Project Report

Home Credit Default Risk Analysis

Submitted by

Jeet Hathiwala

Souham Sengupta

Annanya Bansal

Fathima Nafeesa VY

Gaurav Kumar

Bibek Kumar Agrawal

IN PARTIAL FULFILMENT OF THE REQUIREMENT OF

Executive Programme In Applied Data Science Using

Machine Learning & Artificial Intelligence

3rd August 2024

Under the Guidance of Prof. Niladri Chatterjee

Professor of Artificial Intelligence

Department of Mathematics

IIT Delhi

Acknowledgement

We would like to thank, Prof. Niladri Chatterjee, Professor of Computer Science & Statistics in the Department of Mathematics, IIT Delhi who gave us the opportunity to develop a project for the partial fulfilment of the requirement of, Executive Programme in Applied Data Science Using Machine Learning & Artificial Intelligence, conducted by IIT, Delhi.

We thank all the team members, who contributed their might for the project.

Preamble

The absence of a credit history might mean a lot of things, including young age or a preference for cash. Without traditional data, someone with little to no credit history is likely to be denied. Consumer finance providers must accurately determine which clients can repay a loan and which cannot and data is key. If data science could help better predict one’s repayment capabilities, loans might become more accessible to those who may benefit from them the most.

Currently, consumer finance providers use various statistical and machine learning methods to predict loan risk. These models are generally called scorecards. In the real world, clients' behaviours change constantly, so every scorecard must be updated regularly, which takes time. The scorecard's stability in the future is critical, as a sudden drop in performance means that loans will be issued to worse clients on average. The core of the issue is that loan providers aren't able to spot potential problems any sooner than the first due dates of those loans are observable. Given the time it takes to redevelop, validate, and implement the scorecard, stability is highly desirable. There is a trade-off between the stability of the model and its performance, and a balance must be reached before deployment.

Founded in 1997, competition host Home Credit is an international consumer finance provider focusing on responsible lending primarily to people with little or no credit history. Home Credit broadens financial inclusion for the unbanked population by creating a positive and safe borrowing experience. We previously ran a competition with Kaggle that you can see here.

Your work in helping to assess potential clients' default risks will enable consumer finance providers to accept more loan applications. This may improve the lives of people who have historically been denied due to lack of credit history.

Table of contents

1. Introduction

Many individuals face difficulties in obtaining loans due to insufficient or non-existent credit histories, leaving them vulnerable to exploitation by untrustworthy lenders. Home Credit is committed to enhancing financial inclusion for this unbanked population by offering a secure and positive borrowing experience. To ensure that these underserved individuals have access to fair loan opportunities, Home Credit leverages a range of alternative data sources, such as telecommunications and transactional information, to assess their clients' repayment capabilities.

Currently, Home Credit employs various statistical and machine learning techniques to make these predictions. We are trying to help the community to help maximise the potential of their data. The goal is to ensure that clients who are capable of repayment are not unfairly rejected and that loans are provided with appropriate principal amounts, maturities, and repayment schedules, enabling clients to succeed financially.

2. Background

2.1. Importance of credit risk analysis

Credit risk analysis is crucial for financial institutions as it helps assess the likelihood of a borrower defaulting on a loan. This analysis is essential for maintaining the stability and profitability of lenders, as it informs decisions on loan approvals, interest rates, and terms.

Accurate credit risk assessment not only protects the financial institution from potential losses but also ensures that borrowers are offered fair and appropriate financial products.

This is particularly important for unbanked or underbanked populations, who may lack traditional credit histories and thus require innovative approaches to evaluating their creditworthiness. Effective credit risk analysis fosters financial inclusion by enabling responsible lending practices that empower borrowers to build credit and achieve financial stability.

2.2. Role of machine learning in credit risk analysis

Machine learning plays a transformative role in credit risk analysis by enhancing the accuracy and efficiency of assessing borrowers' creditworthiness. Unlike traditional methods that rely heavily on historical financial data, machine learning algorithms can analyse a wide range of data sources, including alternative data such as social media activity, utility payments, and telecommunication records. This capability allows for a more comprehensive and nuanced understanding of a borrower's risk profile.

By leveraging vast amounts of data and learning from patterns, machine learning models can identify subtle correlations and trends that may be overlooked by conventional approaches. This leads to more accurate predictions of default risk, enabling lenders to make better-informed decisions about loan approvals, pricing, and terms. Additionally, these models can be

continuously updated and refined as new data becomes available, ensuring that the risk assessment remains current and relevant. Ultimately, the application of machine learning in credit risk analysis supports fairer lending practices and broader financial inclusion.

3. Objective

The objective of this project is to enhance the accuracy of credit risk assessment for underserved populations with insufficient or non-existent credit histories. By leveraging a diverse set of alternative data, such as telecommunications and transactional information, and utilising advanced statistical and machine learning techniques, we aim to refine our predictive models. This will ensure that capable borrowers are not unfairly rejected and that loans are structured with appropriate terms, fostering financial inclusion and empowering clients to achieve financial stability.

4. Dataset

4.1. Description

In this project, we are trying to predict client defaults using both internal and external data available for each client. The evaluation of the predictions is based on a custom metric that not only measures the Area Under the Curve (AUC) but also assesses the stability of the prediction model across the entire range of the test dataset.

4.1.1. Table description

This dataset contains a large number of tables as a result of utilising diverse data sources and the varying levels of data aggregation used while preparing the dataset.

- `base tables (train_base.csv)`: Base tables store the basic information about the observation and `case_id`. This is a unique identification of every observation and we need to use it to join the other tables to base tables.
- `applprev_1`: This contains information about the applicant's personal details, including marital status, number of children, education level, monthly annuity amount, and other relevant attributes.
- `tax_registry_a_1`: This contains information about the tax deductions amount tracked by the government registry, unique case id and date of tax deduction record.
- `tax_registry_b_1`: This contains information about the unique case id, tax deductions amount tracked by the government registry and the name of the employer.
- `tax_registry_c_1`: This contains information about the case id, name of the employer, tax deductions amount for credit bureau payments and date when the tax deduction is processed.
- `credit_bureau_a_1`: This contains information about interest rate for both active and closed contracts, classification of active and closed contract, credit limit for active loan etc.
- `credit_bureau_b_1`: This contains information about the credit amount of the active contract provided by the credit bureau, classification of active contract, end date of active contract, credit limit for active loan etc.
- `deposit_1`: This contains information about the deposit amount, contract end date and deposit amount opening date.
- `person_1`: The contains information about the applicant and has information related to his date of birth, number of children he/she has, zip code of a contact person's address etc.
- `debitcard_1`: This file contains information related to the last 180 days (6 months) average balance of the applicant, last 180 days turnover and also last 30 days total turnover.
- `applprev_2`: This file contains information about the card blocking reason, contact type information, card status of the previous credit account.
- `person_2`: The file contains information about the applicant’s district, employment timeline, zip code, name of the employer etc.

- `credit_bureau_a_2`: The file contains information about both collateral valuation type (active & closed contract), type of collateral that was used as a guarantee for both closed and active contract and days past due of the payment for terminated contract according to the credit bureau.
- `credit_bureau_b_2`: The file contains information related to payment date for an active contract according to credit bureau, value of past due payment for active contract and active contract that has overdue payments.

4.1.2. Special columns

- `case_id`: This is the unique identifier for each credit case. We'll need this ID to join relevant tables to the base table.
- `date_decision`: This refers to the date when a decision was made regarding the approval of the loan.
- `WEEK_NUM`: This is the week number used for aggregation. In the test sample, `WEEK_NUM` continues sequentially from the last training value of `WEEK_NUM`.
- `MONTH`: This column represents the month and is intended for aggregation purposes.
- `target`: This is the target value, determined after a certain period based on whether or not the client defaulted on the specific credit case (loan).
- `num_group1`: This is an indexing column used for the historical records of `case_id` in both `depth=1` and `depth=2` tables.
- `num_group2`: This is the second indexing column for `depth=2` tables' historical records of `case_id`. The order of `num_group1` and `num_group2` is important and will be clarified in feature definitions.

4.1.3. Features

4.1.3.1. Depth values

- `depth 0`: These are static features directly tied to a specific `case_id`.
- `depth 1`: Each case_id has an associated historical record, indexed by `num_group1`.
- `depth 2`: Each case_id has an associated historical record, indexed by both `num_group1` and `num_group2`.

4.1.3.2. Transforms

- `P`: Days past due
- `M`: Masking categories
- `A`: Amount
- `D`: Date
- `T`: Unspecified Transform
- `L`: Unspecified Transform

4.2. Loading the dataset

The dataset has been made available to us in two different formats, `csv` and `parquet`. For our data loading purposes, we have chosen to use the `parquet` format in conjunction with `polars` instead of `pandas`. This is because `parquet` files are much smaller than their `csv` counterparts, and `polars` is much faster than `pandas` and is a lot more memory efficient as well.

We'll merge the different files described in an earlier section using joins. We'll be joining these files using the `case_id` column as the foreign key.

When joining these files, we also need to specify aggregation functions - which describes how the data is to be treated if we have more than an element with more than one foreign key, since we don't want multiple rows with the same `case_id`. We've chosen to go with aggregation using the median function.

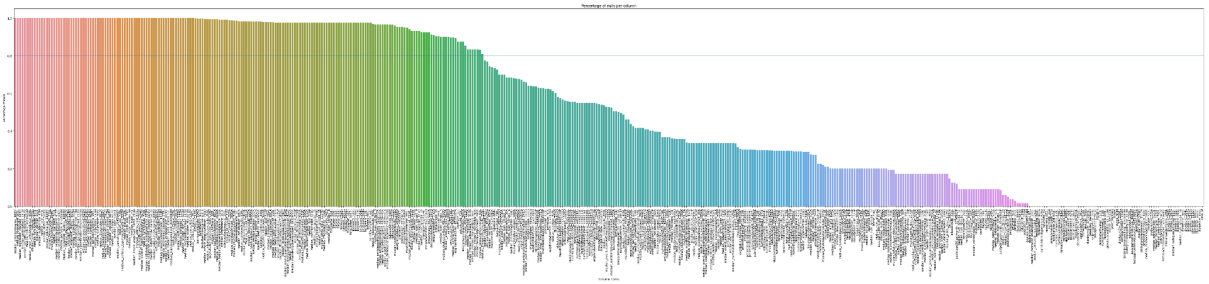
```
def get_aggregates(df, aggr_fns=['median']):
    aggrs = Aggregates.num_aggr(df, aggr_fns) + \
            Aggregates.date_aggr(df, aggr_fns) + \
            Aggregates.str_aggr(df, aggr_fns) + \
            Aggregates.other_aggr(df, aggr_fns) + \
            Aggregates.count_aggr(df, aggr_fns)

    return aggrs
```

We also know the columns are in a variety of formats, which has been specified earlier by the suffix attached to the column name. We'll simply cast the column to an appropriate datatype, which will make future operations on the column easier to do.

```
def set_dtypes(df):
    for col in df.columns:
        if col in ['case_id', 'WEEK_NUM', 'num_group1', 'num_group2']:
            df = df.with_columns(pl.col(col).cast(pl.Int64))
        elif col in ['date_decision']:
            df = df.with_columns(pl.col(col).cast(pl.Date))
        elif col[-1] in ('P', 'A'):
            df = df.with_columns(pl.col(col).cast(pl.Float64))
        elif col[-1] in ('M',):
            df = df.with_columns(pl.col(col).cast(pl.String))
        elif col[-1] in ('D',):
            df = df.with_columns(pl.col(col).cast(pl.Date))
    return df
```

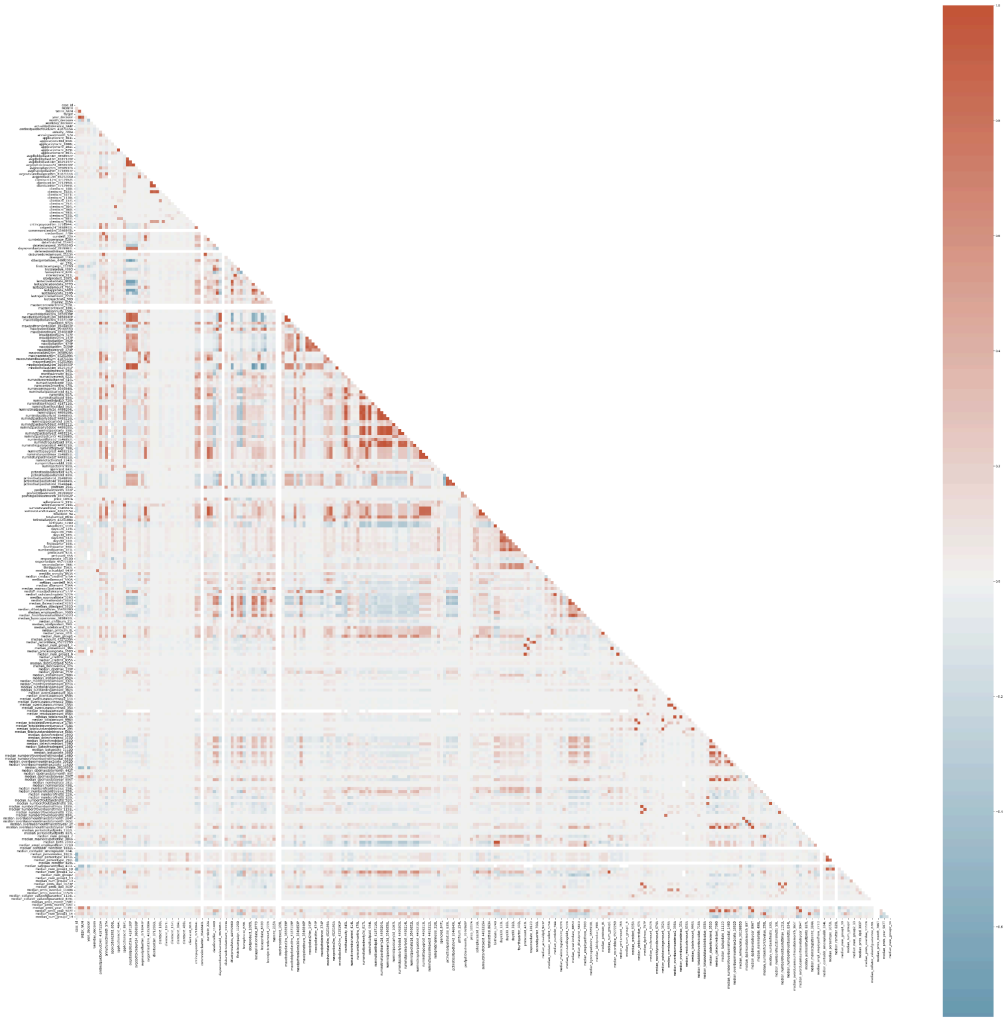
5. EDA

5.1. Columns with null values



As we can see above, a lot of columns have simply null values, and since they don't provide much information, we should be good to drop these entirely.

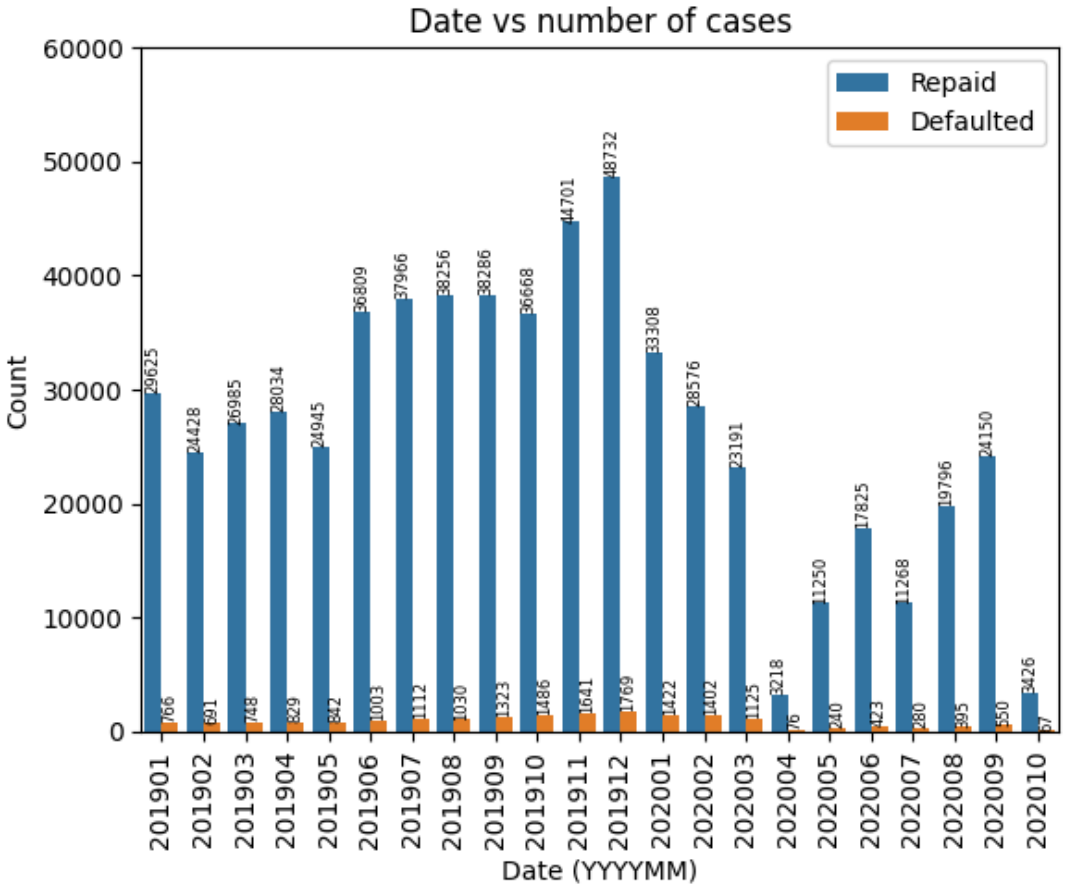
5.2. Correlation matrix



As we can see above, there is a significant negative correlation of target with `pctinstlsallpaidearl3d_427L`, `numinstpaidearly3dest_4493216L`, `numinstmatpaidtearly2d_4499204L`, `numinstpaidearly5dobd_4499205L`, `numinstpaidearlyest_4493214L`, `numinstpaidearly3d_3546850L`, `numinstlallpaidearly3d_817L`, and a positive correlation with `avgmaxdpdlast9m_3716943P`, `numinstlswithdpd10_728L`, `pctinstlsallpaidlat10d_839L`, `pctinstlsallpaidlate6d_3546844L`, `pctinstlsallpaidlate4d_3546849L` and `pctinstlsallpaidlate1d_3546856L`.

This is expected behaviour. If the user has paid most of their instalments early, then there's a higher chance that they are good with their credit and won't default - hence the negative correlation. Au contraire, if all their instalments are paid late, there's a higher chance they are bad with their credit and are more likely to default - hence the positive correlation.

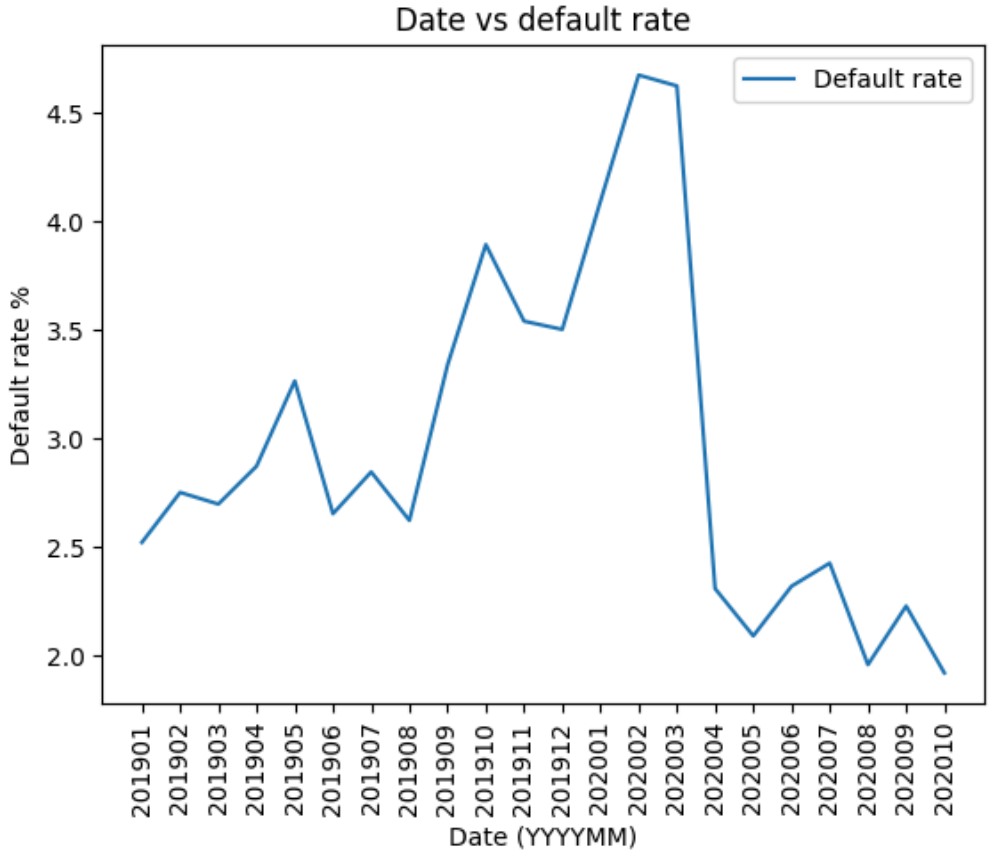
5.3. Plot of values over time



Worth noting is that the data that we have is right around covid - the loan defaults and repayments drop drastically ever since December 2019, and dip to almost zero during the first wave.

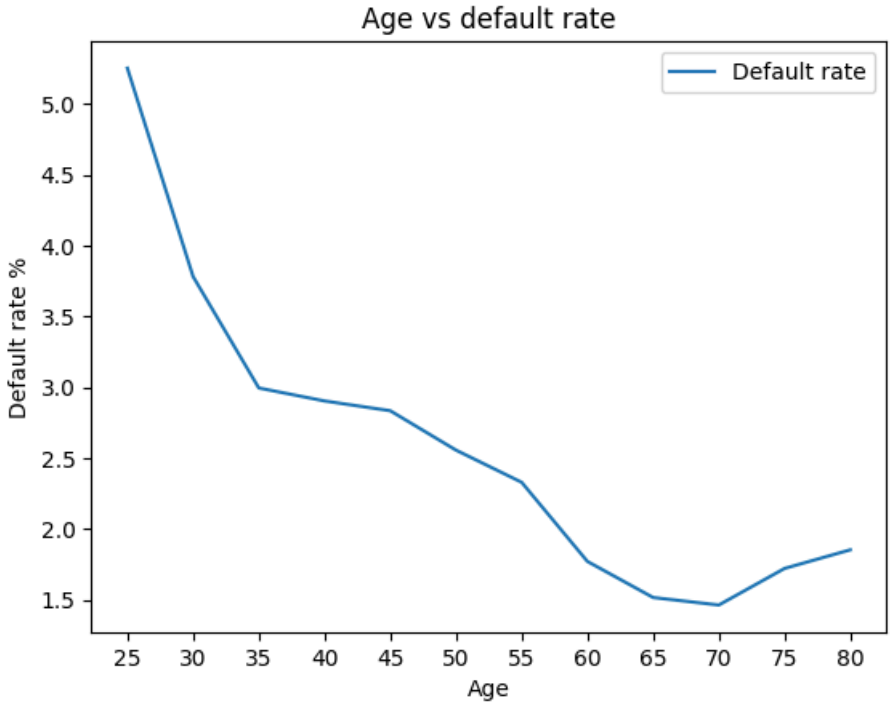
```
{'201912': {'defaulted': 1769, 'repaid': 48732},
 '202004': {'defaulted': 76, 'repaid': 3218}}
```

5.4. Rate of defaulting over time



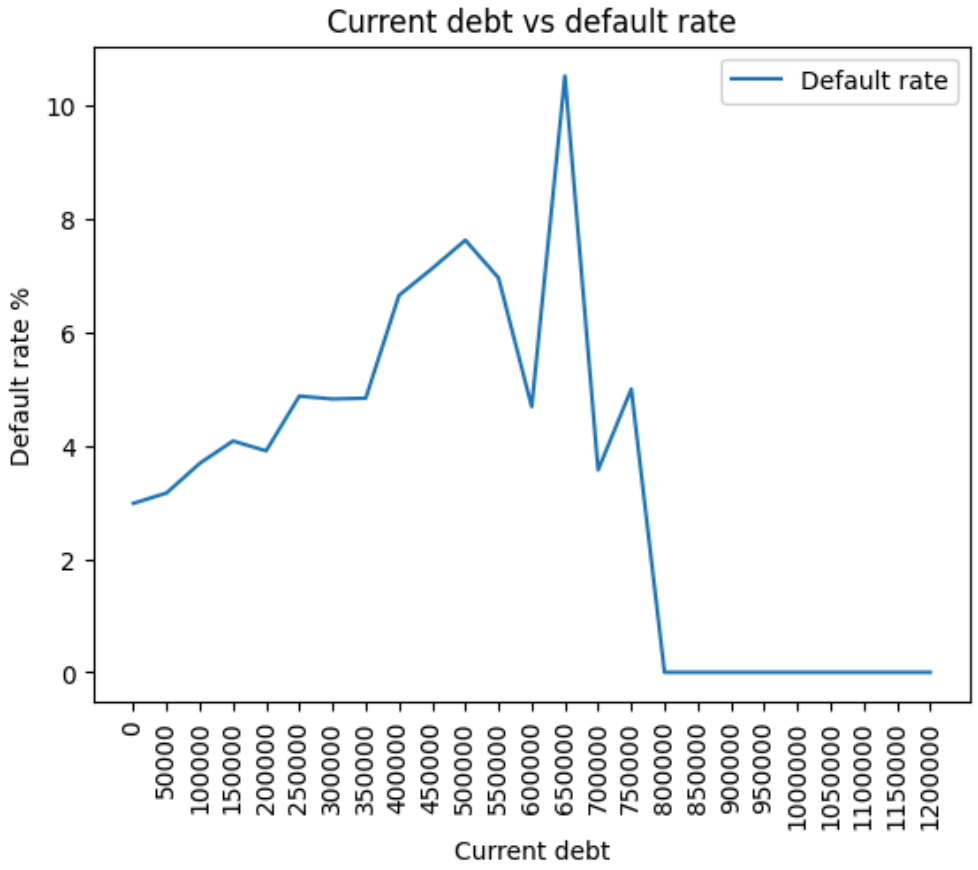
Above, we can see that the default rate slowly increases from August of 2019 onwards, all the way to April of 2020, which is when it stops. This is probably due to the banks being very selective to whom they give loans during tougher times due to Covid.

5.5. Rate of defaulting by age



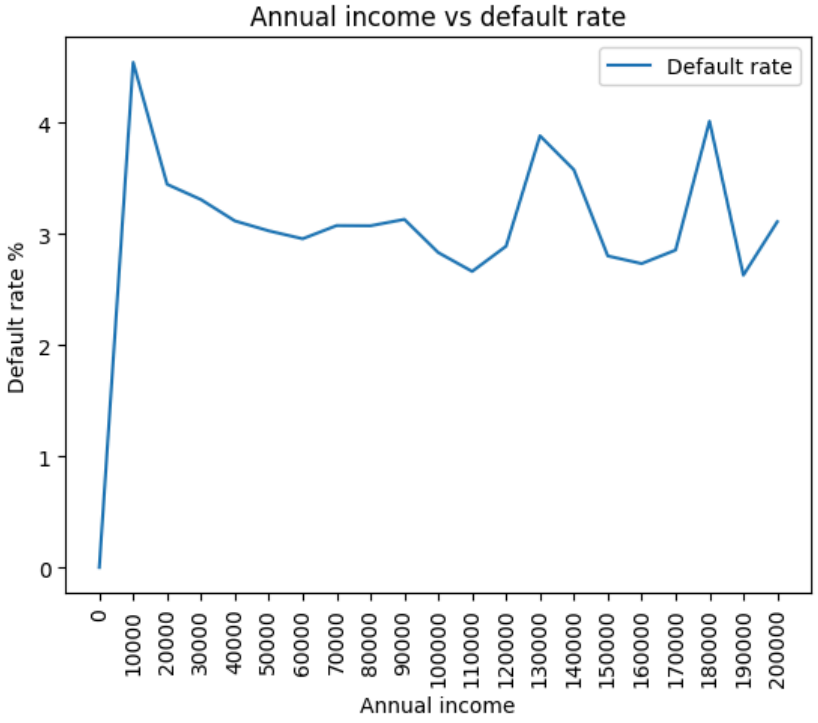
Above, we can see that the likelihood of defaulting decreases with an increase in age. This is likely due to the fact that someone older usually has a longer career behind them, and is less likely to default due to it.

5.6. Rate of defaulting by debt



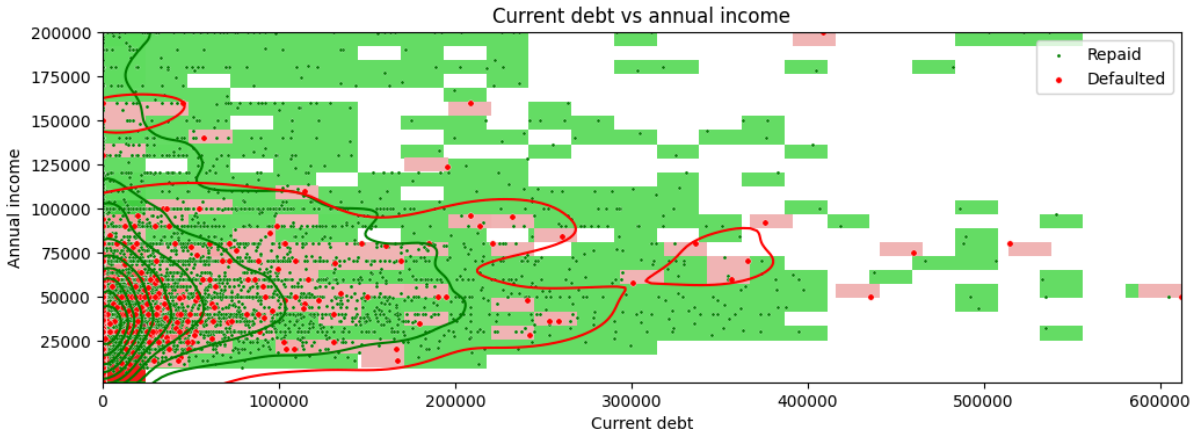
Above, we can see that the likelihood of defaulting increases as the amount of debt increases. Fairly standard observation. The rate of defaulting is very low in the case of very high debts. This is probably due to the banks not approving loan requests for people with higher debts.

5.7. Rate of defaulting by income



As we can see above, the rate of defaulting is higher among low income groups, and then stabilises around an income of $40000. Strangely, it increases around an income level of $120,000. This could be due to higher income individuals opting for more expensive properties and larger loan amounts.

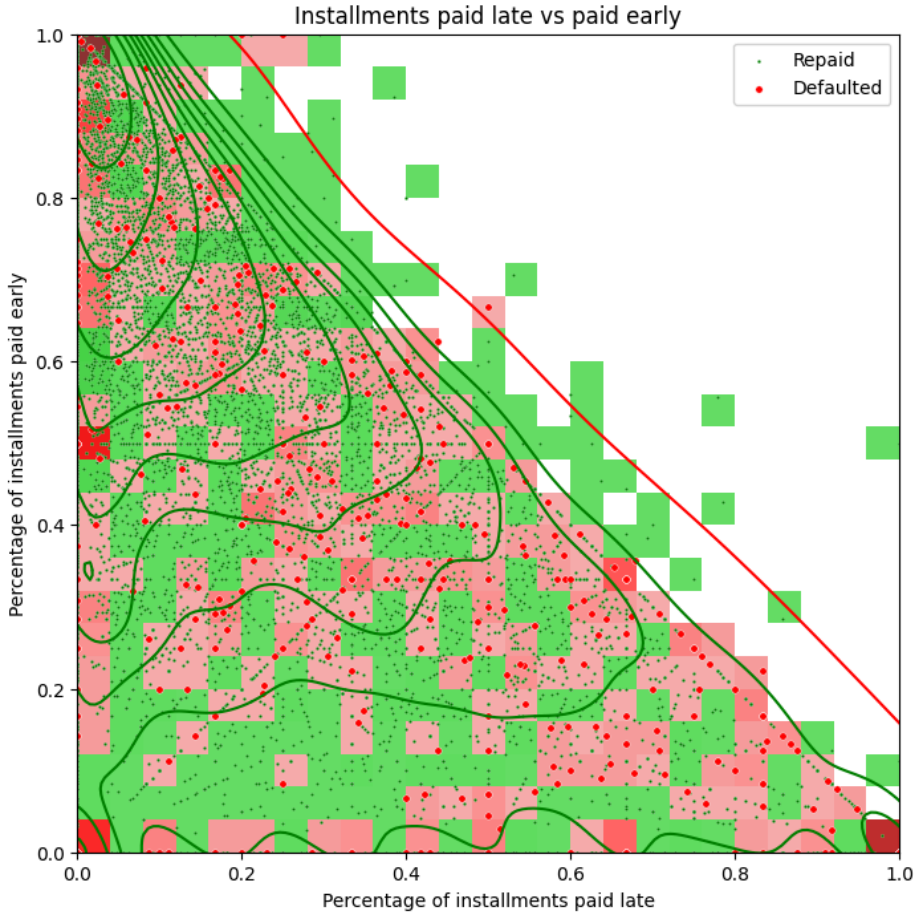
5.8. Plot of income and debt



Above, we can see that there is a higher likelihood of a person defaulting if they're in a lower income group with a potentially higher debt. High income individuals have a lower chance of defaulting unless their debt is also equally high.

The higher the income, the lesser the chance of defaulting. The lower income groups with higher debts are at a very high chance of defaulting as compared to the rest of the sample.

5.9. Plot of percentage of early and late instalments



Above, we can see that the highest number of non-defaulters are centred around the top left quadrant - most number of instalments paid early and the least number of instalments paid late. That said, the defaulters are scattered throughout the graph and there's no single point where they gravitate.

6. Model

6.1. Preprocessing and feature engineering

6.1.1. Feature engineering

6.1.1.1. Dates

Dates and times are rich sources of information that can be used to train machine learning models. However, these datetime variables do require some feature engineering to turn them into numerical data.

For our feature engineering, we're calculating how many days it has been between the current date column and the date of the decision. This would give us insight on how far along inquires have happened for the customer, and how long each inquiry has lasted, which would be valuable insight in letting us know what is going on

```
def process_dates(df):
    processed_cols = []
    for col in df.columns:
        if col[-1] in ('D',):
            processed_cols.append(col)
            df = df.with_columns(pl.col(col) - pl.col('date_decision'))
            df = df.with_columns(pl.col(col).dt.total_days())

    df = df.drop('date_decision')
    return df
```

6.1.1.2. Categorical features

Categorical features are usually of the `String` datatype, and we can't fit those columns into the model either without preprocessing them. To convert these into numerical form, we need to use an encoding such as a one hot encoding or label encoding. We're preferring a one hot encoding using `sklearn`'s `OneHotEncoder` since label encoding can imply false relationships between the categories.

6.1.2. Preprocessing steps

6.1.2.1. Dropping diverse columns

We would like to drop columns which have too many unique values. We're dropping off string columns which have 200 or more unique values. While it could prove to be informational, at such a large variety of unique values, we'd be running into memory issues trying to represent the column as a one hot encoded version of itself.

```
def drop_string_cols_with_numerous_values(df, is_test):
    if is_test:
        df = df.drop(list(DataFrameOps.dropped_train_cols))
        return df

    DataFrameOps.dropped_train_cols = {}

    for col in df.columns:
        if col in ['target', 'case_id', 'WEEK_NUM']:
            continue

        if df[col].dtype != pl.String:
            continue

        freq = df[col].n_unique()
        if freq == 1 or freq > CFG.COL_MAX_UNIQUE_VALUES:
            DataFrameOps.dropped_train_cols[col] = freq
    df.drop(list(DataFrameOps.dropped_train_cols))

    return df
```

We could go into these features on a case-by-case basis and see if there could be a common factor using which we could limit the number of unique values.

6.1.2.2. Dropping columns with too many null values

We would also like to drop columns that have a large ratio of incomplete data. We could (and do) handle missing values for columns in which it's feasible, but for a column that has more than 80% of rows which are null or empty, we have chosen to drop that column itself.

```
def drop_nullish_columns(df, null_stats_df):
    original_df_shape = df.shape

    for col in [colname for colname in null_stats_df[null_stats_df['null_percentage'] >
CFG.NULL_CUTOFF]['name']]:
        df = df.drop(col)

    updated_df_shape = df.shape

    return df, original_df_shape, updated_df_shape
```

6.1.2.3. Handling missing values

For columns with null values, we chose to handle missing values with `sklearn`'s `SimpleImputer`, which will replace all occurrences of `np.nan` with the mean of the column. This strategy has the potential of affecting the dataset by skewing the values in the dataset toward the dataset mean. We could instead also choose to go and label any missing values with a particular constant (such as `0`), which would make more sense for categorical columns in which a certain value doesn't imply a magnitude.

6.1.2.4. Scaling and normalising the dataset

We need to normalise and scale the values in the dataset as it can help to balance the impact of all variables and can help to improve the performance of the algorithm. We're scaling the dataset using `sklearn`'s `StandardScaler`.

```
cat_cols = [col for col in df.columns if df[col].dtype == pl.String]
num_cols = [col for col in df.columns if df[col].dtype != pl.String]

col_transformer = ColumnTransformer(transformers=[
    ('imputer', SimpleImputer(), num_cols),
    ('cat_cols', OneHotEncoder(), cat_cols),
    ('num_cols', StandardScaler(), num_cols),
], n_jobs=-1).fit(df)

X = col_transformer.transform(df)
```

6.2. The model

XGBoost (eXtreme gradient boosting), is an advanced implementation of the gradient boosting algorithm which creates decision tree ensembles. We used it for its performance, robustness and flexibility. It builds models one after another, with each new model fixing the mistakes of the previous ones, leading to better accuracy and performance compared to models like random forest that build trees independently. It uses regularisation to prevent overfitting and works well with finding complex interactions between features in structured data like ours.

```
from xgboost import XGBClassifier

eval_metrics = ['error', 'logloss', 'auc', 'pre']

model = XGBClassifier(eval_metric=eval_metrics, early_stopping_rounds=15)
```

It delivers good baseline results with minimal setup, unlike neural networks that typically need extensive model configuration and hyperparameter tuning to perform effectively.

6.3. Training

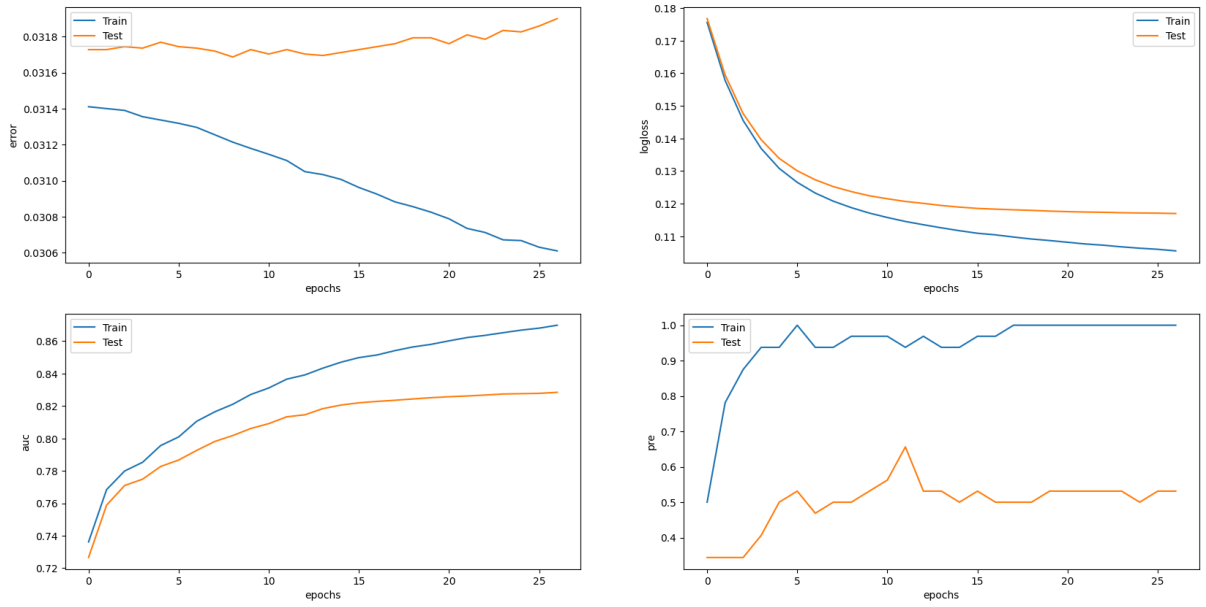
In training, we're using our train data but with 20% of it set aside as the validation set - which won't be used to train the data, but will instead be used to observe the performance of the model, since it is previously unseen data.

```
model.fit(_X_train, _y_train,
          eval_set=[(_X_train, _y_train), (_X_val, _y_val)])
```

We're also using an early stopping callback via `early_stopping_rounds` which will stop training when the model stops improving on our validation scores. This is so that we don't overfit our train data.

7. Results

Overall, we're getting a validation AUC score of around 0.8, and a validation precision of around 0.55, which is about par with the other notebooks in the competition. The competition test metric has a slightly different, undisclosed error metric, and we can’t confirm our model’s performance on the same.



We keep running out of memory quite frequently and are only training on a subset of the entire data because of it, so we could probably improve with a bit more feature engineering and other alternatives of streaming data instead of holding it in memory.

8. Challenges and limitations

One of the major challenges that we faced with this dataset was that it was too large - both in terms of the number of features we had, and the number of rows. This proved to be a challenge when loading the dataset and fitting our imputers and scalers, as the model would often go out of memory. As a result, we’ve trained the model on a sample of our actual dataset. Choosing anything higher than 50% of the rows would result in our kernel running out of memory and crashing.

Another problem was that we had certain columns with too many null values, and we could’ve done a bit more EDA to understand why that was the case, and treat them on a case-by-case basis, instead of bulk removing features.

A third challenge we had was that we had too many columns with a variety of string data, and converting it all using one hot encoding would’ve increased our already high memory usage. Similar to the null values, we could’ve done more EDA to understand what the individual features were, and treated them on a case-by-case basis instead of dropping them all off if they had too many unique values.

9. Future steps

Building from our core challenges, a future improvement would be to stream data instead of holding it all in memory. This would allow us to train on the entirety of our data and would result in our kernel not running out of memory quite as easily.

We could also do a bit more EDA to understand what each of our columns do individually, and treat them on a case-by-case basis instead of treating them en masse.

Keeping the above in mind, we could do a bit more feature engineering for the columns on a case by case basis as well instead of just engineering them in bulk.

We could also use different types of models and compare their performance among each other, and tune them using different hyperparameters using something like GridSearchCV to see which one fares best for us. In the real world, depending upon the machine on which we deploy and how much latency we can afford to have with our data, we might be willing to sacrifice sheer accuracy for a smaller and simpler, but faster model.