

**NANYANG
TECHNOLOGICAL
UNIVERSITY**

SINGAPORE

CZ4041 Machine Learning

Project Report:

Elo Merchant Category Recommendation

Tan Ching Fhen
Ng Yue Jie Alphaeus
Lim Yun Han, Darren
Tan Xin Kai

U1920787D
U2021469L
U1921275J
U2022499H

Table of Contents

Table of Contents

Introduction	1
Data	2
Overfitting Management	3
Model Selection	4
Anomaly Detection	5
Hyperparameter Tuning	6
Final Model	7
Research and Experimentation	8
Conclusion	9

1 Introduction

In this project, we conducted comprehensive studies and experiments in feature engineering, feature selection, model selection, and hyperparameter tuning. We also adopted unique strategies such as generating node2vec embedding features, and anomaly detection. Sections 2-7 describe our main machine learning pipeline, while section 8 describes the additional research and experimentations we performed.

1.1 Problem Statement

Elo Merchant Category Recommendation is a *regression* task. The objective is, given the user attributes, merchant attributes, and transaction attributes, predict customer loyalty for each user.

2 Data

2.1 Data Overview

The descriptions of individual features are available in the Data_Dictionary.xlsx file, thus we will skip the introduction of those original features. Instead, this section will give an overview of the original data.

The target variable, “target”, is a numerical loyalty score. Its distribution is shown in Figure 2.

The original data features have a graphical structure, with two main players/nodes - user and merchants. Every user has a unique *card_id* and every merchant has a unique *merchant_id*. Users can have transactions with multiple merchants and multiple transactions with each merchant. These transactions can be thought of as edges between user and merchant nodes. All three entities have attributes such as *purchase_amount* (transaction attribute), *feature_1* (user attribute), and *most_recent_sales_range* (merchant attribute) just to name a few. Therefore, we can think of the data as a hypergraph shown in Figure 1.

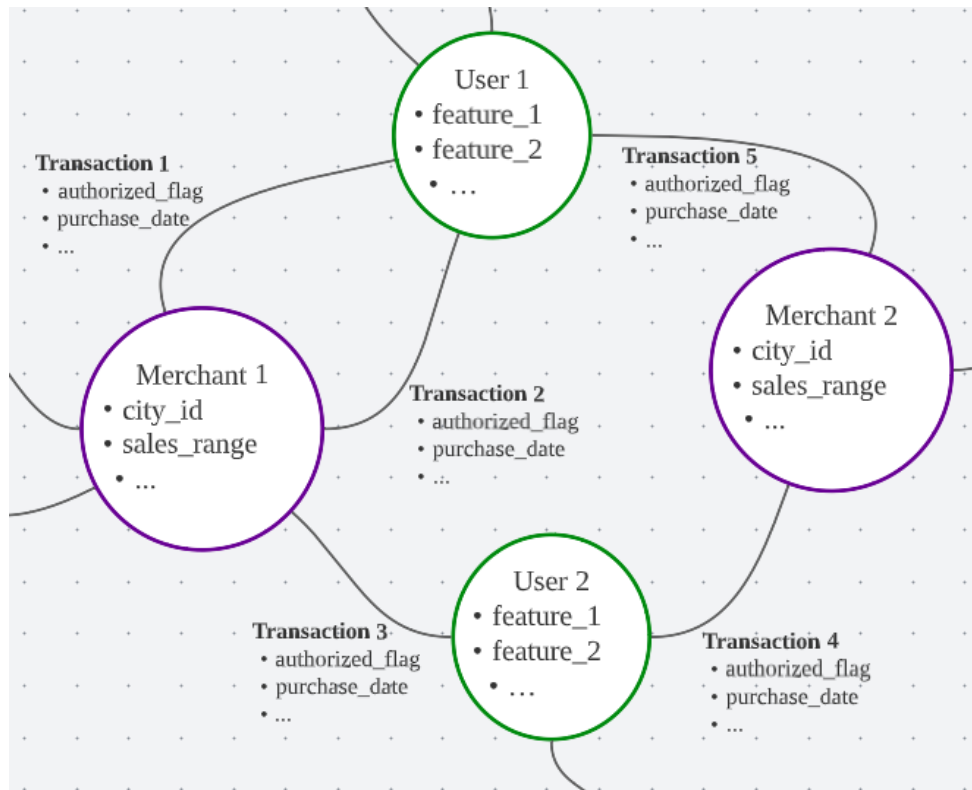


Figure 1 - Simple Hypergraph Illustration of Elo Merchant Dataset

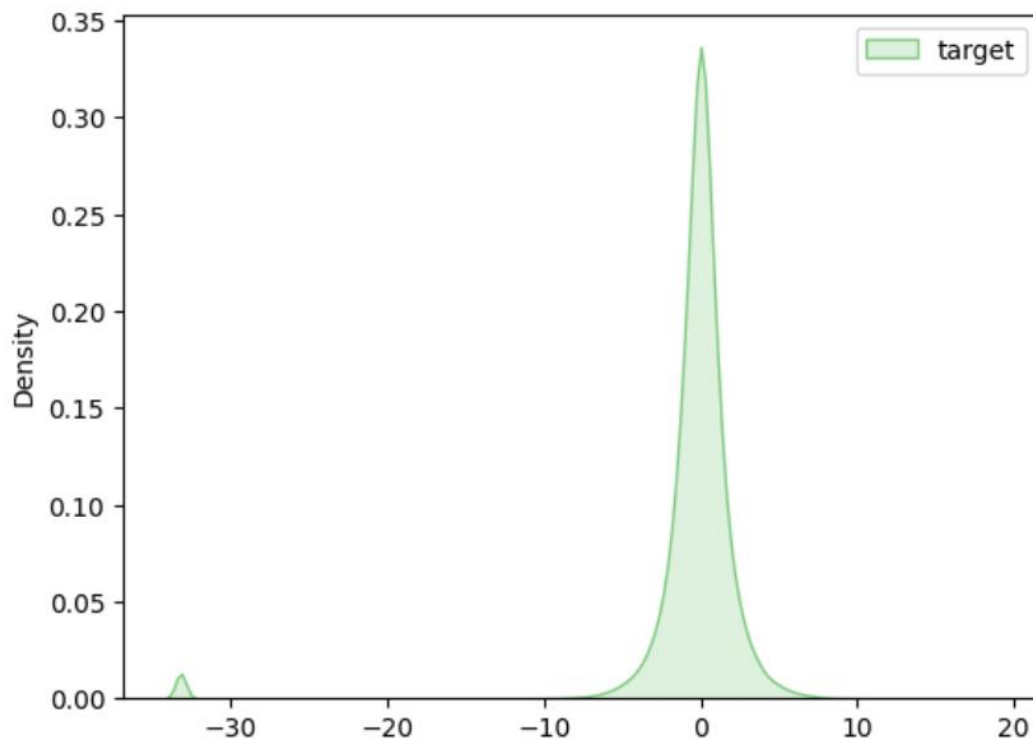


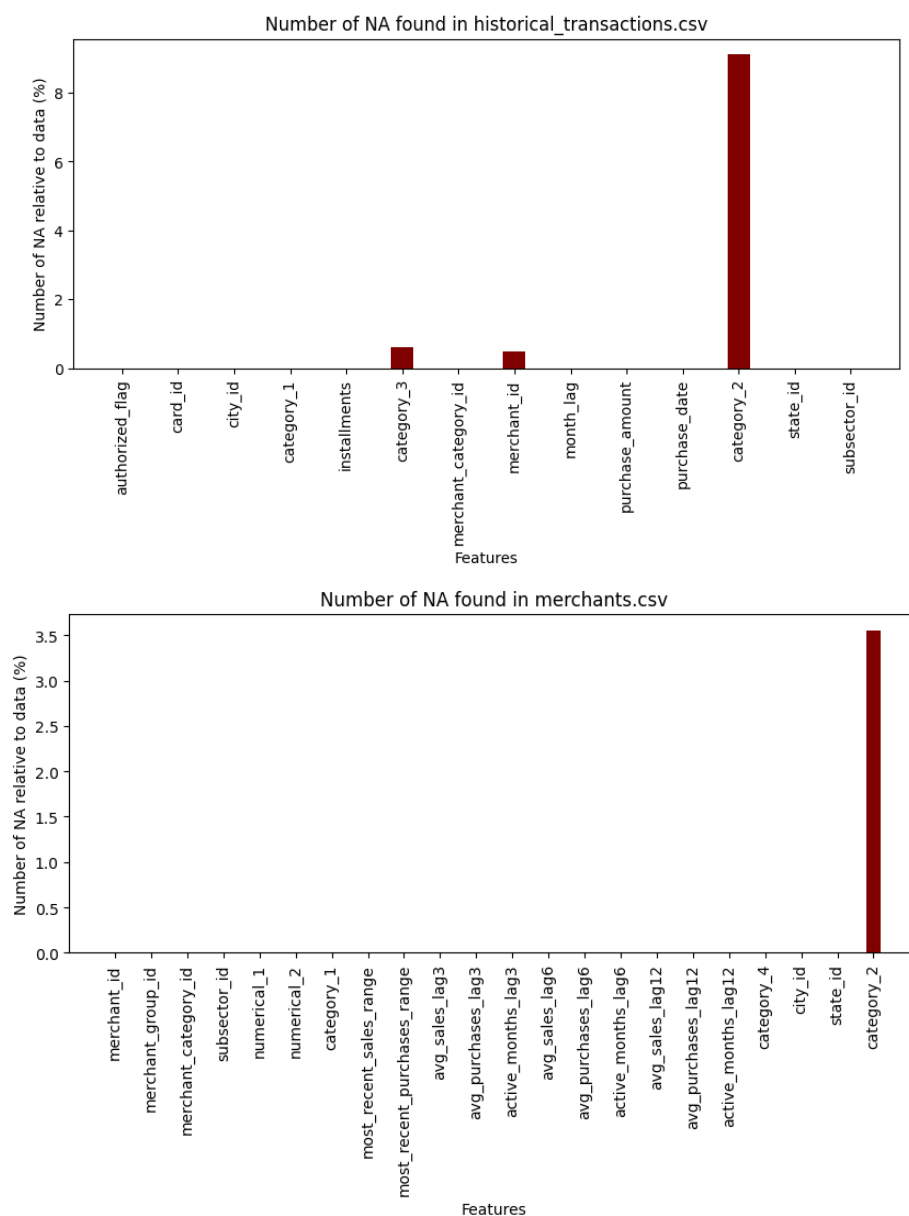
Figure 2 - Distribution of target feature - loyalty score

2.1 Data Cleaning

This section describes the basic cleaning steps we took.

There were several features that contain missing values (Figure 3). For those features, we fill the missing values with mean or mode, depending on whether it is a numerical feature or categorical feature respectively. For instance, some transactions have missing values on the “merchant_id” column, so we replaced it with the mode “M_ID_00a6ca8a8a”. Though there are many other advanced missing value imputation techniques like “missForest”, we decide that it is more productive and efficient to utilise simple imputation on such a large dataset, because advanced imputation techniques will require significantly more computation resources.

Next, all duplicate transactions and merchant rows were dropped. Finally, categorical features were encoded using ordinal encoding.



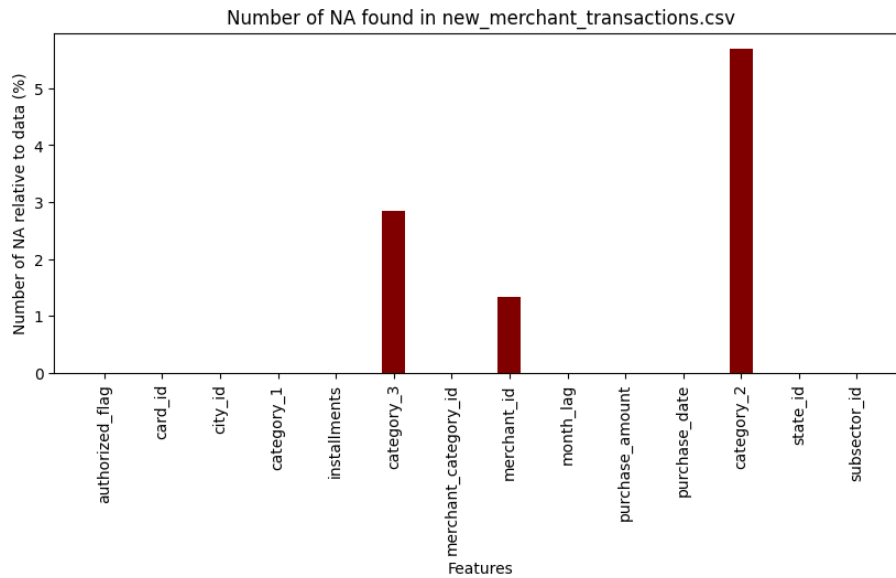


Figure 3 - Proportion of missing values

2.2 Feature Engineering

This section will describe the extensive feature engineering we performed.

Recall that the original dataset features were in the form of a hypergraph (Figure 1), thus it cannot be used as input directly into non-graph machine learning models that we will use. The idea is, for every *card_id*, summarise user, merchant, and transactional attributes into a single vector of features $\mathbf{x} \in \mathbb{R}^{1 \times D}$, where D is the number of features in the input data, $\mathbf{X} \in \mathbb{R}^{325540 \times D}$. To do so, we performed feature engineering in two main ways: aggregation functions and graph embeddings (Figure S1).

Feature Engineering

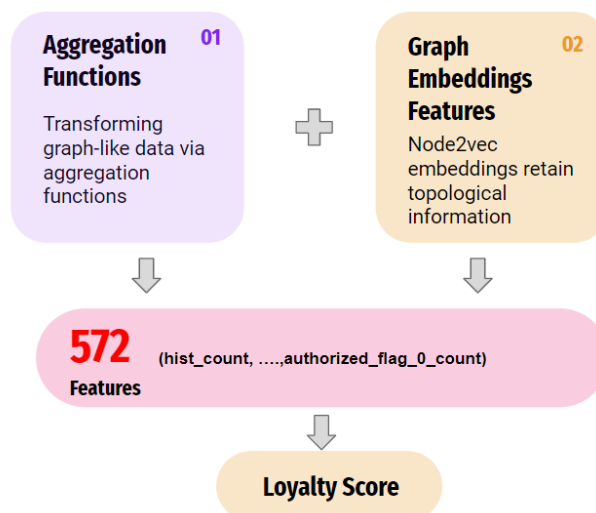


Figure S1 - Overview of feature engineering (from slides)

2.2.1 Aggregation Features

When transforming graph-like data (Figure 1) into **X**, information will naturally be lost, but the goal here is to retain as much information by performing logical transformations and by sheer quantity of features. The aggregation functions we used were mean, median, minimum, maximum, standard deviation, variance, skew, summation, ratio, count, and number of unique classes etc (refer to Figures S2 and S3 for illustration). We applied different combinations of these aggregation functions to the *original features(italicised)* below where appropriate:

1. Between *feature_1*, *feature_2*, and *feature_3*, compute minimum, maximum, summation, and ratio.
E.g. *feature_sum*, *feature_mean*, *feature_min*, *feature_max*...
2. Groupby *card_id*, and compute count and fraction of each *authorized_flag* class.
E.g. *authorized_flag_0_count*, *authorized_flag_1_count*, *authorized_flag_1_fraction*...
3. Groupby *card_id*, and compute count and fraction of each *category_1*, *category_2*, and *category_3* class
E.g. *category_1_0_count*, *category_1_1_count*..., *category_2_1_fraction*, *category_2_2_fraction*...
4. Group by *card_id*, and compute the number of unique classes, and counts for *merchant_id*, *merchant_group_id*, *merchant_category_id*, *subsector_id*, *city_id* and *state_id*.
E.g. *nunique_city_id*, *count_state_id*...
5. Groupby *card_id*, and compute the minimum, maximum, median, std, skew, and summation for *instalments*, *month_lag*, *purchase_amount*.
E.g. *purchase_amount_min*, *purchase_amount_max*...
6. Groupby *card_id*, and compute minimum, and maximum for *purchase_date*.
E.g. *purchase_date_min*, *purchase_date_max*...
7. For (6), compute the difference between the min and max features, and compute the number of days since then.
E.g. *purchase_date_diff*, *purchase_date_uptonow*...
8. Groupby *card_id*, and compute the mean number of days from *purchase_date* to holidays. E.g. *Mothers_Day_mean*, *Christmas_Day_mean*, *Valentines_Day*...
9. Group by *card_id* and compute counts and ratio of classes for *category_4*, *most_recent_sales_range*, *most_recent_purchases_rang*.
E.g. *most_recent_sales_range_1.0_count*, *most_recent_sales_range_1.0_frac*...
10. Group by *card_id* and compute min, max, sum, variance, median, and skew for *avg_sales*, *active_months*, *numerical_2* features.
E.g. *numerical_2_max*, *avg_sales_lag12_skew*...

Since there were two types of transactional data files (historical_transactions.csv contains older transactions and new_merchant_transactions.csv contains more recent transactions), we performed the above feature engineering on each transactional data file *separately* to retain as much information as possible. These two feature types are differentiated by the prefix “old_” and “new_”.

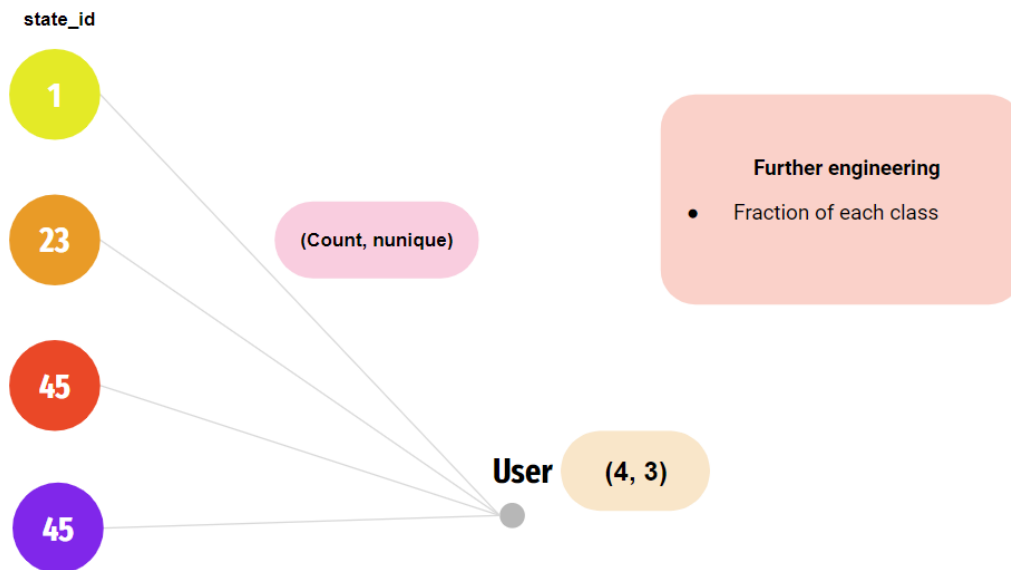


Figure S2 - Illustration of aggregating categorical features (from slides)

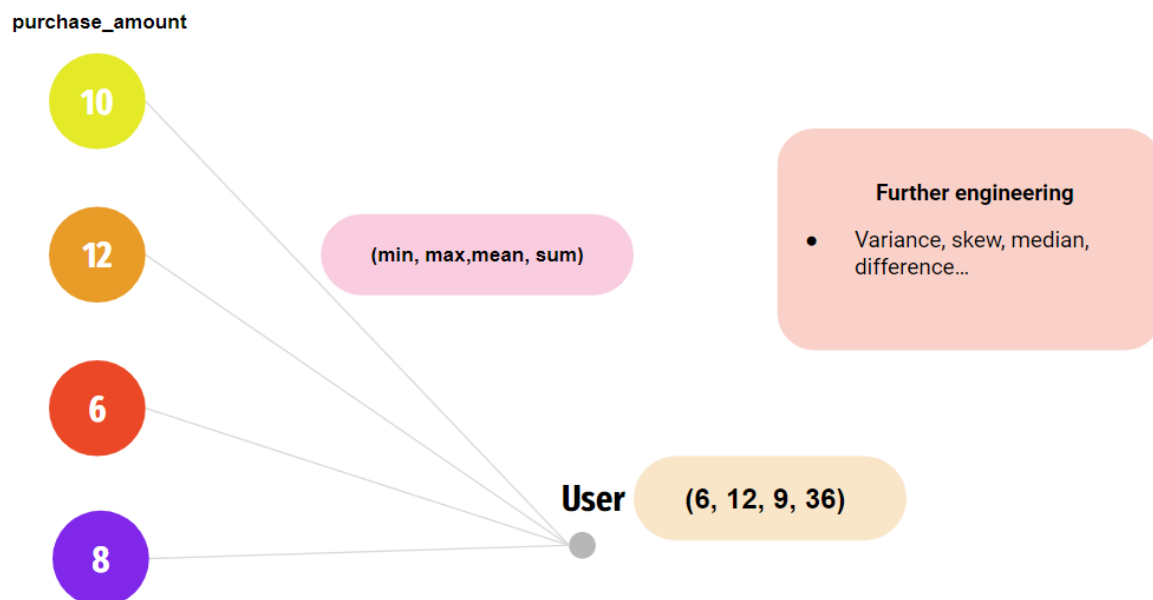


Figure S3 - Illustration of aggregating numeric features (from slides)

2.2.2.1 Graph Embeddings Features

Next, we generated graph embeddings to retain topological information. Since the original dataset has a hypergraph structure (Figure 1), topological information may be potentially useful in predicting customer loyalty, however many previous works from other competitors do not utilise such features, thus our approach is unique.

For *card_id*, *merchant_id*, *merchant_group_id*, *merchant_category_id*, *subsector_id*, *city_id*, and *state_id*, treat each unique identifier as a node. Create an edge between the nodes if they are related by a transaction. Next, we generated node2vec embeddings using the *nodevectors [1]* library - this library makes use of compressed matrices to generate the embeddings, which is immensely beneficial given the large dataset. Thus, each node is associated with an embedding of size 16 - we experimented with various other sizes and found dimensionality 16 to be optimal.

Finally, group by *card_id* and compute the mean of embeddings and for *merchant_id*, *merchant_group_id*, *merchant_category_id*, *subsector_id*, *city_id*, and *state_id* (Figure S4). Thus, each *card_id* is associated with a set of embeddings of size 16.

E.g mean_embedding_merchant_id, mean_embedding_merchant_group_id, mean_embedding_merchant_category_id...

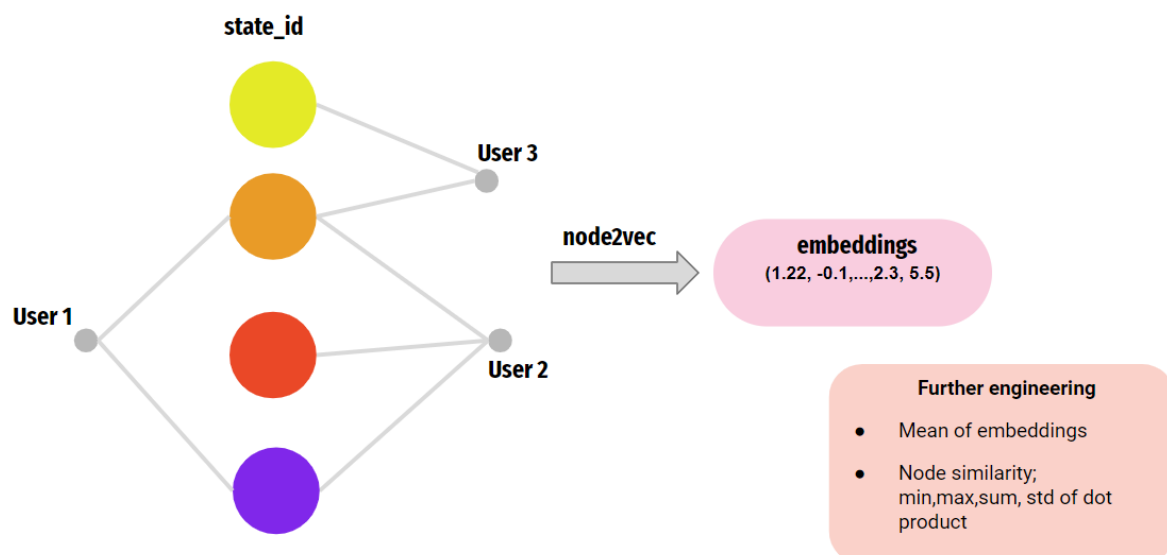


Figure S4 - Illustration of node2vec embeddings (from slides)

2.2.2.2 Graph Embeddings Evaluation

To test our hypothesis that graph embeddings were indeed beneficial, we conducted a simple experiment as follows: Train a baseline model with only *feature_1*. Train a second model with the graph embeddings. Train a third model with a subset of *related* aggregated features from Section 2.2.1 (aggregated features number 4). Lastly, train a model on all features in this experiment.

Figures 4 to 6 illustrate how the embedding features performed. From Figures 4 and 5, we can observe that *merchant_category_id* embeddings and *state_id* embeddings performed better than baseline, this implies that topological information is useful in predicting customer loyalty. When compared to simple aggregation (+*id_features*) however, embedding features performed slightly poorer in the test set, implying some degree of overfitting (Figures 4 and 5). Lastly, the model trained on *both* embeddings and *related* aggregation features performed the best. This implies that the two different types of features are not correlated, although they were transformed from the same set of original features.

However, not all embeddings were useful predictors of customer loyalty. For instance, embeddings generated from *merchant_id* tended to cause overfitting, with very little improvement in generalisation (Figure 6). A possible reason is that the *merchant_id* feature had a high cardinality.

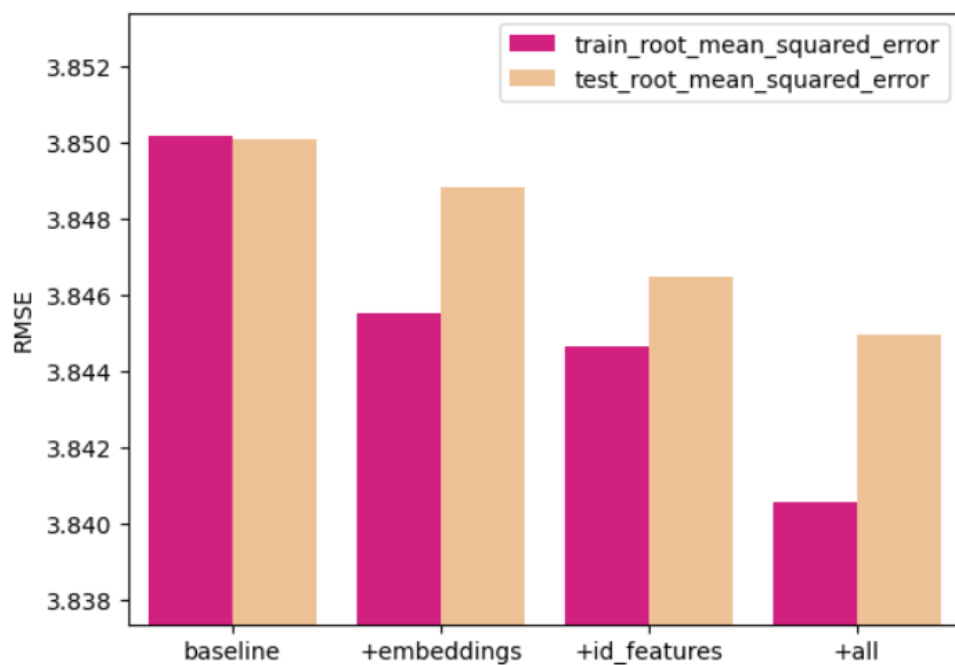


Figure 4 - Evaluation of *merchant_category_id* embeddings against other models

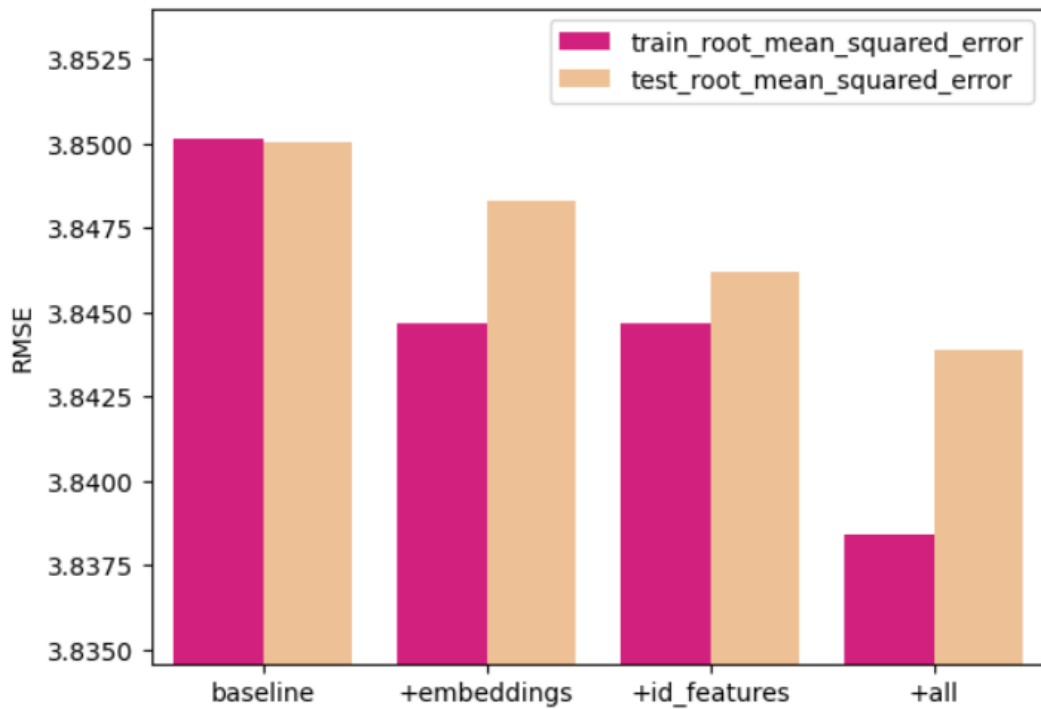


Figure 5 - Evaluation of *state_id* embeddings against other models

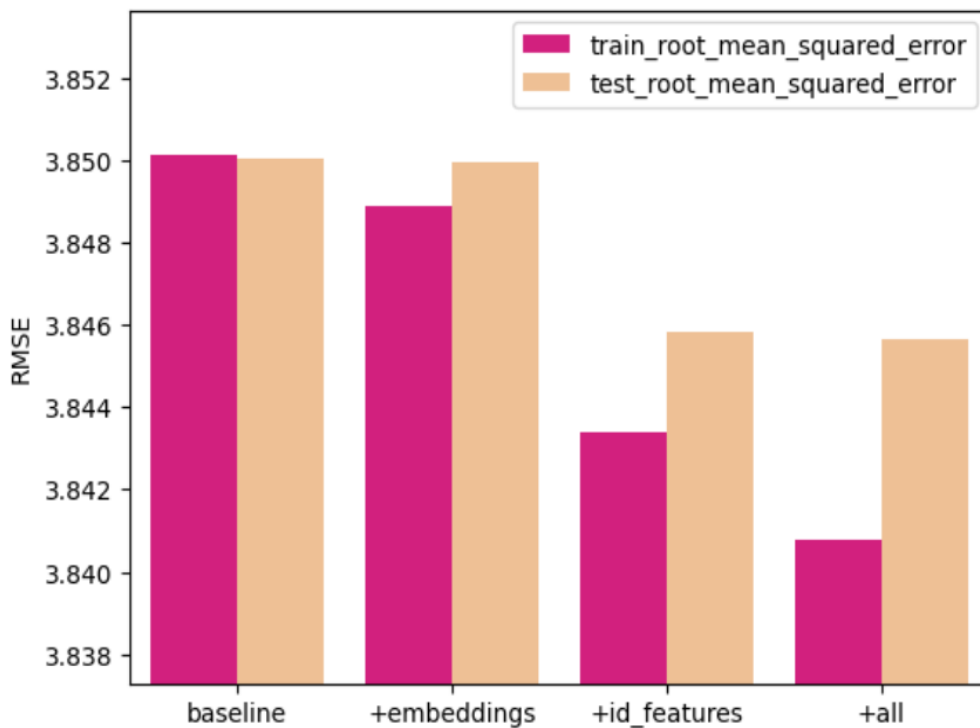


Figure 6 - Evaluation of *merchant_id* embeddings against other models

Nonetheless, we showed that embedding features contains useful topological information that were not correlated with just simple aggregation. Thus, for all embeddings that produced an improvement in model generalisation, we included them in our final model.

2.2.3 Graph Embeddings Similarity Features

Finally, we also generated node similarity features using the embeddings generated in the previous section. For each node embedding, we computed the dot product similarity with the *card_id* embeddings, grouped by *card_id*, then performed min, max, std, and sum aggregation of these similarity scores (Figure S4). E.g merchant_category_id_similarity_min, merchant_category_id_similarity_max...

In summary, we engineered new features using aggregation functions (Section 2.2.1), node2vec embeddings (Section 2.2.2.1) and embedding similarity features (Section 2.2.3). In total, our input dataset **X** constitutes 572 features (Figure S1).

All features are listed in the appendix.

2.2.4 Feature Importance

Based on Figure 7, the top 20 most important features in predicting customer loyalty were all features engineered using simple aggregation from Section 2.2.1.

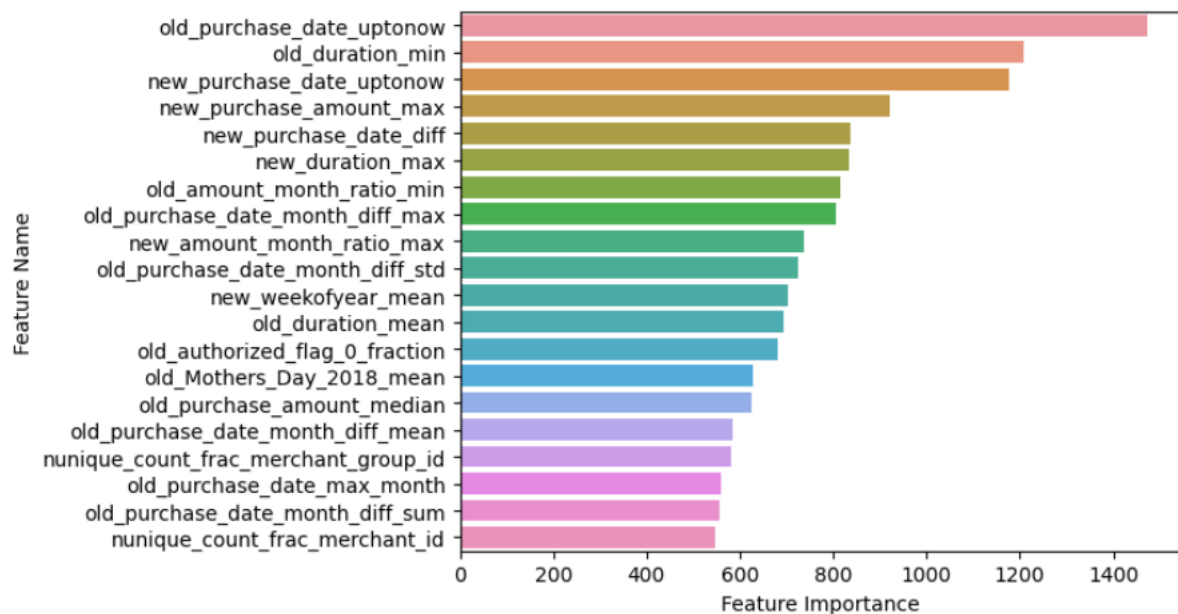


Figure 7 - LightGBM Feature Importances of top 20 features

3 Overfitting Management

Given the extensiveness of feature engineering, managing the complexity of our model is a priority because the more complex our model, the more likely it is to overfit. This section will describe some techniques we utilised to reduce overfitting

3.1 Feature Selection

There are many methods to perform feature selection such as forward selection, backward elimination, and Boruta. A key consideration when choosing our feature selection approach is efficiency. Due to the sheer size of the input data **X**, our attempts at using more advanced techniques like Boruta were futile because of the computational requirement. Thus, we resorted to a simple, and effective backward elimination approach to reduce our feature set:

1. Perform a threefold cross validation of a LightGBM model [2] on **X**
2. Compute the feature importance scores
3. Remove the least important feature from **X**
4. Repeat 1-3 30 times.

From Figure 8, observe that as more features were removed from input **X**, performance improves - this observation is in line with the Occam's razor principle. After the consecutive removal of 17 features, further removal of features saw a worsening of performance. Thus, for our final model, we used a reduced model with these 17 removed features.

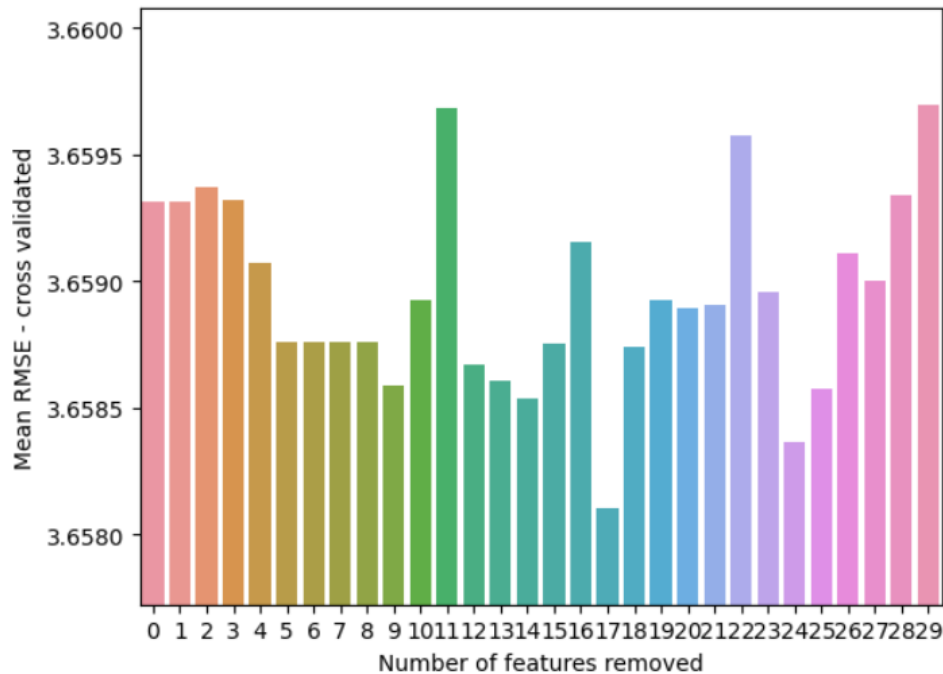


Figure 8 - Backward Elimination Experiment results

3.2 Other overfitting management techniques

Besides feature selection, we adopted various other measures to manage overfitting during all our experiments, training, and hyperparameter tuning. First, we always utilise early stopping to halt training when overfitting is observed i.e validation score worsens. Also, we prioritise tuning of hyperparameters that manage overfitting such as L1 and L2 regularisation values, bagging fractions and feature fractions. In addition, we utilised at least a 3-fold cross validation to ensure a reliable and unbiased estimate of model performances. Finally, features engineering steps in Section 2.2 were performed on training and test sets separately to avoid target leakage.

4 Model Selection

We performed comprehensive studies and experimentations to select the best model based on speed and performance. Among Lightgbm, random forest, extra trees model, Catboost model, XGBoost, and support vector machines (SVR), we found that LightGBM was the most optimal in terms of speed and performance (Figure 9), thus we utilised LightGBM in subsequent parts of our model pipeline.



Figure 9 - Comparison of models by RMSE. LightGBM(Blue), XGBoost(Yellow), Catboost(Pink), Random Forest(Red), SVR(Green).

For more details of model comparison, please refer to our research and experiment in Section 8.

5 Anomaly Detection

In this section, we describe how we utilised anomaly detection to improve model generalisation.

5.1 Isolation Forest

We utilised the isolation forest algorithm from the scikit-learn library. The algorithm 'isolates' observations by randomly selecting a feature and then randomly selecting a split value between the maximum and minimum values of the selected feature. Since recursive partitioning can be represented by a tree structure, the number of splittings required to isolate a sample is equivalent to the path length from the root node to the terminating node. This path length, averaged over a forest of such random trees, is a measure of normality and our decision function. Finally, the isolation forest algorithm returns the anomaly score of data points.

5.2 Isolation Forest Tuning

There were two main hyperparameters to tune for the isolation forest algorithm: `n_estimators` and contamination fraction. The former signifies the number of base estimators for the algorithm and the latter signifies the fraction of the dataset which we deem to be an anomaly.

From Figures 10 and 11, we can observe that 750 and 1.5 were the best values for `n_estimator` and contamination fraction respectively. We used these hyperparameters for anomaly detection in subsequent sections.

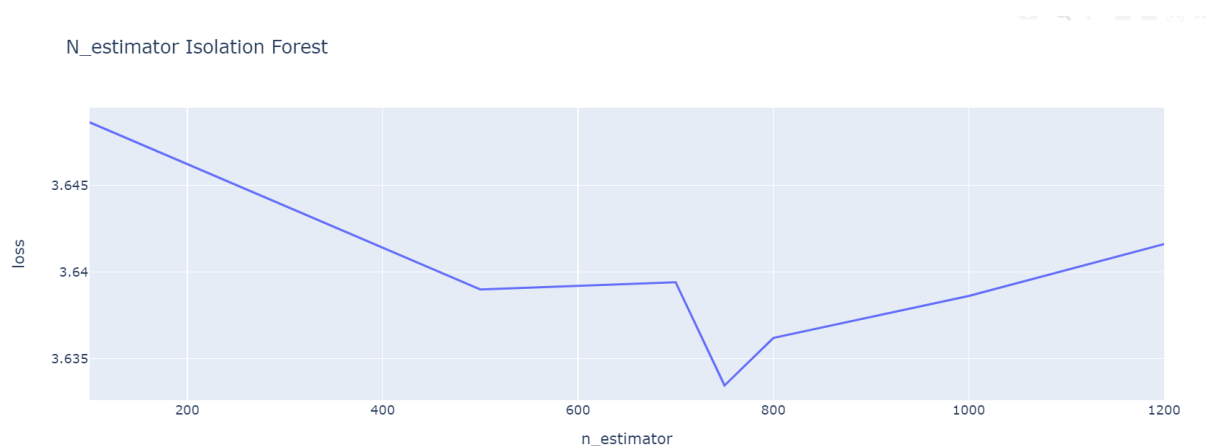


Figure 10 - Isolation forest loss against `n_estimator`

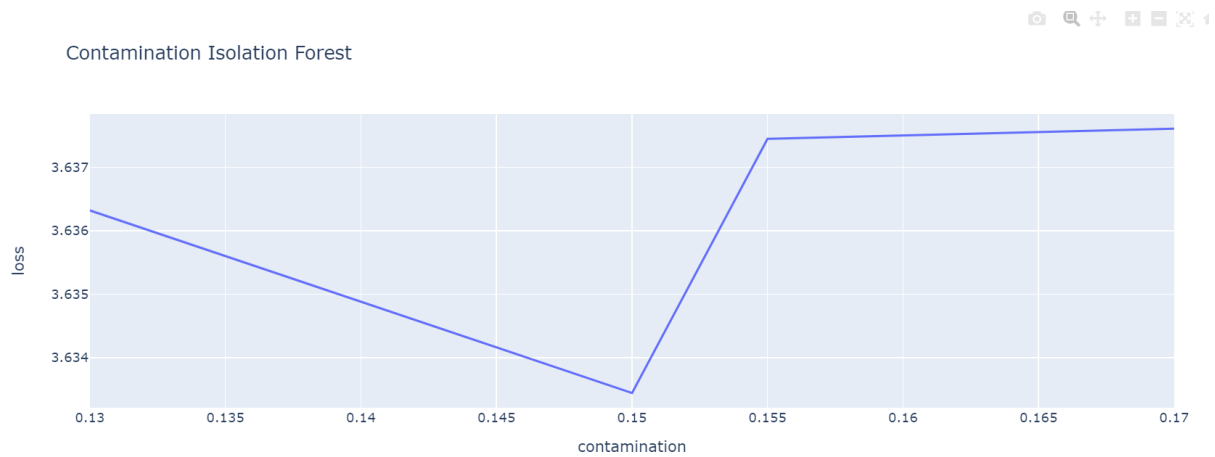


Figure 11 - Isolation forest loss against contamination

5.3 Model Training

We hypothesised that anomalies in our input data X could have worsened our model generalisation. Thus, we attempted to alleviate this problem by training another model on anomalies separately.

Our strategy was as follows:

1. Train a model on the anomalies detected from the best isolation forest in Section 5.2
2. Train another model on all other data points
3. Concatenate the predictions from both models

The figures below illustrate hyperparameter tuning of two LightGBM models, one on anomaly data points (Figure 12) and another on non-anomaly data points (Figure 13). We can observe that model generalisation was much more difficult for the anomaly-trained model as the performance gain from hyperparameter tuning was less pronounced.

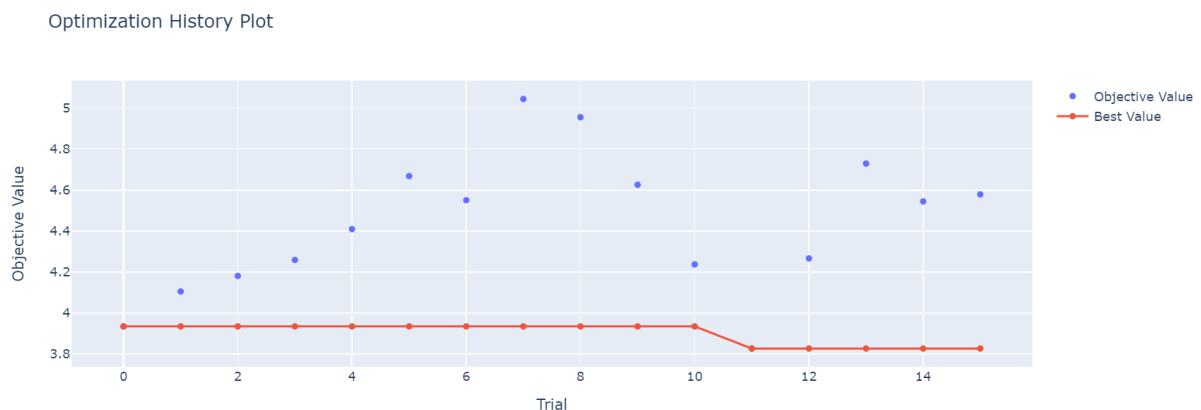


Figure 12 - Optimization history on anomaly data

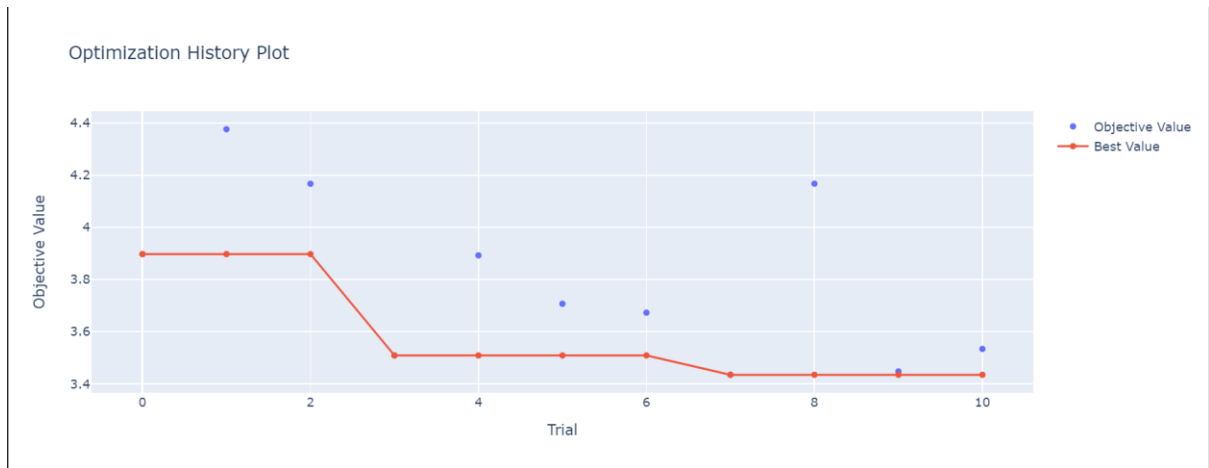


Figure 13 - Optimization history on data without anomaly

5.4 Evaluation of strategy

Finally, we compared the performance between a model trained on the entire dataset against our strategy in the previous section. From Figure 14, we can observe that training a separate model on anomalies does yield a slight performance gain of about 0.0011 root mean squared error (RMSE). Thus, we utilised this strategy for our final model.

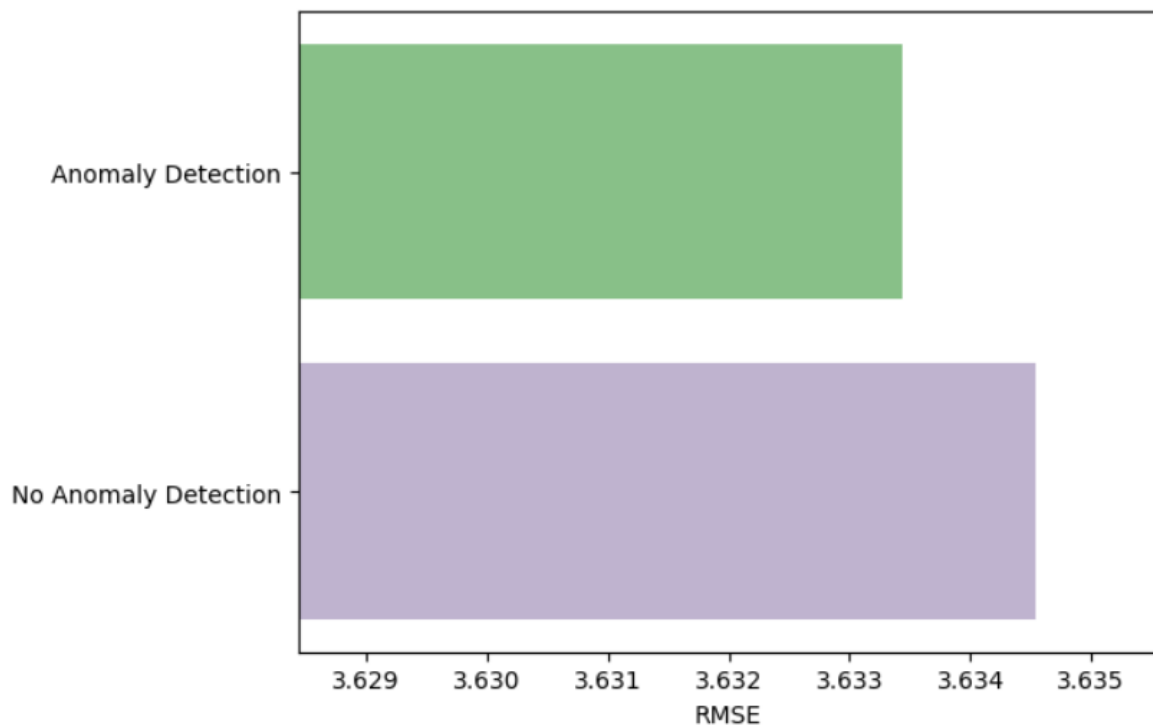


Figure 14 - Performance improvement from anomaly detection

6 Hyperparameter Tuning

Parameter tuning is an essential step to optimise the performance of our model. There are many free softwares available to perform hyperparameter tuning, but we chose Optuna because it provides advanced sampling, pruning and visualisation capabilities. This section describes our hyperparameter tuning process on the LightGBM model.

6.1 Optuna

Optuna is an automatic hyperparameter optimization software framework. It has many types of sampling algorithms including basic algorithms like grid search and random search to more complex algorithms such as Tree-structured Parzen Estimator algorithm and CMA-ES based algorithm. Furthermore, the library also includes different pruning algorithms which stops the trials early should the trial prove to be unpromising. Pruning algorithms include Asynchronous successive halving algorithm, Hyperband algorithm and Median pruning algorithm. Using Optuna, we significantly improved performance and reduced training time.

6.2 Tuning Strategy

Our hyperparameter tuning strategy was as follows:

1. Fix learning rate at a relatively large value e.g 0.1
2. Perform hyperparameter tuning with Optuna on all other hyperparameters
3. Using the best hyperparameter set, perform hyperparameter tuning on the learning rate

This strategy is efficient because, having a high learning rate reaches convergence much faster, reducing training time and allowing more trials to be performed. From Figure 9, we can observe that for LightGBM, the lambda regularisation hyperparameter and the maximum depth of the boosting model were the most significant hyperparameters. Thus, we conducted more trials on those hyperparameters during the tuning process. In total, we conducted more than 500 trials of hyperparameter tuning on LightGBM.

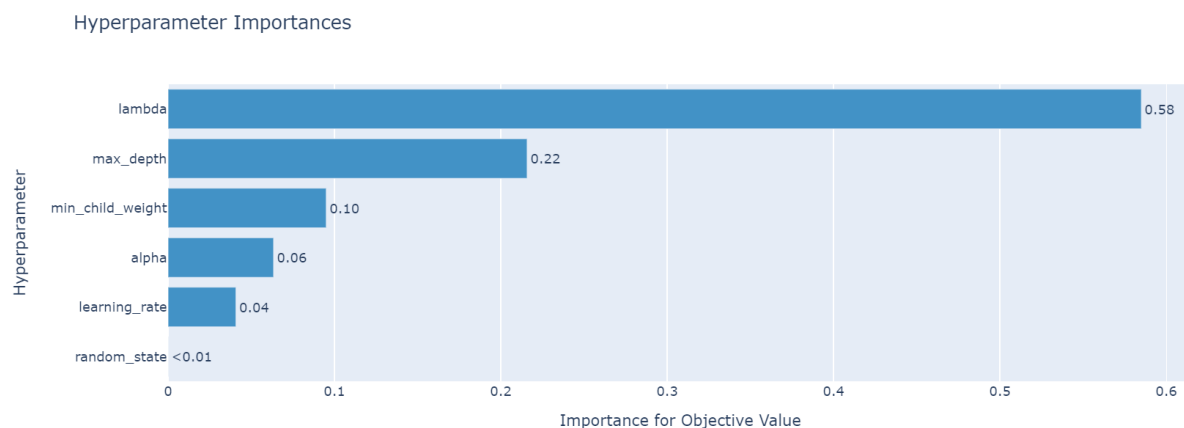


Figure 15 - Hyperparameter importance of LightGBM - Optuna

7 Final Model

This section describes the training of our final model based on the outcomes of the previous sections.

7.1 Model Training

Using the reduced input data **X** from Section 3.1, separate the anomaly data points found in Section 5. Perform a ten-fold cross validation of a LightGBM model on the anomaly data points and another on the non-anomaly data points, using the best hyperparameters found during hyperparameter tuning (Section 6). For both models, stop model training once the cross validation does not improve for 10 epochs.

7.2 Model Prediction

Perform the same steps of feature engineering (Section 2.2) separately to prevent target leakage. Using the tuned isolation forest model from Section 5.2, perform anomaly detection on the test set. Using the anomaly-trained model and non-anomaly-trained model from the previous section, make the predictions on the test set and concatenate the predictions from the two models.

7.3 Leader Board Score

Our private leader board score was 3.62083, while our public leader board score was 3.70878. Based on the private leader board, our final ranking would be 2053 out of 4111 submissions, which is under 50% (Figure 16a).





2050	▲ 177	zhangtestone		3.62082	53	4y
2051	▲ 421	Amit Munje		3.62082	4	4y
2052	▼ 197	z z		3.62086	5	4y
2053	▲ 371	Ladomi		3.62086	2	4y

Figure 16a - Leader board ranking



Figure 16b –Best Score

8 Research and Experimentation

In this section, we will describe the research and experiments we performed that contributed to our model pipeline from Sections 2 to 7.

8.1 Models used

We have researched and experimented using many different models to compare against one another to see which model could lead to better performance. The models we have utilised are: LightGBM, Random Forest, Extra Trees, Catboost and XGBoost. Each model has their own approach to tackling regression problems and handling multi data type inputs (categorical, integers, dates etc.)

8.1.1 LightGBM

LightGBM or light gradient-boosting machine, is a free, open-source distributed gradient-boosting framework that is based on decision tree algorithms. LightGBM can be used for ranking, classification and other machine learning tasks. The development focus is on performance and scalability. This can be observed as LightGBM splits the decision tree leaf-wise with the best fit while other boosting algorithms splits the tree by level or depth. LightGBM can lower the loss more compared to level based algorithms in growing the same leaf. Therefore, this results in production of more complex trees leading to higher accuracy. However, in experimentation we note that LightGBM is prone to overfitting due to its sensitivity and performs worse with smaller datasets. Hence by training on suitable datasets, and tuning the max-depth parameter, LightGBM is incredibly useful to produce results rapidly while having lower memory usage.

8.1.2 Random Forest

Random forests or random decision forests is a machine learning technique utilising ensemble learning. Random forests use the basis of decision trees while differing through random selection of root node and node segregation. It fits a number of classifying decision trees on various sub-samples of the dataset and uses averaging that improves accuracy and control over-fitting from the bootstrap sample. In our experimentation as it is a regression task, Random forest returns the mean or average prediction of the individual trees.

We are utilising `sklearn.ensemble.RandomForestRegressor` from the scikit-learn machine learning library in python thus it is also important to note the `max_features` and `bootstrap` parameters. Where `max_features` control the number of features to consider when looking for the best split and default `max_feature = 1.0` is equivalent to bagged trees where more randomness can be achieved by setting smaller values. `Bootstrap` controls whether bootstrap samples are used to build trees and default `bootstrap=True` (default), if false the whole dataset is used to build each tree.

8.1.3 Extra Trees

Extra trees is an ensemble supervised machine learning method. It works on top of random forest, adding more randomisation to yield extremely randomised trees. In extra trees, each

tree is trained using the whole learning sample, not the bootstrap sample. Furthermore, top-down splitting in the tree learner is randomised. Cut-points are randomly selected instead of computing the locally optimal cut-point for each feature under consideration with gini or entropy. We are utilising `sklearn.ensemble.ExtraTreesRegressor` from the scikit-learn machine learning library in python where it is similar to random forest and parameters important to note are the `max_features` and `bootstrap` parameters.

8.1.4 Catboost

Catboost is an open source machine learning algorithm that uses gradient boosting on decision trees. While catboost itself is a model to tackle categorical problems, the library has `CatboostRegressor` which handles regression problems which is the model we have used from the Catboost library. The advantages to using the catboost library is that it provides many in build methods for both parameter tuning and model evaluation. An additional benefit of using catboost is that there is little data preparation needed before fitting the model with inputs as the model is capable of handling most data. However there are some drawbacks to the catboost model as well. Through our experimentation, we have found the use of external parameter tuning libraries such as optuna to be a better parameter tuning library compared to the catboost's parameter tuning library, effectively making this feature obsolete. While catboost is a relatively well known library for machine learning, there are some third party libraries that do not support catboost models for example, parameter tuning library Ray. This could be attributed to the fact that catboost is relatively new, being released in 2017 as compared to LightGBM, 2016, and XGBoost, 2014, 2 very well known machine learning libraries.

8.1.5 XGBoost

XGBoost is an optimised distributed gradient boosting library designed to be highly efficient, flexible and portable. It implements machine learning algorithms under the Gradient Boosting framework. XGBoost provides a parallel tree boosting (also known as GBDT, GBM) that solves many data science problems in a fast and accurate way. Being released in 2014, it is a very well known and widely used library for machine learning. In the library, they have many different types of models available and while we did not manage to experiment with all their models, the models that we did experiment on produced good results. Owing to the fact that the library has been released for quite some time now, and the fact that it is a widely used library especially for kaggle problems, it supports many third party libraries for feature selection, feature removal and parameter tuning.

8.1.6 SVR

Support Vector Regression (SVR) is a supervised learning algorithm that is used to predict discrete values. SVR uses the same principle as the SVMs. The basic idea behind SVR is to find the best fit line. In SVR, the best fit line is the hyperplane that has the maximum number of points. SVR carries certain advantages that are as mentioned below:

1. It is robust to outliers.
2. Decision model can be easily updated.
3. It has excellent generalization capability, with high prediction accuracy.
4. Its implementation is easy.

Some of the drawbacks faced by Support Vector Machines while handling regression problems are as mentioned below:

1. They are not suitable for large datasets.
2. In cases where the number of features for each data point exceeds the number of training data samples, the SVM will underperform.
3. The Decision model does not perform very well when the data set has more noise i.e. target classes are overlapping.

8.2 Model Speed

After tuning the models for accuracy we wanted to see if we could improve our machine learning workflow by obtaining a model or process with greatest time based performance without sacrificing accuracy. As such more experimentation was conducted into techniques to speed up data fitting and understanding the time complexity disparity between the different models with accuracy comparisons to evaluate our models..

8.2.1 Speed Comparison

As mentioned, we attempted to evaluate the models based on their respective speed performance to understand if there is an optimum model to employ in order to improve speed performance while preserving accuracy. Thereafter, we also evaluated the different algorithm factors that resulted in time complexity disparity in a bid to understand the conditions required to employ a suitable model. Eventually, results of which are applied to our training dataset in order to determine if we are able to use a specific model that provides both speed and accuracy performance to solve our problem statement for ELO Merchant.

In our benchmark test, we standardised the input for train and test, and the parameters used are as follows: (n_estimators = 100, random_state = 0, max_depth=3).

This was enforced for the 6 models tested above.

In order to determine the comparative performance of the respective models, we trained the models on a subset of features and iteratively increased the size of training data by introducing more features. This was an attempt to simulate scaling of training data size and graph out the time complexity for the different models as well as determine the models accuracy performance. We obtained the RMSE of model predictions as the quantifier of accuracy and execution time of model training as a quantifier of model training speed.

Models	Colour
Lightgbm	Blue
Random Forest	Red
Extra Trees	Green
Catboost	Magenta
XGBoost	Yellow
SVR	Black

Figure 17 - Model Legend

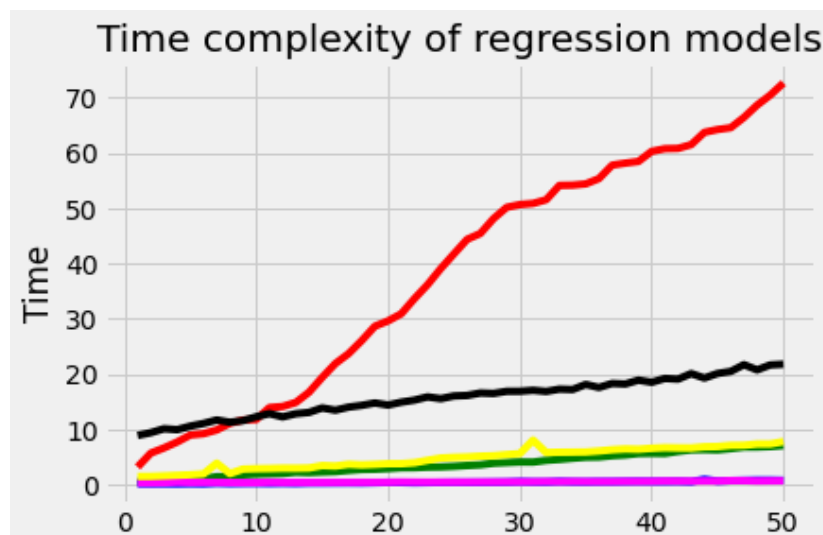


Figure 18 - Time complexity for increasing number of features for 6 Models

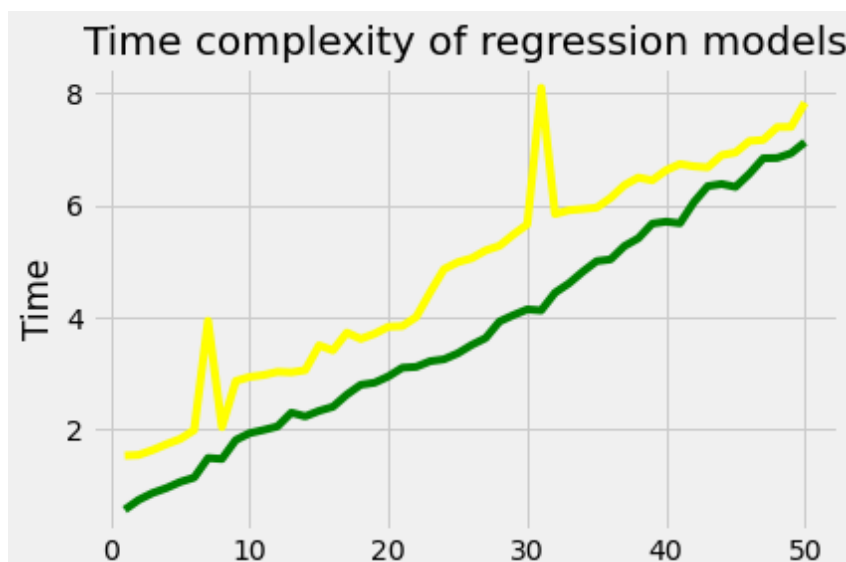


Figure 19 - Closer comparison of Extra Tree and XGBoost

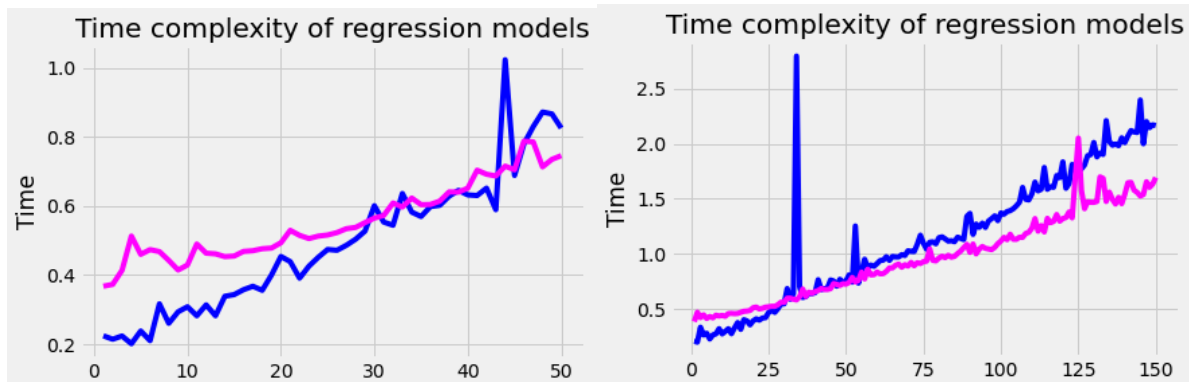


Figure 20 - Closer comparison of LightGBM and CatBoost

In order to understand if we are able to optimise the training time of our models, we observed discrepancies between different models and examined the reasoning behind the speed performance difference to determine if we can use a model that is suitable for our dataset.

8.2.2 Discrepancies in decision tree forests

Random Forest	Extra Tree
Random forest makes use of bootstrap replicas. Subsampling input data with replacement.	Extra Trees uses the whole original sample in the default scikit-learn implementation.
Random forest does selection of cut points to split nodes is done via the optimum split.	Extra Trees choose cut points randomly. Upon selecting the split points are selected, the two algorithms choose the best one between all the subset of features.

8.2.3 Discrepancies in gradient boosting algorithms

CatBoost	LightGBM	XGBoost
Minimal variance sampling (MVS) at tree-level. A weighted version of Stochastic Gradient Boosting.	Gradient-based One-Side Sampling (GOSS), keeps data instances with large gradients and performs random sampling for data instances with small gradients.	XGboost does not use any weight sampling techniques, the splitting process is the slowest. Pre-sorted and histogram-based algorithm for computing the best split
Handling of missing values with 2 modes, "min" and "max" where missing values are processed as the	Missing values will be allocated to the side that reduces the loss in each split. LightGBM grows leaf-wise for best fit tree. Growing leaf that	Missing values will be allocated to the side that reduces the loss in each split. Splits up to the specified max_depth then prunes the

minimum value for a feature if “min”.	minimises loss, allowing a growth of an imbalanced tree. Resulting in it offering faster speed.	tree backwards, removing splits beyond which there is no positive gain.
---------------------------------------	---	---

8.3 Performance Comparison

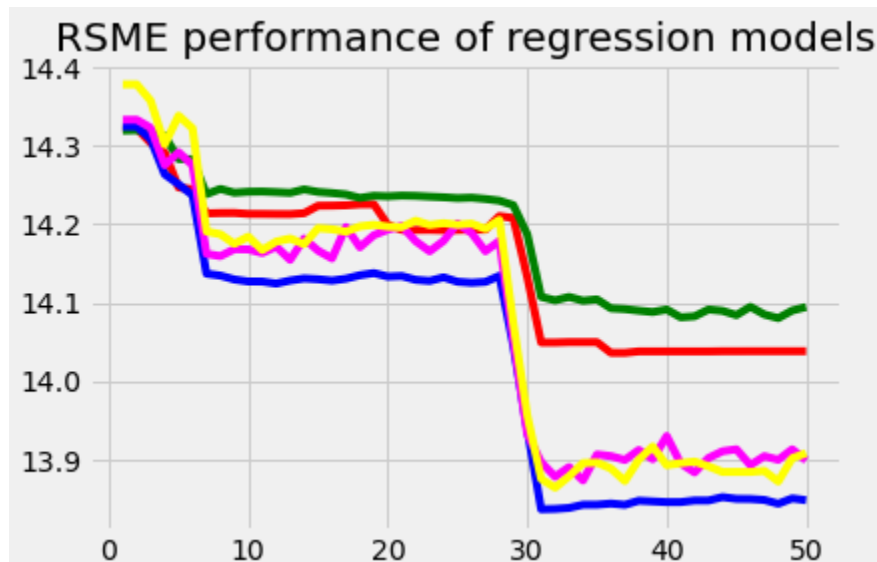


Figure 21 - Accuracy for increasing number of features for 6 Models

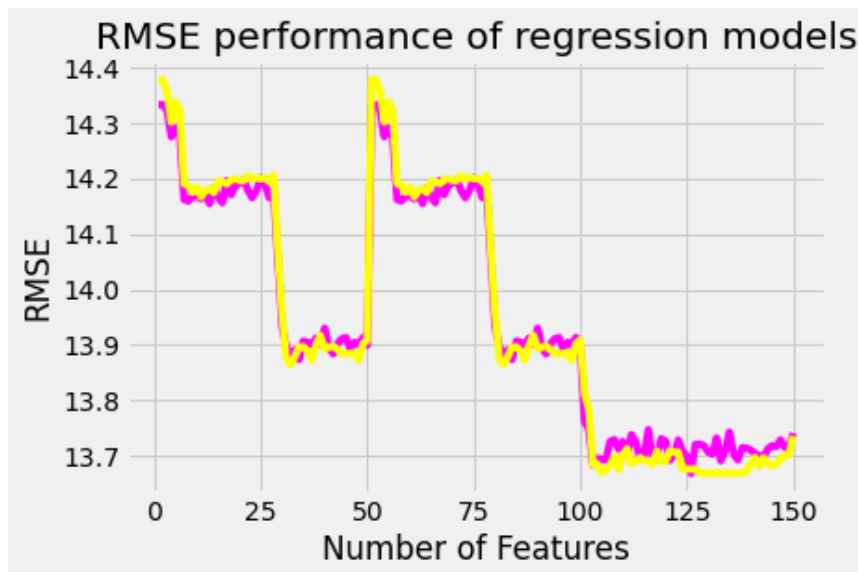


Figure 22 - Closer comparison of XGBoost and CatBoost

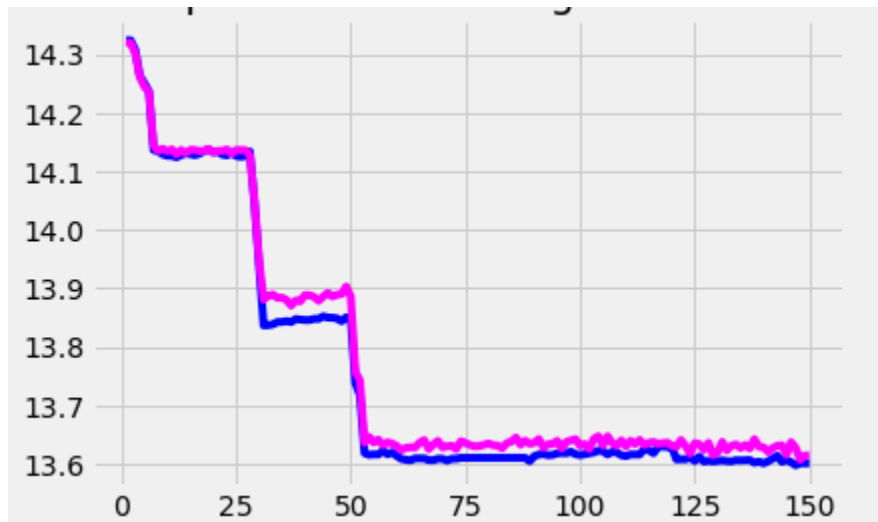


Figure 23 - Closer comparison of Catboost and LightGBM

In Figure 21, we can observe that models utilising gradient boosting outperforms the models with using random forest and extra tree methods. We wanted to ensure that such performance would be reflected in our project and thus looked into the reason for discrepancies to justify the difference in performance.

8.3.1 Discrepancies in gradient boosting vs random forest algorithms

Algorithms	Random Forest Algorithms (RF)	Gradient Boosting Algorithms (GB)
Strengths	<p>RF generates many trees randomly which reduces variance.</p> <p>Hyperparameters are not needed in Random Forest and it is easier to understand and visualise Random Forest algorithms with few parameters present in the data.</p>	<p>GB builds each additional tree sequentially as it tries to get the model closer to the target, this reduces the bias of the model rather than the variance, leading to better performance.</p> <p>In GB, you may keep improving as the ensemble size increases.</p>
Weakness	<p>In RF, increasing the ensemble size past a certain point doesn't lead to an increase in performance.</p>	<p>GBs are more sensitive to overfitting if the data is noisy.</p> <p>Training generally takes longer because of the fact that trees are built sequentially.</p>

		GBs are harder to tune and more sensitive to hyperparameter tunings than RF.
--	--	--

8.3.2 Discrepancies in gradient boosting algorithms

CatBoost	LightGBM	XGBoost
Use a combination of one-hot encoding and advanced mean encoding. MVS in tree-level. Each boosting tree observation are sampled to maximise accuracy of split scoring.	Accepts parameter to check which column is a categorical column and split on equality.	No equal handling of categorical features like in CatBoost and LightGBM. Treats categorical features as numerical variables. Result in lower accuracy for our project with multiple categorical features.
CatBoost results in symmetric trees while other results in asymmetric trees.	May be prone to overfitting with small data, controlled by tuning the max_bin, num_leaves parameters. Works better with larger training data in our project.	XGBoost has a "Coverage" method, the relative number of observations per feature used to decide the leaf node for

8.4 Summary of model comparison

The following is a summary of our findings on accuracy and speed performance ranking. Colour coded for ease of comparison to the graph visualisation.

Models	Lightgbm	Random Forest	Extra Trees	Catboost	XGBoost	SVR
Accuracy Rank	1	4	5	2	3	6
Speed Performance Rank	2	6	3	1	4	5

Ultimately, performance for machine learning models should prioritise accuracy of predictions. Therefore, the best model that our team selected to use in our project was the LightGBM model that outperformed the other models in terms of accuracy based on RMSE scores and is ranked 2nd in training execution time.

In summary, in Section 8, we have conducted comprehensive analysis to discover the best model, in terms of speed and performance, to use for our main pipeline (Section 2-7). This contributed to the training and development of our models.

9 Conclusion

In this project, we have conducted comprehensive studies and experiments in feature engineering, feature selection, model selection, and hyperparameter tuning with Optuna. Furthermore, we have utilised unique techniques to enhance model performance, such as generating node2vec embeddings, and utilising anomaly detection. Lastly, we adopted the best practices like cross validation, early stopping, and target leakage prevention. Our final model was within the top 50% of the private leaderboard.

In the future, we could explore other creative ways of feature engineering to improve data quality as we believe data quality and quantity is the main source of performance gain here. Furthermore, we could utilise bigger models like stacking to combine the predictions of Catboost, XGboost, and LightGBM.

References

- [1] [VHRRanger/nodevectors: Fastest network node embeddings in the west \(github.com\)](https://github.com/VHRRanger/nodevectors)
- [2] [Welcome to LightGBM's documentation! — LightGBM 3.3.2 documentation](https://lightgbm.ai/docs/3.3.2/)

Appendix

We append all the engineered features here for reference.

old_hist_count, old_authorized_flag_0_count, old_authorized_flag_1_count,
old_authorized_flag_0_fraction, old_authorized_flag_1_fraction, old_category_1_0_count,
old_category_1_1_count, old_category_1_0_fraction, old_category_1_1_fraction,
old_category_2_1_0_count, old_category_2_2_0_count, old_category_2_3_0_count,
old_category_2_4_0_count, old_category_2_5_0_count, old_category_2_1_0_fraction,
old_category_2_2_0_fraction, old_category_2_3_0_fraction, old_category_2_4_0_fraction,
old_category_2_5_0_fraction, old_category_3_A_count, old_category_3_B_count,
old_category_3_C_count, old_category_3_A_fraction, old_category_3_B_fraction,
old_category_3_C_fraction, old_installments_mean, old_month_lag_mean,
old_purchase_amount_mean, old_installments_max, old_month_lag_max,
old_purchase_amount_max, old_installments_median, old_month_lag_median,
old_purchase_amount_median, old_installments_std, old_month_lag_std,
old_purchase_amount_std, old_installments_skew, old_month_lag_skew,
old_purchase_amount_skew, old_installments_mad, old_month_lag_mad,
old_purchase_amount_mad, old_installments_sum, old_month_lag_sum,
old_purchase_amount_sum, old_purchase_date_month_diff_mean,
old_purchase_date_month_diff_median, old_purchase_date_month_diff_std,
old_purchase_date_month_diff_max, old_purchase_date_month_diff_min,
old_purchase_date_month_diff_sum, old_month_mean, old_month_min, old_month_max,
old_hour_nunique, old_hour_mean, old_hour_min, old_hour_max, old_weekofyear_nunique,
old_weekofyear_mean, old_weekofyear_min, old_weekofyear_max, old_weekday_mean,
old_weekday_min, old_weekday_max, old_day_nunique, old_day_mean, old_day_min,
old_day_max, old_weekend_mean, old_price_mean, old_price_max, old_price_min,
old_Christmas_Day_2017_mean, old_Children_day_2017_mean,
old_Black_Friday_2017_mean, old_Mothers_Day_2018_mean, old_duration_mean,
old_duration_min, old_duration_max, old_duration_var, old_duration_skew,
old_amount_month_ratio_mean, old_amount_month_ratio_min, old_amount_month_ratio_max,
old_amount_month_ratio_var, old_amount_month_ratio_skew, old_purchase_date_diff,
old_purchase_date_uptonow, old_purchase_date_uptomin, new_hist_count,
new_category_2_1_0_count, new_category_2_2_0_count, new_category_2_3_0_count,
new_category_2_4_0_count, new_category_2_5_0_count, new_category_2_1_0_fraction,
new_category_2_2_0_fraction, new_category_2_3_0_fraction, new_category_2_4_0_fraction,
new_category_2_5_0_fraction, new_category_3_A_count, new_category_3_B_count,
new_category_3_C_count, new_category_3_A_fraction, new_category_3_B_fraction,
new_category_3_C_fraction, new_installments_mean, new_month_lag_mean,
new_purchase_amount_mean, new_installments_max, new_month_lag_max,
new_purchase_amount_max, new_installments_median, new_month_lag_median,
new_purchase_amount_median, new_installments_std, new_month_lag_std,
new_purchase_amount_std, new_installments_mad, new_month_lag_mad,
new_purchase_amount_mad, new_installments_sum, new_month_lag_sum,
new_purchase_amount_sum, new_month_mean, new_month_min, new_month_max,
new_hour_nunique, new_hour_mean, new_hour_min, new_hour_max,
new_weekofyear_nunique, new_weekofyear_mean, new_weekofyear_min,
new_weekofyear_max, new_weekday_mean, new_weekday_min, new_weekday_max,
new_day_nunique, new_day_mean, new_day_min, new_day_max, new_weekend_mean,
new_price_mean, new_price_max, new_price_min, new_Christmas_Day_2017_mean,

new_Children_day_2017_mean, new_Black_Friday_2017_mean,
new_Mothers_Day_2018_mean, new_duration_mean, new_duration_min, new_duration_max,
new_duration_var, new_amount_month_ratio_mean, new_amount_month_ratio_min,
new_amount_month_ratio_max, new_amount_month_ratio_var, new_purchase_date_diff,
new_purchase_date_uptonow, new_purchase_date_uptomin, feature_1, feature_2, feature_3,
quarter, elapsed_time, days_feature1, days_feature2, days_feature3, days_feature1_ratio,
days_feature2_ratio, days_feature3_ratio, feature_sum, feature_mean, feature_max,
feature_min, feature_var, feature1_1, feature1_2, feature1_3, feature1_4, feature1_5,
feature2_1, feature2_2, feature2_3, feature3_0, feature3_1, embedding_merchant_id_0,
embedding_merchant_id_1, embedding_merchant_id_2, embedding_merchant_id_3,
embedding_merchant_id_4, embedding_merchant_id_5, embedding_merchant_id_6,
embedding_merchant_id_7, embedding_merchant_id_8, embedding_merchant_id_9,
embedding_merchant_id_10, embedding_merchant_id_11, embedding_merchant_id_12,
embedding_merchant_id_13, embedding_merchant_id_14, embedding_merchant_id_15,
embedding_merchant_group_id_0, embedding_merchant_group_id_1,
embedding_merchant_group_id_2, embedding_merchant_group_id_3,
embedding_merchant_group_id_4, embedding_merchant_group_id_5,
embedding_merchant_group_id_6, embedding_merchant_group_id_7,
embedding_merchant_group_id_8, embedding_merchant_group_id_9,
embedding_merchant_group_id_10, embedding_merchant_group_id_11,
embedding_merchant_group_id_12, embedding_merchant_group_id_13,
embedding_merchant_group_id_14, embedding_merchant_group_id_15,
embedding_merchant_category_id_0, embedding_merchant_category_id_1,
embedding_merchant_category_id_2, embedding_merchant_category_id_3,
embedding_merchant_category_id_4, embedding_merchant_category_id_5,
embedding_merchant_category_id_6, embedding_merchant_category_id_7,
embedding_merchant_category_id_8, embedding_merchant_category_id_9,
embedding_merchant_category_id_10, embedding_merchant_category_id_11,
embedding_merchant_category_id_12, embedding_merchant_category_id_13,
embedding_merchant_category_id_14, embedding_merchant_category_id_15,
embedding_subsector_id_0, embedding_subsector_id_1, embedding_subsector_id_2,
embedding_subsector_id_3, embedding_subsector_id_4, embedding_subsector_id_5,
embedding_subsector_id_6, embedding_subsector_id_7, embedding_subsector_id_8,
embedding_subsector_id_9, embedding_subsector_id_10, embedding_subsector_id_11,
embedding_subsector_id_12, embedding_subsector_id_13, embedding_subsector_id_14,
embedding_subsector_id_15, embedding_city_id_0, embedding_city_id_1,
embedding_city_id_2, embedding_city_id_3, embedding_city_id_4, embedding_city_id_5,
embedding_city_id_6, embedding_city_id_7, embedding_city_id_8, embedding_city_id_9,
embedding_city_id_10, embedding_city_id_11, embedding_city_id_12, embedding_city_id_13,
embedding_city_id_14, embedding_city_id_15, embedding_state_id_0, embedding_state_id_1,
embedding_state_id_2, embedding_state_id_3, embedding_state_id_4, embedding_state_id_5,
embedding_state_id_6, embedding_state_id_7, embedding_state_id_8, embedding_state_id_9,
embedding_state_id_10, embedding_state_id_11, embedding_state_id_12,
embedding_state_id_13, embedding_state_id_14, embedding_state_id_15,
nunique_merchant_id, count_merchant_id, nunique_count_frac_merchant_id,
nunique_merchant_group_id, count_merchant_group_id,
nunique_count_frac_merchant_group_id, nunique_merchant_category_id,
count_merchant_category_id, nunique_count_frac_merchant_category_id,
nunique_subsector_id, count_subsector_id, nunique_count_frac_subsector_id, nunique_city_id,
count_city_id, nunique_count_frac_city_id, nunique_state_id, count_state_id,

nunique_count_frac_state_id, city_id_similarity_min, city_id_similarity_max,
city_id_similarity_sum, city_id_similarity_mean, city_id_similarity_std,
merchant_category_id_similarity_min, merchant_category_id_similarity_max,
merchant_category_id_similarity_sum, merchant_category_id_similarity_mean,
merchant_category_id_similarity_std, merchant_group_id_similarity_min,
merchant_group_id_similarity_max, merchant_group_id_similarity_sum,
merchant_group_id_similarity_mean, merchant_group_id_similarity_std,
merchant_id_similarity_min, merchant_id_similarity_max, merchant_id_similarity_sum,
merchant_id_similarity_mean, merchant_id_similarity_std, state_id_similarity_min,
state_id_similarity_max, state_id_similarity_sum, state_id_similarity_mean,
state_id_similarity_std, subsector_id_similarity_min, subsector_id_similarity_max,
subsector_id_similarity_sum, subsector_id_similarity_mean, subsector_id_similarity_std,
old_purchase_date_max_month, old_purchase_date_max_year,
old_purchase_date_min_month, old_purchase_date_min_year,
new_purchase_date_max_month, new_purchase_date_max_year,
new_purchase_date_min_month, new_purchase_date_min_year, first_active_month_month,
first_active_month_year