Future Sales Prediction

Abstract

This paper proposes an ensembling of decision tree model based future sales prediction. In particular, we present a detailed description of feature engineering necessary to generate trainable parameters and then perform detailed model performance study with various state of the art decision tree based models. **Our best performing model resulted in a RMSE of 0.87605 achieving a class rank of 6 and Global Kaggle leaderboard rank of 62**.

1 Introduction

In this paper we present detailed description and evaluation of the Kaggle future sales prediction challenge. The remaining of the paper is oriented as follows: In Section 2 and 3, we present details of data pre-processing. Section 4 elaborates about feature engineering to generate the trainable parameters. Section 5 describes the model description of the tree based models which were used. We present detailed results in Section 6 before concluding in Section 7.

2 Data description

Training data given in *train.csv* has 2935849 rows and has 6 columns: *date*, *date_block_num*, *shop_id*, *item_id*, *item_price*, *item_cnt_day*. The date range is from 1st January, 2013 to October 2015. The month number is listed from 0 to 33 in date_block_num. Each row is associated with a per day sale of a specific item along with its shop id. The per day sale of each item in a shop wise mannner (per shop id wise) is listed under *item_cnt_day*. Shop information is provided for 60 shops in the form of shop name and shop id in *shops.csv*. Details about items are provided in *items.csv* regarding *item_id*, *item_name* and *item_category_id*. There are total 22170 items. Further details about the item categories are provided in *item_categories.csv* in the form of item_category_name and item_category_id. The goal is to predict the monthly sales for each item-shop id pair in test set.

3 Data-Preprocessing

The data preprocessing part consists of modifying the outlier elements and merging duplicate information. For cleaning the outlier elements, we consider the fields *item_cnt_day* and *item_price*. We plot the distribution of *item_cnt_day* and find the number of entries having values less than zero. The distribution of entries less than zero are shown in Figure 1(b)-(c). From Fig 1(b)-(c), we can see that there are 7356 entries with *item_cnt_day* values less than zero. The negative values span from -22 to -1. We fix these 7356 entries from training data by setting the *item_cnt_day* values to 0. Since *item_price* cannot be negative, we keep only those entries whose *item_price* values are positive.

From the given shop data information in *shops.csv*, we have details of 60 shops with their shop names and ids. From the shop name information, we can see that there are multiple cases where shops with

(a) (b) (c)

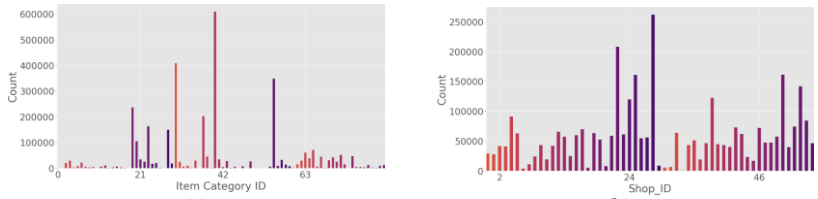
Figure 1: (a) Example of same shop names for different shop ids, (b) Distribution and (c) per element count of negative values of feature *item_cnt_day*

same names have different shop ids. We merge those entries in training data. We can see from Fig 1(a) that shop id pairs $(0,\ 57)$ and $(1,\ 58)$ have the same shop names. Similar trend is observed for shop id pairs $(10, 11)$ and $(40, 39)$. Training entries for same shop names are merged together.

4 Feature Engineering

4.1 Data Exploration

In order to find the distribution of data across different shops and item categories, we plot the monthly item count with different item category ids (84 in total) and shop ids (60 in total). The monthly count distributions are shown in Fig 2(a) and Fig 2(b), respectively.



(a) (b)

Figure 2: (a) Distribution of monthly count with item category id. 84 item categories are listed in x axis. (b) Distribution of monthly item count with shop id. 60 shop ids are listed in x axis.

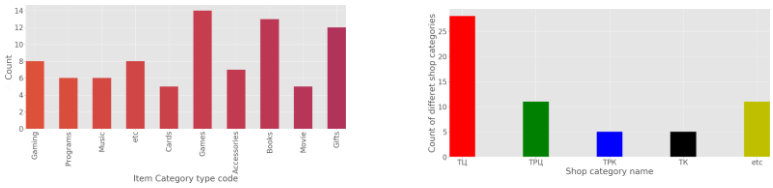
4.2 Item category and name based features

In order to extract information from the item category data given in *item_categories.csv*, we split the *item_category_name* into fields *subtype* and *type*. While splitting using the delimiter " ", we consider the first word obtained from split as the *type*. We further filter out the field *type* by considering only those *type* entries whose number of occurrences are greater than 4. The *type* entries whose number of occurrences are less than 4 are grouped together in one field called 'etc'. In order to extract the *subtype*, the *item_category_name* is split using the delimiter "-" and the second split is considered as the *subtype*.

We also extract features related to item name from the *item_name* field in *items.csv*. We split the *item_name* using two delimiters "[" and "(" into fields *name2* and *name3* respectively. We further extract the item type information from *name2* section. We then find the count of each item type from 22170 item ids available in *items.csv*. Using the item type information, we group those entries in *name2* together whose counts are less than 40.

4.3 Shop based features

To extract shop based features, we use the information given in *shops.csv*. Each entry in *shop_name* column is split into *shop_category* and *shop_name*. We split the *shop_name* based on " " delimiter and use the first split string as city and the subsequent one as the shop category. We broadly group the shop categories into 5 segments based on number of occurrences. The distribution of shop categories are listed in Fig 3a. In Fig 3b, the category names are listed in Russian.



(a) (b)

Figure 3: (a) Distribution of type codes of various 84 item categories. The type codes are translated from Russian to English. (b) Distribution of shop categories among the 60 shops listed in **shops.csv**. Shop category names are listed in Russian.

4.4 Mean encoded features and lag feature generation

For generating the training data, we first consider each valid shop id-item id pair for each month. This is done to make the training data similar to the testing data where predictions have to be done for each shop-item id pair. In the above shop id-item id based training data, we add average statistics for various combinations. For each month, we first extract the average item count as *date_avg_item_cnt*. We extract the average monthly item count features based on item id (*date_item_avg_item_cnt*), shop id (*date_shop_avg_item_cnt*), shop id and item id pair(*date_shop_item_avg_item_cnt*), shop id and subtype code (*date_shop_subtype_avg_item_cnt*), shop city(*date_city_avg_item_cnt*) and shop city and item id pair(*date_item_city_avg_item_cnt*). In standard time-series forecasting problem, features from prior time-steps help in capturing the temporal dynamics in future. After extracting the mean encoded features, we pass those features through a lag generation pipeline for adding prior information for the current month. The features from previous time steps are included as *lag_i*(feature from ith previous month, i=1,2,3).

4.5 Other features

We also use revenue based feature where revenue is computed as the product of *item_cnt_day* and *item_price*. For each month, we compute the total *shop_id* wise revenue as *date_shop_revenue*. From the *date_shop_revenue* estimates, we compute shop-id wise average revenue as *shop_avg_revenue*. The *delta_revenue* feature is computed as the deviation of the *date_shop_revenue* from *shop_avg_revenue* for that month. We pass *delta_revenue* through the lag generation pipeline to generate *delta_revenue_lag* feature. Auxiliary information is also added in terms of first month of sale for each shop and item id pair(*item_shop_first_sale*) and item id (*item_first_sale*). The *item_price* trend information is added in terms of *delta_price_lag*, where we measure earlier month's deviation calculated as the difference between mean price of each item id and their aggregated mean price. We also add the number of days for each month as a feature and the actual month number(*date_block_num* mod 12)

4.6 Final feature set

We encode all the categorical data like *name_2*,*name_3*,*type_code* etcinto integer features using the **Label Encoder** utility in *scikit-learn*. For the final set of experiments, we use a set of 32 features for training our model, shown in Table 1. We consider the data from the 3rd month to 32nd month as our training set (9106486 samples). The data from 33rd month(238172 samples) is taken as the validation set.

Feature type	Feature set
Preprocessing based feature	date_block_num, Shop_id, Item_id, Shop_category,Shop_city, Item_category_id, name2, name3, subtype_code, type_code
Lag based feature	item_cnt_month_lag_1,item_cnt_month_lag_2, item_cnt_month_lag_3, date_avg_item_cnt_lag_1, date_item_avg_item_cnt_lag_1,date_item_avg_item_cnt_lag_2, date_item_avg_item_cnt_lag_3, date_shop_avg_item_cnt_lag_1 date_shop_avg_item_cnt_lag_2, date_shop_avg_item_cnt_lag_3, date_shop_item_avg_item_cnt_lag_1, date_shop_item_avg_item_cnt_lag_2 date_shop_item_avg_item_cnt_lag_3, date_shop_subtype_avg_item_cnt_lag_1, date_city_avg_item_cnt_lag_1,date_item_city_avg_item_cnt_lag_1, delta_revenue_lag_1, delta_price_lag
Other feature	month, days, item_shop_first_sale,item_first_sale

Table 1: List of final 32 features used in training the model. The features are grouped by their types: Preprocessing based, Lag based and Other types(month,day, day of first sale information)

5 Model Selection

This section describes the models which we have used for our future sales prediction. In particular, we have successfully used three decision tree based models namely, *XGBoost*, *random forest*, and *light GBM* to get competitive performance. The models are described in the following subsections.

5.1 XGBoost

XGBoost [1] stands for extreme Gradient Boosting. It is an open source library providing a high-performance implementation of gradient boosted decision trees (GBDT). The implementation of the algorithm was engineered for efficiency of compute time and memory resources. A major design goal is to make the best use of available resources to train the model. Some key algorithm implementation features include: (a) Sparsity Aware implementation with automatic handling of missing data values. (b) block Structure to support the parallelization of tree construction. To learn the set of functions used in the model, XGBoost minimizes the following regularized objective.

$$L(\varphi) = \sum_i l(\hat{y}_i, y_i) + \sum_k \Omega(f_k) \tag{1}$$

$$\Omega(f) = \gamma T + \frac{1}{2}\lambda ||w||^2 \tag{2}$$

Here l is a differentiable convex loss function that measures the difference between the prediction $\hat{y}_i$ and the target y_i. The second term Ω penalizes the complexity of the model (i.e., the regression tree functions). The additional regularization term helps to smooth the final learnt weights to avoid over-fitting.

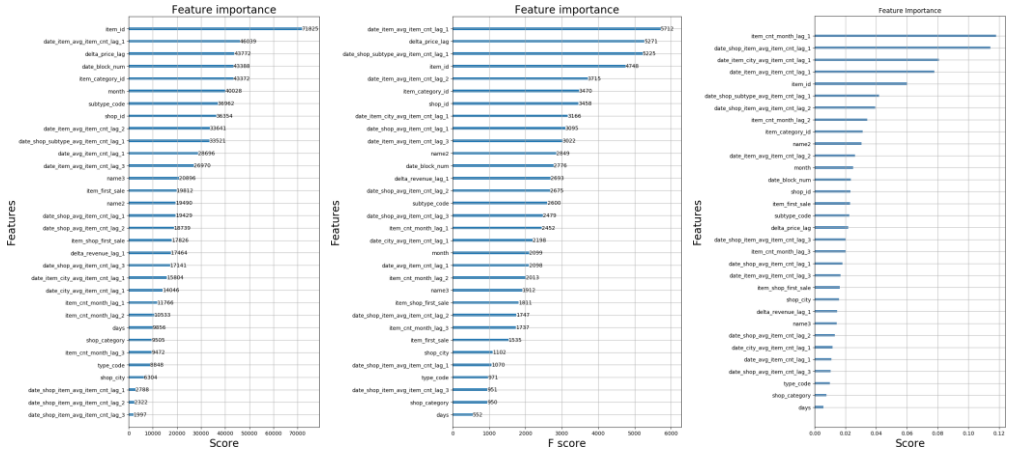
5.2 Random Forest

The random forest is a model made up of many decision trees. Rather than just simply averaging the prediction of trees (which we could call a "forest"), this model uses two key concepts that gives it the name random, and are described next. 1. **Random Sampling of Training Observations** When training, each tree in a random forest learns from a random sample of the data points. The samples are drawn with replacement, known as bootstrapping, which means that some samples will be used multiple times in a single tree. The idea is that by training each tree on different samples, although each tree might have high variance with respect to a particular set of the training data.Overall, the entire forest will have lower variance but not at the cost of increasing the bias.At test time, predictions are made by averaging the predictions of each decision tree. 2. **Random Subsets of features for splitting nodes** The other main concept in the random forest is that only a subset of all the features are considered for splitting each node in each decision tree. Generally this is set to sqrt(*n_features*) for classification meaning that if there are 16 features, at each node in each tree, only 4 random features will be considered for splitting the node. (The random forest can also be trained considering all the features at every node as is common in regression.

5.3 Light GBM

For every feature,conventional implementations of GBDT scan all the data instances to estimate the information gain of all the possible split points. Therefore, their computational complexities will be proportional to both the number of features and the number of instances. This makes these implementations very time consuming when handling big data. To tackle this issue Light GBM (LGBM) [2] uses two novel techniques, namely *Gradient-based One-Side Sampling* (GOSS) and *Exclusive Feature Bundling* (EFB). In GOSS technique, when down sampling the data instances, in order to retain the accuracy of information gain estimation, it is recommended to keep those instances with large gradients (e.g., larger than a pre-defined threshold, or among the top percentiles), and only randomly drop those instances with small gradients. EFB converts feature binding problem in sparse feature space, to a graph coloring problem (by taking features as vertices and adding edges for every two features if they are not mutually exclusive), and solving it by a greedy algorithm with a constant approximation ratio. LGBM splits the tree leaf wise with the best fit whereas other boosting algorithms split the tree depth wise or level wise rather than leaf-wise. So when growing on the same leaf in Light GBM, the leaf-wise algorithm can reduce more loss than the level-wise algorithm and hence results in much better accuracy which can rarely be achieved by any of the existing boosting algorithms. Also, it is surprisingly very fast, hence the word 'Light'.

Fig. 4 shows the relative importance of features for different decision tree models.



(a) (b) (c)

Figure 4: Comparison of feature wise importance for (a) LGBM, (b) XGBoost, and (c) random forest.

6 Model Evaluation

6.1 Results

Table 2 describes the performance on the validation and test set for our XGBoost, random forest and LGBM based decision-tree based models. Also, here we present the model performances with weighted ensembling of different models. As we can see the best performing model in terms of both validation RMSE and test RMSE (leader board score) is the ensembling model of LGBM and random forest with weight factor of 0.5 each. It is quite intuitive as ensembling tries to reduce the random noise of the classification by balancing between the two models and thus can predict in a better way. **It is noteworthy that our best performing model provides a leaderboard RMSE of 0.87605 which achieved a class rank of 6 and Global rank of 62 in Kaggle.**

Method	Tuning type	Validation RMSE	Leaderboard score
XGBoost	naive (n)	0.898755	0.9075
	grid-search (gs)	0.892129	0.89044
	hyperopt (h)	0.891743	NE
Random-forest	naive (n)	0.896694	0.88198
	grid-search (gs)	0.89587	0.87957
LGBM	naive (n)	0.904814	NE
	grid-search (gs)	0.88636	NE
	hyperopt (h)	0.8823	NE
Weighted ensemble			
0.5*XGBoost(h) + 0.5*random-forest(gs)		0.887585	0.87808
0.5*LGBM(gs) + 0.5*random-forest(gs)		0.883526	**0.87605**[1]
0.7*random-forest(gs) + 0.2*LGBM(gs) + 0.1*XGBoost(h)		0.887478	0.8786

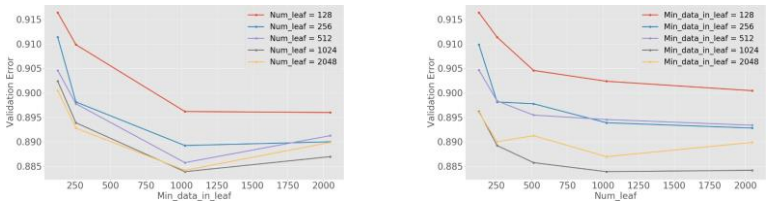
Table 2: Validation RMSE and leaderboard scores for different models with different form of hyper-parameter tuning schemes. Here NE=not evaluated.

6.2 Insights

Here we first present our insights from the models which worked better for the proposed problem statement before summarizing the possible loop-holes for the failure of other models which did not provide competitive RMSE.

6.2.1 Models Which Performed Well

Hyperparameter selection of the model to be trained plays a key role in the final performance. Here we present some empirically achieved insights to guide proper training with the selected models. Fig. 6(a) shows validation RMSE vs hyper-opt trials for XGBoost training. We see that the best performing hyperparameterized model is found within 100 trial runs. Thus we can infer that 100 hyper-opt trials are enough for us to provide the best suited hyperparameter settings. In LGBM based decision tree training two hyperparameters play key role for the model to not over-fit: 1. *min data in*



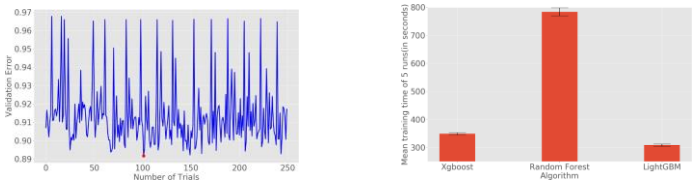
(a) (b)

Figure 5: Val RMSE performance of LGBM [3] as a function of two major hyperparameters (a) min data in leaf and (b) num leaf .

leaf and 2. *num leaf*. We performed parameter sweep of one keep the other fixed to some value. As shown in Fig. 5(a) we achieve the best performing validation RMSE at *min_data_in_leaf* value of 1000. We found the best value of *num_leaf* 1000 shown in Fig. 5(b).

6.2.2 Models Which Did Not Performed Well

Apart from the various decision-tree based models we also tried to formulate the prediction problem as multi-layer perceptron (MLP) based regression problem. For the MLP we had chosen a model having similar trainable parameters as the total number of training examples to avoid over or under-fitting. Also, as there were various types of input variables having various range of values (for example, the months are always in the range of 1 to 33, where as the item category wise count can be as large as 600000) we choose to *min-max* normalize Our objective was to minimize the *mean square error* loss. However, after training the test RMSE could not reach the one achievable through the decision tree based approaches. In particular, we recon few issues might be the reason for MLP to fail: 1. incorrect normalization, 2. presence of both int and float type features, 3. improper selection of activation and loss function. Another model we tried that did not work is support vector regressor (SVR). We used sci-kit learn package of SVR. Two major reason to avoid SVR for the sales prediction problem are: 1. Choice of kernel is tricky (we chose RBF kernel for our experiments), 2. SVR runs extreme slow.



(a) (b)

Figure 6: (a)Variation of validation error with hyper-opt trials.The trial giving the lowest validation error is considered and marked here with a red dot. The hyperparameter corresponding to that trial is selected as the best setting for XGBoost (b) Plot of mean training time (mean of 5 runs) for XGBoost, random forest, LGBM. Standard deviation of run times denoted by black bars.

7 Conclusions

In this work we presented future sales prediction models based on decision tree structures. Our evaluation showed the best performing model can be achieved through ensembling of LGBM and random-forest giving equal weight to each.

References

[1] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIG-KDD international conference on knowledge discovery and data mining*, pages 785–794, 2016.

[2] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. Lightgbm: A highly efficient gradient boosting decision tree. In *Advances in neural information processing systems*, pages 3146–3154, 2017.

[3] Guolin Ke et al. lightgbm.readthedocs.io. In *https://lightgbm.readthedocs.io/en/latest/Features.html*, 2017.