOYED'] = train['DAYS_EMPLOYED'].mask(train['DAYS_EMPLOYED'] > 0)
train['AMT_INCOME_TOTAL'] = train['AMT_INCOME_TOTAL'].mask(train['AMT_INCOME_TOTAL'] > thresh_emp
loyment_train)
test['DAYS_EMPLOYED'] = test['DAYS_EMPLOYED'].mask(test['DAYS_EMPLOYED'] > 0)
test['AMT_INCOME_TOTAL'] = test['AMT_INCOME_TOTAL'].mask(test['AMT_INCOME_TOTAL'] > thresh_employ
ment_test)
```

As we said before, now we will replace the outliers of two columns *DAYS_EMPLOYED, AMT_INCOME_TOTAL* with null values. Next, we create columns *YEARS_EMPLOYED, AGE* calculated by *DAYS_BIRTH* and *DAYS_EMPLOYED*. From the distribution chart, most of the clients making a loan have about 1-10 YOEs and are mostly from nearly 30 to 45 years old.





For categorical columns, we approached them in different ways. In terms of *CODE_GENDER*, after replacing XNA with mode, we applied label encoding to change values to binary values. But for *NAME_EDUCATION_TYPE*, we mapped it to numerical values effectively. This encoding will make it easier to leverage educational levels of clients. Moreover, we have created some new features that put some columns in one group. *FLAG_ASSET* classifies clients based on asset ownership with values ranging from 0 to 3, which indicate different combinations of owning a car and/or real estate. Besides, *INCOME_BAND* categorized individuals into income ranges and *EXP_CAR* grouped car age into 7 bins.

We created 4 new aggregated features which are *FLAG_CONTACTS, FLAG_DOCS, FLAG_ADDRESS, EXP_HOUSE*. *FLAG_CONTACTS* calculates sums of those binary columns that are related to contact methods or information. Besides, *FLAG_DOCS* aggregates all the *FLAG_DOCUMENTS_x* columns to provide a count of the number of documents that each client has submitted. This may correlate with their commitment to repaying the loan. While *FLAG_ADDRESS* sums various binary columns related to address verification, which helps in evaluating the stability and reliability of the client's residence, *EXP_HOUSE* aggregates housing-related attributes into a single column, which represents the total housing expenditure. This is crucial for assessing their overall financial burden and repayment ability.

```
train['FLAG_CONTACTS'] = train[list_col_new_flagCont].sum(axis=1)
test['FLAG_CONTACTS'] = test[list_col_new_flagCont].sum(axis=1)

train['FLAG_DOCS'] = train[list_col_new_flagDoc].sum(axis=1)
test['FLAG_DOCS'] = test[list_col_new_flagDoc].sum(axis=1)

train['FLAG_ADDRESS'] = train[list_col_new_flagAddr].sum(axis=1)
test['FLAG_ADDRESS'] = test[list_col_new_flagAddr].sum(axis=1)

train['EXP_HOUSE'] = train[list_col_new_house[1:15]].sum(axis=1)
test['EXP_HOUSE'] = test[list_col_new_house[1:15]].sum(axis=1)
```

We created a new column named *NEW_AGE_RANGE* by binning the *AGE* column in both the train set and the test set into 5 bins. By dividing borrowers into age groups, we are able to classify them more easily, as borrowers in the same age tend to have a somewhat similar financial behaviour. The *NEW_WORKING_YEAR_RANGE* column is also created by binning the *YEARS_EMPLOYED* into 4 different bins. Similar to the above column, the amount of experience of labourers often lead to quite similar habits or ways of living, including debt repaying because they have working experiences that are a little close to each other, considered by the number of years they have joined the labour force. Besides, the *DAYS_LAST_PHONE_CHANGE* is also cut into 6 categories for easier tracking and analysing. This feature didn't need to stay as a continuous variable, cutting them into ranges is enough for analysing. With borrowers who changed phones more than 5 times, we grouped them into one; and with values that are less than 0, we also labeled them as 0. Beside those changes, we kept everything else as they were. Finally, we removed values that are erroneous in the *OBS_30_CNT_SOCIAL_CIRCLE* and *OBS_60_CNT_SOCIAL_CIRCLE*.

After researching this dataset, we referred to the top 1 team in the original competition from HOME CREDIT, and they used KNN to process the training set. Therefore, we researched what KNN is in order to use it in our preprocessing. KNN is used to create one more feature named *neighbors_target_mean_500*. We experimented with different neighbors values, such as 100, 300, 1000, etc., but we found that 500 is the most reasonable. If the value of neighbors is too small, it can lead to overfitting, while if it's too large, it lacks sufficient classification capability. The mean TARGET value of the 500 closest neighbors of each row, where each neighborhood was defined by the three external sources and the credit/annuity ratio.

```
#adding the means of targets of 500 neighbors to new column
knn = KNeighborsClassifier(500, n_jobs = -1)

train_data_for_neighbors = train[['EXT_SOURCE_1','EXT_SOURCE_2','EXT_SOURCE_3','CREDIT_ANNUITY_RATIO']].fillna(0)
train_target = train['TARGET']
test_data_for_neighbors = test[['EXT_SOURCE_1','EXT_SOURCE_2','EXT_SOURCE_3','CREDIT_ANNUITY_RATIO']].fillna(0)

knn.fit(train_data_for_neighbors, train_target)

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

train['TARGET_NEIGHBORS_500_MEAN'] = [train['TARGET'].iloc[ele].mean() for ele in train_500_neighbors]
test['TARGET_NEIGHBORS_500_MEAN'] = [train['TARGET'].iloc[ele].mean() for ele in test_500_neighbors]
```

- bureau

In this table, we calculated the duration of the credit, the difference between the credit end date and actual end date, and the difference between the credit end date and the last update date providing insight into the most recent activity on the credit.

For some numerical columns relating to debt amount, some new features like debt percentage or difference between total credit amount and overdue amount, etc. are created. There are 2 binary indicators: whether the credit is currently active and whether the credit end date has passed.

We had some aggregated features: number of past loans, number of unique credit types for each customer, total loan amount (and still in debt) of each customer too. Besides, we also aggregated the sum of overdue amounts and the ratio of overdue amount to total debt for each customer.

- bureau_balance

We use all columns in the ***bureau_balance*** table to aggregate in the following steps.

- previous_application

We replaced placeholder values (365243) with NaN and created new features as ratio and difference calculations using common columns (*AMT_CREDIT, AMT_APPLICATION, AMT_ANNUITY, AMT_DOWN_PAYMENT*). Moreover, based on domain knowledge, our group calculated total interest paid over the duration of the loan and annualized interest rate.

We also made date calculations by creating *NEW_RETURN_DAY* (estimated return date), *NEW_DAYS_TERMINATION_DIFF* (difference between termination day and estimated return day), *NEW_DAYS_DUE_DIFF* (difference between the first due day and last due day), *NEW_END_DIFF* (difference between termination day and last due day).

Lastly, *NEW_CNT_PAYMENT* categorizes the number of payments into short, middle, and long-term, and *WEEKDAY_APPR_PROCESS_START* indicates if the application process started on a weekend or weekday.

- POS_CASH_balance

This code was added features to the pos dataset: *LATE_PAYMENT* flags late payments based on *SK_DPD, SK_DPD_RATIO* calculates the ratio of *SK_DPD* to *SK_DPD_DEF*, and *TOTAL_TERM* sums *CNT_INSTALMENT* and *CNT_INSTALMENT_FUTURE* to represent the total credit term.

- instalments_payments

The *DAYS_LATE_PAYMENT* feature calculates the number of days a payment is late by taking the difference between the entry payment date and the installment due date. Any negative values (early payments) are clipped to zero, focusing only on late payments. A value of amount payment to instalment ratio greater than 1 indicates an overpayment, while a value less than 1 indicates a partial payment.

- credit_card_balance

We created features in the credit dataset, including ratios like *USAGE_RATIO* (balance to credit limit) and *BALANCE_LIMIT_RATIO*. It calculates sums for different types of drawings (*AMT_DRAWING_SUM* and *CNT_DRAWING_SUM*) and payment ratios (*MIN_PAYMENT_RATIO*

and *MIN_PAYMENT_TOTAL_RATIO*). We also computed payment differences (*PAYMENT_MIN_DIFF*), interest receivables (*AMT_INTEREST_RECEIVABLE*), and the ratio of DPD to defended DPD (*SK_DPD_RATIO*).

4.2. Handle missing values of categorical columns

First, we wrote a function which has a pattern that can be applied to all tables.

```
def clean(df, group_by=None, group=True):
    result = df.copy()
    if 'Unnamed: 0' in result.columns:
        result.drop(columns=['Unnamed: 0'], inplace=True)
    if 'SK_ID_BUREAU' in result.columns:
        result = result.dropna(subset=['SK_ID_BUREAU'])
    if 'SK_ID_PREV' in result.columns:
        result = result.dropna(subset=['SK_ID_PREV'])
    if 'SK_ID_CURR' in result.columns:
        result = result.dropna(subset=['SK_ID_CURR'])

    result.replace(r'^X$', np.nan, regex=True, inplace=True)
    result.replace(r'^Unknown type of loan$', np.nan, regex=True, inplace=True)
    result.replace(r'^XNA$', np.nan, regex=True, inplace=True)

    object = result.select_dtypes(include='object').columns
    result[object] = result[object].fillna('Unknown')
```

We dropped the column *Unnamed: 0* because it is the spare column in train and test tables that only marks the ordinal numbers of each row. Next, because *SK_ID_CURR*, *SK_ID_BUREAU* and *SK_ID_PREV* are primary keys and also foreign keys in all tables, thus a row with null value in these columns provides no meaningful information and should be dropped.

Next, we investigated through the data and figured out there were different labels for unknown information in object columns, so we replaced all these values with a null value and later filled all null values with the value “Unknown”. The step is to ensure all null values and other labels for missing values are replaced with a consistent value.

After cleaning all tables, we added table names as prefixes to each table’s columns to ensure consistency when tables are joined.

4.3. Encode categorical columns

To easily fit in models, categorical columns need to be transformed to numerical type. There are several methods to encode categorical columns. The most common one is one hot encoding. We tried this method and realized it created too many extra columns and noises, which led to poor performance later when we fitted in models. Label encoding is also inappropriate because the order of levels among categories is unknown. Therefore we accessed target encoding, a method that does not create any extra column. This method computes the mean target for each category across all applications in the dataset, which is suitable for the purpose of credit risk prediction in our case. We used the built-in target encoder of library category_encoders, which already includes smoothing the target means to reduce overfitting risk when a category has very few samples. In the code block below, we used a dataframe to create target encoding values for categories, then applied the encoded values to replace categories in the relevant dataframe.

```
def encode(df, df_result):
    result = df_result.copy()
    object = df.select_dtypes(include='object').columns
    for i in object:
        # Define and fit the TargetEncoder
        encoder = ce.TargetEncoder(cols=[i])
        df[i] = encoder.fit_transform(df[i], df['train_TARGET'])
        result[i] = encoder.transform(result[i])
    return result
```

To encode categorical columns in ***bureau_balance*** table, first we joined it with ***bureau*** table to get the corresponding *SK_ID_CURR* of each *SK_ID_BUREAU* because ***bureau_balance*** table does not have *SK_ID_CURR* column. We used the inner join method to ensure the distribution of categorical data is least sensitive to null values. Then to get the target corresponding to each *SK_ID_BUREAU* through *SK_ID_CURR*, we joined with column *TARGET* in the train table. The result table was then passed into the encode function to target encoded categories, then applied the encoded values to the original ***bureau_balance*** table.

```
bureau_bal_hi = bureau_bal.join(bureau[['bureau_SK_ID_CURR', 'bureau_SK_ID_BUREAU']].
set_index('bureau_SK_ID_BUREAU'),
                                    'bureau_bal_SK_ID_BUREAU', how='inner')
bureau_bal_hi = bureau_bal_hi.join(train[['train_SK_ID_CURR', 'train_TARGET']].set_in
dex('train_SK_ID_CURR'),
                                   'bureau_SK_ID_CURR', how='inner')
bureau_bal = encode(bureau_bal_hi, bureau_bal)
```

The other tables were then processed with the same method, except that all these tables already have *SK_ID_CURR* column so it is unnecessary to join with an intermediate table. In terms of ***test*** table, to ensure consistency between the encoding used in the training set and the encoding applied to the test set, we used the precomputed mapping from the training set to encode the test set categories. This is crucial to avoid data leakage and ensure valid predictions.

4.4. Data aggregation

To be able to join all tables to a single one whose each row contains all information of an applicant, grouping and aggregating is a crucial step. After exploring, we figured out that each applicant (*SK_ID_CURR*) has a number of previous Credit Bureau applications (*SK_ID_BUREAU*) and a number of other previous applications (*SK_ID_PREV*) with different kinds reported in four tables ***previous_application***, ***credit_card_balance***, ***installments_payments*** and ***POS_CASH_balance***. First, in terms of the Credit Bureau applications, each application has tracking information each month in ***bureau_balance*** table. Therefore, we aggregated each application's information by grouping ***bureau_balance*** table by *SK_ID_BUREAU* column with functions we found appropriate after exploring column description (mean, max, sum). We tried using min too but the function did not provide much useful information, thus leading to poor performance when we trained models. After aggregation, the table was joined with ***bureau*** table, then grouped by *SK_ID_CURR* to aggregate information of each applicant's bureau credit history.

```
bureau_bal = bureau_bal.groupby('bureau_bal_SK_ID_BUREAU').agg(['mean', 'max', 'sum'])
bureau_bal.columns = [f"{col[0]}_{col[1]}" for col in bureau_bal.columns]
bureau_summary = bureau.join(bureau_bal, 'bureau_SK_ID_BUREAU')
bureau_summary.drop(columns='bureau_SK_ID_BUREAU', inplace=True)

bureau_summary = bureau_summary.groupby('bureau_SK_ID_CURR').agg(['mean', 'max', 'sum'])
bureau_summary.columns = [f"{col[0]}_{col[1]}" for col in bureau_summary.columns]
```

The same process was then applied to other four tables that have *SK_ID_PREV* column to aggregate applicants' previous application history. Each table after grouping was joined to ***train*** and ***test*** table to create the final feature dataset used for training and testing.

4.5. Handle missing values of final train and test dataset

Null values in both encoded categorical columns and numerical columns were filled with mean method. Null values in test dataset were filled with mean of train dataset instead of mean of itself to avoid data leakage, where information from the test dataset influences the training process, leading to overly optimistic performance metrics.

```
train = train.fillna(train.mean())
test = test.fillna(train.drop(columns='train_TARGET').mean())
```

5. Running Model

```
train = pd.read_csv('train_6.csv')
test = pd.read_csv('test_6.csv')

X = train.drop(columns=['train_SK_ID_CURR', 'train_TARGET'])
y = train['train_TARGET']
X_test = test.drop(columns=['train_SK_ID_CURR'])

# # Standardize the features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
X_test_scaled = scaler.transform(X_test)

# Split the training data for validation
X_train, X_val, y_train, y_val = train_test_split(X_scaled, y, test_size=0.1, random_state=42)
```

First of all, we read the train and test file, drop unnecessary columns and scale both the ***train*** and ***test*** dataset by using StandardScaler. This way, the data will be scaling based on standardization, making the training process more easily, more accurate and faster. After that, we split it into the train set and validation set. The validation set accounts for 10% of the ***train*** dataset.

```python
# Create Logistic Regression model
lr = LogisticRegression(max_iter=5000, random_state=42)

# Define the parameter grid for RandomizedSearchCV
param_dist = {
    'penalty': ['l2'],
    'C': np.logspace(-2, 2, 6),
    'solver': ['lbfgs', 'newton-cg'],
}

# Setup RandomizedSearchCV for randomized hyperparameter search with cross-validation
random_search = RandomizedSearchCV(
    estimator=lr,
    param_distributions=param_dist,
    scoring='roc_auc',
    cv = 3,
    verbose=2,
    n_jobs= 3,
    random_state=42
)

# Fit the model with randomized search
random_search.fit(X_train, y_train)

# Get the best model from RandomizedSearchCV
best_model = random_search.best_estimator_
```

We used LogisticRegression to train our model, and RandomizedSearchCV to cross-validate it to choose the best model for the dataset. Ridge regularization was used, together with two solvers: 'lbfgs' and 'newton-cg' to regularize the model in 3 cross-validations. After fitting and cross-validating, we chose the best model by choosing the attribute *best_estimator_*.

```python
# Predict using the best estimator from RandomizedSearchCV
y_pred = best_model.predict(X_test_scaled)
y_pred_proba = best_model.predict_proba(X_test_scaled)[:, 1]

result = pd.DataFrame({"SK_ID_CURR": test['train_SK_ID_CURR'], "TARGET": y_pred_proba})
result.to_csv("submission.csv", index=False)
```

The best model was then used to predict the probability for the ***test*** dataset that has been standardized. The predictions were put into a dataframe with their ID to create the submission file.

DISTRIBUTION TABLE

STT	Name	Contribution Content	Contribution Percentage
1	Phạm Ngọc Linh	- Leader - Cleaning data - Training model - Writing report - Designing slides	25%
2	Nguyễn Việt Hằng	- Cleaning data - Training model - Writing report - Designing slides	25%
3	Nguyễn Bảo Chi	- Training model - Writing report - Presenting	25%
4	Phạm Mai Linh	- Training model - Writing report - Presenting	25%
Total			100%