Report for PRML Nov-22 data contest

Team name: Natural Neural Networks
Balakrishnan A (CS20B012)
Hemesh DJ (CS20B031)

Contents

1 Feature Engineering 1
1.1 Consolidating sequence entries 1
1.2 Transforming DateTime fields 1
1.3 Encoding categorical data 2
1.4 Final steps 2

2 Modelling the data 2
2.1 Regression vs classification 2
2.2 Histogram-Based Gradient Boosting 2
2.3 Hyperparameter tuning 3

3 Results 3

1 Feature Engineering

There were three significant feature usage problems that required handling:

- Ensuring uniqueness of key *booking_id* in the tables *bookings_data* and *payments_data* by consolidating data from sub-bookings and a sequence of payments into a single entry
- Obtaining useful information from the DateTime fields to use as new fields
- Encoding categorical data

These were addressed as follows:

1.1 Consolidating sequence entries

In the table *payments_data*, each *booking_id* could have multiple payments done at different stages of the booking due to which the *booking_id* field did not have unique values throughout the table. This was handled by consolidating the data from the different rows with the same *booking_id* into a single row with different fields to summarise. The field *payment_sequential*, which specified the payment number in the sequence of payments, became *payments_made*, conveying the total length of the sequence of payments. The fields *payment_type* and *payment_installments* were discarded as their inclusion post-modification did not add useful information. The field *payment_value* became *payments_value* and described the sum of the values in the multiple rows with the same key.

Similarly, in the table *bookings_data*, the key *booking_id* was not unique due to the possibility of sub-bookings within a booking. In this case, the *booking_sequence_id*, which represented the sub-booking number, was replaced by the field *sub_requests*, specifying the number of sub-bookings within the booking. The other fields were not modified and were simply filled with the data from the last entry containing the same *booking_id*.

1.2 Transforming DateTime fields

Since DateTime data cannot be interpreted as numerical data and lose meaning when used as categorical data, they need to be transformed to meaningful features in order to be usable.

The field *booking_create_timestamp* of the table *bookings* was used as the origin date for each row and the other DateTime fields were transformed as their offset from it (either in minutes or in days depending on the field). The month and year of request creation (*booking_create_timestamp*) and checkin (*booking_checkin_customer_date*) were also obtained and stored in 4 new columns, and 2 more columns were created to check whether either month is August (8), as it turned out to be the most popular month for booking and checkin. A field *is_checkin* was made based on whether or not *booking_checkin_customer_date* was present (i.e., not missing), attempting to interpret missing data as failure to checkin. Another column *quick_approve* was made to convey whether the approval time (*booking_approved_at* - *booking_create_timestamp*) was less than 1 day.

1.3 Encoding categorical data

Several fields contained categorical data in the form of strings, and in order to make better use of them, they were encoded into integer data by hashing the unique values in each categorical column to the first few natural numbers (as many as needed for distinct mapping) and using it to extract the required integer replacement for each category.

1.4 Final steps

Following feature engineering, complete train and test data were obtained by merging the different tables, and the missing values (now that all columns contained numerical data) were replaced by -1. The targets for train data were separated, and columns *booking_id* and *customer_id* were dropped.

2 Modelling the data

2.1 Regression vs classification

Given the evaluation metric of *mean_squared_error* (a typical regression metric instead of a classification metric) and the fact that the target (being a rating) retains its meaning when treated as a continuous variable, regressors vastly outperformed classifiers owing to their ability to show their confusion between 2 classes via a value between the two rather than an all-or-nothing choice.

2.2 Histogram-Based Gradient Boosting

A Histogram-Based Gradient Boosting Regressor was used to fit the data. Gradient boosting is a generalized boosting technique, using an ensemble of weak prediction models, typically decision trees. A major problem of gradient boosting is that it is slow in training since efficiently building the decision trees requires sorting the features at every node of the tree. Training the trees can be hastened by discretizing (binning) the continuous input variable to a few unique values, making use of histograms that only need to be made once for the entire boosting process. Gradient boosting ensembles that implement this technique and attune the training algorithm around the transformed input variables are referred to as Histogram-Based Gradient Boosting Ensembles.

Python's machine learning library, scikit-learn, provides a *HistGradientBoostingRegressor* class that supports the histogram technique. By default, the ensemble uses 255 bins for each input feature, and this can be set via the *max_bins* argument. Setting this to smaller values may result in further efficiency improvements, although perhaps at the cost of some model skill. The number of trees can be set via the *max_iter* argument and defaults to 100.

2.3 Hyperparameter tuning

Cross-validation was performed majorly on the parameters *max_iter*, *max_depth*, *max_leaf_nodes* and *max_bins*. *max_leaf_nodes* showed best performance at a value of 63 and *max_bins* at its default value of 255. The first two parameters produced nearly constant validation errors for the chosen range of hyperparameters, i.e.,

$$max\_iter \in [100, 200, 500, 1000, 5000]$$
$$max\_depth \in [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]$$

Hence, all 50 possible combinations of these parameters were used and the average of the predictions by these models was chosen to be the final prediction. Since possible true rating scores are only one of $\{1, 2, 3, 4, 5\}$, any value below 1 or above 5 was rounded off to 1 or 5 respectively.

3 Results

A train-validation split of 80-20 percent was used and the model trained on the 80% produced the following results:

$$\text{Train error} = 1.22980$$
$$\text{Validation error} = 1.35996$$

The final model was then trained on the whole train data and produced the results as follows.

$$\text{Train error} = 1.25220$$
$$\text{Test error (public)} = 1.37045$$

There is some amount of overfitting since the *DecisionTreeRegressor* in itself does not do a bad job in fitting the data, with a *max_depth* of even 2 producing a *mean_squared_error* of less than 1.5, providing the boosting algorithm with a not-so-weak learner, but nonetheless it performs satisfactorily and produces a test error close to the validation error.