Home Credit Risk Modeling: Insights and Technical Strategies

For more personal insights, please refer to the threads by Nhut, Nhat, and myself, as well as Quan's comment.

Introduction

This is our first `Kaggle` competition, and facing a challenging, complex problem like the Home Credit - Credit Risk Model Stability is a big challenge for us. However, we believe that we have done our best in applying machine learning to solve this problem. We will share some perspectives on our attempt to tackle this task.
Data in this domain is highly heterogeneous, collected over different time frames and from various sources that may change during the data collection process. Additionally, the vast amount of data requires techniques that can adapt to these conditions. Developing a proper target variable is also a tricky process, requiring deep domain knowledge and refined business analysis skills. Furthermore, the evaluation metric used in this competition, the gini stability metric, is quite new, posing an additional challenge in understanding and improving our model. We want to commend Home Credit and Kaggle for providing such a great dataset, which was leak-free and very amenable to machine learning techniques. Based on what is known about home credit and insights from reference notebooks, two essential elements for building a good model for this competition are:

1. A good process of Feature Engineering.
2. A diversity set of base algorithms.
3. Good use of the new provided metrics.

We were able to utilize four main sources of feature diversity, along with a few minor additional ones.

Data and Feature Engineering

EDA (Nhat)

With such complex and huge amount of data, the first things I focused on is to understand the data types of the features based on their suffixes. Said notation can be found on the data section. With a intuitive notebook I got data types contributions based on their suffix, which are: `P` : $100%$ `Float64` `M` : $100%$ `String` `A` : $100%$ `Float64` `D` : $100%$ `Date` `T` : $77%$ `Float64` , $23%$ `String` `L` : $76.4%$ `Float64` , $16.6%$ `String` , $7.0%$ `Boolean` This classification helped me in reformatting the data types appropriately and ensuring consistency across similar groups of features. I came around with a notebook, which provide a clear and adequate EDA about the data, I realized that the `target` class is highly imbalanced, moreover, there is also a large amount of records that is NaN across all columns, so we had to do something to tackle those issues.

Feature Selection (Nhat)

Reference to the baseline notebook and some other public notebooks, with said reason, I started with reformatting the data types based on their notation of similar groups of transformation. to get a more intuitive structure. For instance, I had a function

that converted these columns (`case_id` , `WEEK_NUM` , `num_group1` , `num_group2`) to `int64` . Columns with the suffix `P` and `A` were converted to `float64` , those with the suffix `M` were converted to `string` , and those with the suffix `D` and the column `date_decision` were converted to `Date` type. I did not utilize features with `T` and `L` suffixes due to their mixed types and lower utility in initial experiments.

```
  def set_table_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"] or col[-1] in ("D"):
                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))
        return df
```

I realized that large datasets can consume significant amounts of memory, leading to slower performance and potential memory errors. Then I found handy function `reduce_mem_usage` , which significantly reduced the memory usage of the DataFrame by adjusting the data types of its columns from this: HomeCredit: Population Stability Index.

```
def reduce_mem_usage(df, df_name, verbose=True):
"""
This function changes pandas numerical dtypes (reduces bit size if possible)
to reduce memory usage

:param df: pandas DataFrame
:param df_name: str, name of DataFrame
:param verbose: bool, if True prints out message of how much memory usage was reduced

:return:  pandas DataFrame
"""
numerics = ['int16', 'int32', 'int64', 'float16', 'float32', 'float64']
start_mem = df.memory_usage().sum() / 1024**2
for col in df.columns:
    col_type = df[col].dtypes
    if col_type in numerics:
        c_min = df[col].min()
        c_max = df[col].max()
        if str(col_type)[:3] == 'int':
            if c_min > np.iinfo(np.int8).min and c_max < np.iinfo(np.int8).max:
                df[col] = df[col].astype(np.int16)
            elif c_min > np.iinfo(np.int16).min and c_max < np.iinfo(np.int16).max:
                df[col] = df[col].astype(np.int32)
            elif c_min > np.iinfo(np.int32).min and c_max < np.iinfo(np.int32).max:
                df[col] = df[col].astype(np.int64)
            elif c_min > np.iinfo(np.int64).min and c_max < np.iinfo(np.int64).max:
                df[col] = df[col].astype(np.int64)
        else:
```

```
            if c_min > np.finfo(np.float16).min and c_max < np.finfo(np.float16).max:
                df[col] = df[col].astype(np.float32)
            elif c_min > np.finfo(np.float32).min and c_max <
np.finfo(np.float32).max:
                df[col] = df[col].astype(np.float64)
            else:
                df[col] = df[col].astype(np.float64)
# calculate memory after reduction
end_mem = df.memory_usage().sum() / 1024**2
if verbose:
    # reduced memory usage in percent
    diff_pst = 100 * (start_mem - end_mem) / start_mem
    msg = f'{df_name} mem. usage decreased to {end_mem:5.2f} Mb ({diff_pst:.1f}%
reduction)'
    print(msg)
return df
```

After that, I handled missing values and reduced columns based on correlation. First, I noticed that there were many missing values. Initially, I considered dropping instances with a proportion of missing values below a threshold of `0.8`. Then, I experimented with increasing the threshold to various values in the range of `0.8` to `0.99`. I discovered that a threshold of `0.95` resulted in a gain in the public score. I also tried replacing missing values with the `mean` and `median`, but this did not seem to improve the score.

Feature Engineering (Nhut)

Continuing Tang Nhat's changes, I experimented with adjusting the value of `freq` in `filter_cols`, successively trying `freq > 100`, `freq > 150`, and `freq > 200`, but overall, there wasn't much variation.

Feature Engineering (Quan)

I randomly dropped some columns as well as those I considered useless. Most of these actions led to a decrease in the public score by **0.001** to **0.006**. However, when the final scores were announced, the private score mostly remained unchanged, and for one group of columns, the private score even increased by **0.003**.

```
def to_pandas(df_data, cat_cols=None):
    df_data = df_data.to_pandas()
    if cat_cols is None:
        # Public: 0.59180   Private: 0.51821
        dropcol = ["deferredmnthsnum_166L", "max_empladdr_zipcode_114M",
                   "last_empladdr_zipcode_114M", "opencred_647L",
                   "max_type_25L", "clientscnt_257L",
                   "last_collaterals_typeofguarante_359M"]
        df_data = df_data.drop(columns = dropcol)
        cat_cols = list(df_data.select_dtypes("object").columns)
    df_data[cat_cols] = df_data[cat_cols].astype("category")
    return df_data, cat_cols
```

Additionally, in the original notebook and most public notebooks, the `WEEK_NUM` column was not used for training. I tried using it and achieved an increase of approximately **0.001** in the public score.

Feature Engineering (Dang)

Reference to the baseline notebook, i changed the isnull in filter_cols(df) from 0.8 to 0.95 but it didn't perform well.

def filter_cols(df): for col in df.columns: if col not in ["target", "case_id", "WEEK_NUM"]: isnull = df[col].is_null().mean() if isnull > 0.95: df = df.drop(col)

Hyperparameter Tuning (Dang)

As the competition entered its final phases, there are many great notebooks that help us to improve our model, particularly the baseline notebook. We try to tune the parameters according to this one.

params = { "boosting_type": "gbdt", "objective": "binary", "metric": "auc", "max_depth": 10,
"learning_rate": 0.05, "n_estimators": 2000,
"colsample_bytree": 0.8, "colsample_bynode": 0.8, "verbose": -1, "random_state": 42, "reg_alpha": 0.1, "reg_lambda": 10, "extra_trees":True, 'num_leaves':64, "device": device, "verbose": -1, }

Using those hyperparameter, I achieved an outstanding private score of `0.67`. This adjustment led the model to learn more stable and reliable.

Feature Selection (Nhut)

By changing the `cv` variable to `n_splits=6` and `n_splits=7`, I achieved an impressive public score of `0.59389` and `0.59266`. This adjustment not only improved the score but also increased the stability and reliability of the model. Increasing the number of data splits allows for a more accurate evaluation through multiple cross-validation iterations, minimizing the risk of overfitting and ensuring better generalization to unseen data. This clearly demonstrates how fine-tuning parameters can significantly enhance model performance. Unfortunately, this result did not perform as well on the test set compared to `n_split=5`.

Model Evaluate (Nhat)

After reviewing several notebooks that achieved high scores in the public test, I decided to experiment with different cross-validation techniques. First, I employed the widely used `KFold` technique. However, due to the significant class imbalance in the target classes, I also tried the `StratifiedKFold` technique. Furthermore, I explored the GroupKFold technique as I believed it could be valuable in addressing the specific challenges posed by different groups of borrowers in the home credit underwriting problem. Finally, I experimented with the `StratifiedGroupKFold` technique, which combines the advantages of both `StratifiedKFold` and `GroupKFold`. The result was really good, maybe which such data (highly imbalanced and group-based), `StratifiedGroupKfold` was a superior choice. I also experimented with adjusting the `n_splits` hyperparameter in `StratifiedGroupKFold`, following advice from my instructor, Mr. Vu. I increased the value to a higher number such as `8`, `10`, aiming to improve the model's performance. But my notebook ran out of time.

Base Models

All base models were trained on StratifiedKFold with 5-folds.

Models Used:

- **Minh Nhut**: LightGBM, CatBoost, XGboost, Voting Model. Voting Model help leverage the diversity of models
- **Quan**: XGBoost, CatBoost, LightGBM with AUC and LightGBM with the organizer's metric , Weighted Voting Model.
- **Tang Nhat**: RandomForest, LightGBM, BoostRFE, CatBoost, Linear Regression (which surprisingly gave us a **0.502** score in public test).
- **Dang**: XGBoost, CatBoost, LightGBM and Linear Regression. <!-- ### Neural Networks (Michael Jahrer)

Neural networks underperformed compared to boosted trees in terms of AUC. Using DAE+NN improved performance slightly. DAE involved denoising autoencoder preprocessing, with 10000-10000-10000 topology for the first models, later changed to fewer, larger hidden layers. Best AUC was achieved with a DAE with one hidden layer of 50k neurons, followed by a 1000-1000 supervised NN. Despite neural net optimizations, LightGBM with a small learning rate outperformed them. -->

Models Selection (Nhut)

The `VotingModel` class is a custom ensemble model that combines multiple estimators to enhance prediction accuracy. This approach leverages the strengths of different models to improve overall performance. The `__init__` method initializes the model with a list of pre-trained estimators. The `fit` method is essentially a placeholder, as the estimators are already fitted. The `predict` method generates predictions by averaging the outputs of all estimators, providing a more robust prediction than any single model could achieve. The `predict_proba` method handles the prediction of class probabilities, converting categorical features to the appropriate type before averaging the probability predictions from all estimators. This class effectively combines the predictions of various models, such as `fitted_models_cat` and `fitted_models_lgb`, to leverage their individual strengths and mitigate their weaknesses, resulting in a more accurate and reliable prediction model. This ensemble method is particularly useful in scenarios where different models capture different patterns in the data, thereby providing a comprehensive prediction framework.

Tuning parameters (Nhut)

To optimize the performance of our models, we employed Bayesian optimization to determine the optimal set of hyperparameters for LightGBM and CatBoost classifiers. Bayesian optimization is a powerful technique for hyperparameter tuning that iteratively narrows down the search space using a probabilistic model. This method is particularly effective for complex models with many hyperparameters, as it balances exploration and exploitation to efficiently find the optimal configuration.

LightGBM Parameters:

```
params_lgbm = {
    "boosting_type": "gbdt",
    "objective": "binary",
    "metric": "auc",
    "max_depth": 10,
    "learning_rate": 0.05,
    "n_estimators": 2000,
    "colsample_bytree": 0.8,
```

```
    "colsample_bynode": 0.8,
    "random_state": 42,
    "reg_alpha": 0.1,
    "reg_lambda": 10,
    "extra_trees": True,
    "num_leaves": 64,
    "device": device,
    "verbose": -1,
}
```

Catboost Parameters:

```
clf = CatBoostClassifier(
    eval_metric='AUC',
    task_type='GPU',
    learning_rate=0.03,
    iterations=n_est,
    random_seed=3107
)
```

It achieved high performance on the public leaderboard with a score of 0.59389.

Ensembling

We employed ensembling to improve model performance by combining CatBoost and LightGBM models.

- **Training:** Both CatBoost and LightGBM models were trained using cross-validation.
- **Voting Model:** A custom voting model averaged predictions from the trained CatBoost and LightGBM models.
- **Final Prediction:** The final prediction was derived by averaging the probabilities from all models in the ensemble.

This approach leverages the strengths of different models to enhance prediction accuracy and stability.

Insights and Recommendations

- Hyperparameter tuning was minimal. We relied on diverse models for ensembling rather than optimizing individual models.
- Train/test prediction had an AUC of over `0.83`, suggesting potential for adversarial validation or pseudolabeling.
- Highly recommend evaluating the model using the Gini metric provided by the host.
- After some review from discussion, incorporating ratio and log-based features is expected to significantly improve the overall performance of the model.
- Top 10 positions on the leaderboard were dominated by ensembles of LightGBM and CatBoost.
- Our top three base models would have placed in the top 10 individually, suggesting feature engineering and selection are crucial.

In conclusion, feature engineering and selection were the most important steps in this competition, and a single, highly optimized model might outperform complex ensembles.