Project Report for Machine Learning and Time Series

Jiachuan Bi
Peking University
bichuan1996@pku.edu.cn

Haiteng Zhao
Peking University
zhaohaiteng@pku.edu.cn

Guanqi Zhu
University of Science and Technology of China
zgq@mail.ustc.edu.cn

Qinghong Han
Peking University
1600014116@pku.edu.cn

Abstract

There are many methods to deal with the *Predict Future Sales* project, among which the xgboost is known to perform best except neural network. Xgboost is a very efficient way about regression and classification problems, based on the combination of regression or classification tree and gradient boosting method. Therefore, we use xgboost as our main framework to make predictions. Before we start, pre-processing should be done to get rid of anomalous data, for which we will extract and filter features from the data. The key step is to adjust parameters. In this project, we adjust several parameters of `XGBRegressor` to gradually get better performance.

1 Introduction

Kaggle is a platform which provides all varieties of competitions for data science and machine learning. Among those *Predict Future Sales* is a competition which needs us to predict the sales for items according to some given past data. There are many ways to deal with it, one is to regard it as a time series problem, for which we can use AR or MA etc., and another is to see it as a regression problem, for which many machine learning methods can be utilized. Empirically, machine learning methods perform better than usual time series methods on this kind of problem, so we use xgboost as our framework.

Xgboost is a very powerful method to deal with regression or classification problems, which combines tree structure and gradient boosting method to efficiently make predictions. However, the key to using this method is to properly choose parameters to fit data best. We directly call `XGBRegressor` to use xgboost and try to adjust several parameters such that better performance can be got.

On the other hand, features can be a very crucial factor which will make a deep impact on the quality of the model. So, it's important to deal well with the raw data, along with extracting and selecting features carefully.

Our work will focus on these two aspects, and we'll give a detailed introduction in the following parts.

2 Related Work

In our project, the basic algorithm we used is boosting regression tree. Boosting is one of the basic thought in machine learning. This thought appeared in the wake of the concept of strongly learnable and weakly learnable. Strongly learnable means that in a probably approximately correct learnable question, there is an learning algorithm that can achieve a high correct rate by learning polynomial training samples; while weakly learnable only requires existence of a algorithm that can achieve a

correct rate better than randomly guessing by learning polynomial training samples. Boosting thought came from such a proposition: a question is weakly learnable if and only if this question is strongly learnable. This proposition led people to pursue method which can boost a weak learning algorithm to a strong learning algorithm. That's how boosting thought came out.

After many years, researchers have raised many boosting algorithms. The basic thought of these algorithms is to combine several weak learning algorithms, so that the combination can perform better than any of them. AdaBoost is one of the most famous boosting algorithms. The concrete boosting method we used for training will be discussed in Section 3.2

3 Methodology

3.1 Feature Extracting

The main idea of our preprocessing follows the wonderful kernel *Feature engineering, xgboost.*[1]. In Section 4.1 we will discuss how to improve this process and compare different results. Here we just introduce a general frame of feature extracting.

First it's essential to wash the data. In order to rule out the outliers, we draw boxplots for **item_cnt_day** and **item_price**. Setting maximum at 1000000 and 1001 in Figure1 1, we abandon outliers and get more reasonable data. Also we use median of **item_price** to replace the negative values of price.

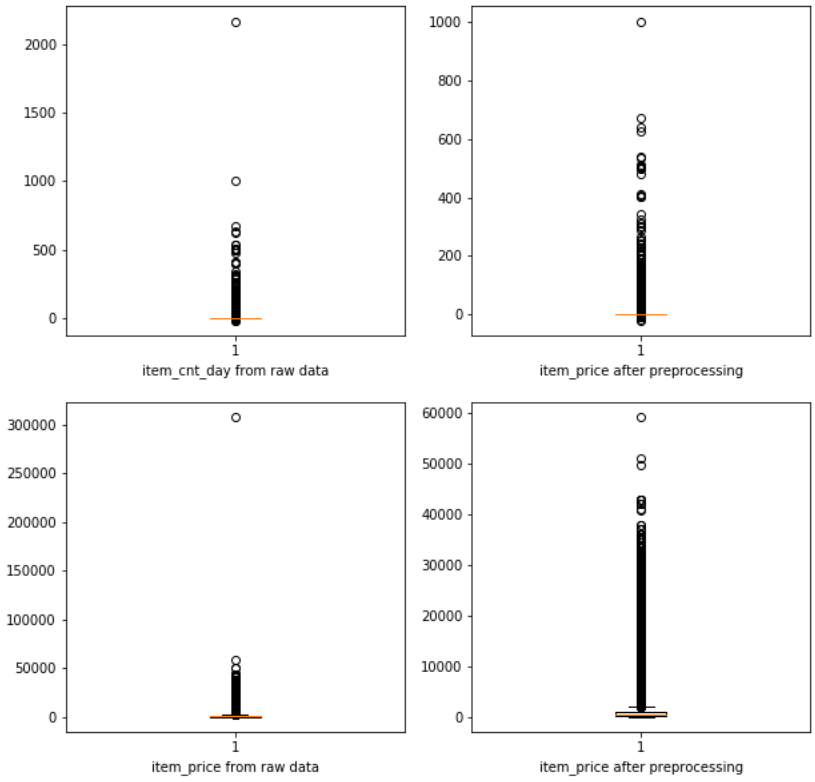

Figure 1: Preprocessing for **item_cnt_day** and **item_price**

[1]https://www.kaggle.com/dlarionov/feature-engineering-xgboost

Table 1: Feature set

Name	Description	Lag
date_block_num	Integer to represent time	None
shop_id	Integer to represent shop	None
item_id	Integer to represent item	None
city_code	Integer to represent city	None
item_category_id	Integer to represent item category	None
type_code	Integer to represent item type	None
subtype_code	Integer to represent item subtype	None
item_cnt_amount	sales of item	1, 2, 3, 6
item_avg_item_cnt	Average sales among all kinds of items and all shops	1
date_item_avg_item_cnt	Average sales of a particular item among all shops	1, 2, 3, 6
date_shop_avg_item_cnt	Average sales of a particular shop	1, 2, 3, 6
date_cat_avg_item_cnt	Average sales of a particular item category	1
date_shop_cat_avg_item_cnt	Average sales of a particular item category in a particular shop	1
date_shop_type_avg_item_cnt	Average sales of a particular item type in a particular shop	1
date_shop_subtype_avg_item_cnt	Average sales of a particular item subtype in a particular shop	1
date_city_avg_item_cnt	Average sales of a particular item type in a particular city	1
date_item_city_avg_item_cnt	Average sales of a particular item in a particular city	1
date_type_avg_item_cnt	Average sales of a particular item type	1
date_subtype_avg_item_cnt	Average sales of a particular item subtype	1
delta_price_lag	Change of price between last two month	1
delta_revenue	Change of shop's revenue between last two month	1
month	Number of month	None
days	Number of days in a particular month	None
item_shop_last_sale	The month of last sales record in a particular shop	None
item_last_sale	The month of last sales record	None
item_shop_first_sale	The month of first sales record in a particular shop	None
item_first_sale	The month of first sales record	None

Second we encode shops and items. For shops, we can extract city information from their name, so we can define **shop_id** and **city_code** as integar. Similarly for items, we extract type and subtype according to its name. Thus we get **item_id**, **type_code** and **subtype_code**.

For each (shop,item) pair, there are 33 months' data for us to develop prediction model. We train our model on the train set with 32 months' data and month 33 is used for validation, then we make prediction on month 34. After taking sum of **item_cnt_day** in each month, we fill invalid value with zero and impose restriction on **item_cnt_month**(monthly sales). More concretely, if sales greater than 20 or less than 0, we set them at 20 or 0, which is also a method for getting rid of outliers.

Then we start extracting features. To predict what will happen on each (shop,item) pair, the most important ingredients are sales records in last one, two or maybe twelve months. Additionally, average sales among all kinds of items in the same shop, or other kinds of means, are also great features, which are named after "mean encoded features". Besides, "trend features", such as price trend, are also required for a better prediction. "Special features", like days in a month, or last sale of item in a shop, are also added into our feature set, which is shown by Table 1.

Actually we don't use all the features in Table 1. For the features with less importance(maybe less F score), we give up them to avoid overfitting.

3.2 Xgboost Regression

As we mentioned above, boosting tree is our main algorithm used for regression.We'd like to introduce the general thought of this algorithm first, and then we will introduce the concrete method we use in our project. As we know, a tree can be used for classification or regression, which is called classification and regression tree (CART). In the boosting algorithm, CART is used as the weak learning algorithm, and we use the sum of several CART to get a great proximity of the train data.

Formally, the model can be written as following:

$$f_m(x) = \sum_{i=1}^{m} T(x; \theta_i)$$

where the $f_m(x)$ is our model, $T(X; \theta_i)$ is the ith CART, and θ_i is the parameter of the ith CART. Our model is sum of several CART.

The learning process is a feedforward algorithm, in which each step we generate a new CART based on the previous trees. Formally, in m_{th} step, we need to calculate θ_m in the following way:

$$\theta_m = \arg\min_{\theta_m} \sum_{i=1}^{N} L(y_i, f_{m-1}(x_i) + T(x_i; \theta_m))$$

After we do this step-by-step, the model can achieve a highly accurate approximation to the training data.

Next, we will introduce the method in our project in detail. What we use is the xgbregressor algorithm of xgboost. The general idea is quite the same as above-mentioned. In each step, our aim is to minimize the risk function as following by calculate θ_m:

$$obj_m = \sum_{i=1}^{N} L(y_i, f_{m-1}(x_i) + T(x_i; \theta_m)) + \Omega(\theta_m)$$

where $\Omega(\theta_m)$ is the regular term.

The function of CART can be rewrited in the following way:

$$T(x_i; \theta_m) = w_{mq_m(x_i)}, q_m(x) \in \{1, 2, ..., d_m\}$$

where the $q_m(x_i)$ is the index of w_m, and w_{mj} is the value for x that belongs to j_{th} region that divided by the tree. We take squired loss function as our loss function and L2 norm as regular term:

$$\begin{aligned} L(y_i, f_{m-1}(x_i) + T(x_i; \theta_m)) &= (y_i - f_{m-1}(x_i) - T(x_i; \theta_m))^2 \\ &= 2(f_{m-1}(x_i) - y_i) w_{mq_m(x_i)} + w^2_{mq_m(x_i)} + \text{const} \\ \Omega(\theta_m) &= \frac{1}{2}\lambda \sum_{j=1}^{d_m} w^2_{mj} \end{aligned} \tag{1}$$

where λ is the parameter of regular term. We define the set of index of x belong to a certain leaf of the tree, formally:

$$I_{mj} = \{i | q_m(x_i) = j\}$$

Thus our risk function takes the following form:

$$\begin{aligned} obj_m &= \sum_{j=1}^{d_m} [\sum_{i \in I_{mj}} 2(f_{m-1}(x_i) - y_i) w_{mj} + \sum_{i \in I_{mj}} w^2_{mj}] + \frac{1}{2}\lambda \sum_{j=1}^{d_m} w^2_{mj} \\ &= \sum_{j=1}^{d_m} [\sum_{i \in I_{mj}} 2(f_{m-1}(x_i) - y_i) w_{mj} + (\sum_{j=1}^{d_m} 1 + \frac{1}{2}\lambda) w^2_{mj}] \\ &= \sum_{j=1}^{d_m} [G_{mj} w_{mj} + \frac{1}{2}(H_{mj} + \lambda) w^2_{mj}] \end{aligned} \tag{2}$$

where

$$
\begin{aligned}
G_{mj} &= \sum_{i \in I_{mj}} 2(f_{m-1}(x_i) - y_i) \\
H_{mj} &= 2 \sum_{i \in I_{mj}} 1
\end{aligned}
\tag{3}
$$

Notice that the risk function is sum of quadratic functions, we can easily get the minimum value:

$$
\begin{aligned}
\min obj_m &= -\frac{1}{2} \sum_{j=1}^{d_m} \frac{G_{mj}^2}{H_{mj} + \lambda} \\
\arg\min_{w_{mj}} obj_m &= -\frac{G_{mj}}{H_{mj} + \lambda}
\end{aligned}
\tag{4}
$$

The learning process will try to generate a CART that can minimal the risk function in m_{th} step. The algorithm we use is a greedy algorithm, in which step we try to divide a leaf node of the tree and decrease the risk function most. If we divide a leaf node into two child nodes, the change of the risk function can be as following:

$$
\Delta = -\frac{1}{2}\left(\frac{G_{mj1}^2}{H_{mj1} + \lambda} + \frac{G_{mj2}^2}{H_{mj2} + \lambda} - \frac{G_{mj}^2}{H_{mj} + \lambda}\right)
$$

And our aim is to minimize Δ. For CART, this can be solved by scan the value of this node linearly to find the best cut point. And to do this repeatedly, we can generate a CART for m_{th} step.

4 Experiment

4.1 Feature Selection

In Table 1 the maximum lag is 12, which means we have to give up the first 12 months' data for there is no record to calculate features before them. It's a big loss, so we tend to set maximum lag to 3. In another word, more data, less features, only experiment can tell us which is better. The results on test set indicate that, lag 1,2,3,6 is better than 1,2,3,6,12 and 1,2,3, so we apply lag 1,2,3,6.

Figure 2 shows the importance of all the considered features represented by F scores. As we can see, all the features including 12 make little difference to helping connect data information and the model. So, we abandon those with little importance and utilize the rest features to help make predictions.

4.2 Parameter Adjustment

There are many parameters in xgboost to be adjusted, including **eta, min child weight, max depth, subsample, colsample bytree** and so on, among which the most important and decisive ones to the behavior of the model are **min child weight, max depth and subsample**, so in order to get the most remarkable performance, we have the others fixed, and at each time, we only adjust one kind of parameter of the three. Specifically, we fix **colsample bytree** as 0.8, **eta** as 0.2, **n estimators** as 1000, and **seed** as 42.

The first parameter we choose to optimize is **max depth**, which is the depth of the regression tree in the model, to avoid overfitting. The larger this value is, the more specific samples the model will learn. The default value is 8, and we've tried 10,14,16,17,22,24. To our disappointment, there is no obvious trend of how the validation RMSE scores change with this value. Of all these values we've tried, 24 behaves the best, so we fix it in the next two experiments. More details can be seen from table 2.

The second parameter we choose is **min child weight**, which is a constrain on the least sum of the weights of the leaf nodes. The same as **max depth**, it also aims to aviod overfitting, and with larger

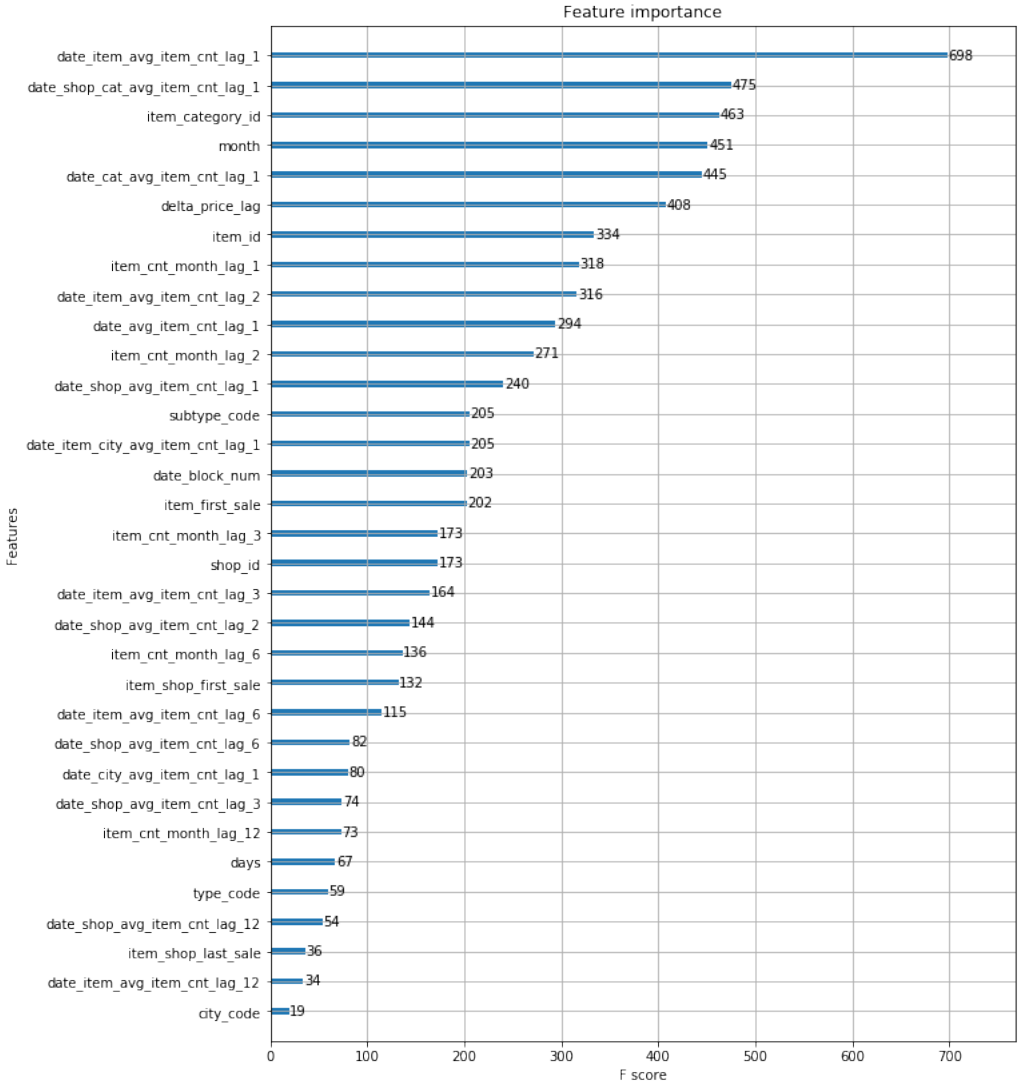

Figure 2: Feature importance

value, the model will ignore more local special samples. The default setting is 300, which we think is too large to fit well, so we gradually decrease it to help improve performance. Result are shown in table 3. We've tried 200,110,100,98,95,90,80,50, among which 110 seems to perform best, but in fact on test sets, 100 performs best. From our perspective, it's because the validation set is so small and special that the scores on the validation set cannot accord well with the test set. Finally we choose 100 and fix it in the last experiment.

At last we try to optimize **sumsample**, which controls the proportion of stochastic sampling. If we decrease this value, the model prefers to be more conservative, with a risk of being underfitting if too small. The default is 0.8, and we've tried 0.7,0.75,0.82,0.85, among which 0.82 performs the best and makes remarkable progress comparing to the others, however, on test set 0.8 gets the best. The reason might be the same as we've analysed about parameter **min child weight**. The results are shown in table 4.

5 Conclusion

In this project, we just simply use existent kernel and carefully select features which have been extracted. Another work is to adjust several parameters, by which we boost the score from about

Table 2: Parameter **max depth**

max depth	best validation RMSE
8	0.90684
9	0.904977
10	0.904206
14	0.898828
16	0.898913
17	0.900075
22	0.896748
24	**0.895897**

Table 3: Parameter **min child weight**

min child weight	best validation RMSE
200	0.895897
100	0.895482
80	0.892255
50	0.894176
110	**0.89054**
95	0.892329
90	0.89302
98	0.892586

Table 4: Parameter **subsample**

subsample	best validation RMSE
0.8	0.895482
0.7	0.892929
0.75	0.895054
0.85	0.894227
0.82	**0.891046**

0.90684 to 0.89432. This is a not bad improvement but not outstanding. The main fault is that we're nearly only considering parameters, not about the whole model, which results in a relatively weak improvement.

The future work should be focused on the model itself. Try to use more efficient models and methods, and make an ensemble to get a better performance on average.

References

[1] Denis Larionov. *Feature engineering, xgboost.* `https://www.kaggle.com/dlarionov/feature-engineering-xgboost`

[2] Bower, J.M. & Beeman, D. (1995) *The Book of GENESIS: Exploring Realistic Neural Models with the GEneral NEural SImulation System.* New York: TELOS/Springer–Verlag.

[3] Schapire, R. E. (1989). The strength of weak learnability. Machine Learning, 5(2), 197-227.

[4] Yoav Freund, Robert E. Schapire. (1995). A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting. Computational Learning Theory. Springer Berlin Heidelberg.