Predict Future Sales

1、Backgrounds

Sales prediction is important for companies which are entering new markets or are adding new services, products or which are experiencing high growth. A precise sales forecast is helpful for companies to balance marketing resources and sales based on their supply capacity. In this report, we forecast the total amount of products sold in every shop by a dataset consisting of daily historical data from January 2013 to October 2015, provided by one of the largest Russian software firms - 1C Company.We are trying to solve this task using different machine learning techniques, which has gained significant attention in the field of marketing in recent years.

2、DataInformation and Exploration

2.1、Data Overview

The data files downloaded on Kaggle are as follows:

Data files

File_Name	Description
sales_train.csv	the training set. Daily data from January 2013 to October 2015.
items.csv	supplemental information about the items/products.
item_categories.csv	supplemental information about the items categories.
shops.csv	supplemental information about the shops.
test.csv	the test set. Our task is to forecast the sales for these shops and products for November 2015.

All data fields are as follows:

Data fields

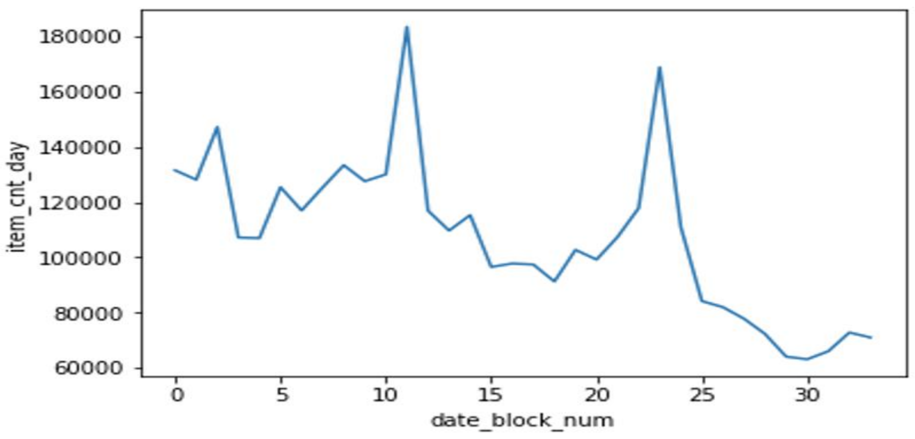
Data_Field_Name	Description
ID	an Id that represents a (Shop, Item) tuple within the test set
shop_id	unique identifier of a shop
item_id	unique identifier of a product
item_category_id	unique identifier of item category
item_cnt_day	number of products sold. You are predicting a monthly amount of this measure
item_price	current price of an item
date	date in format dd/mm/yyyy
date_block_num	a consecutive month number, used for convenience. January 2013 is 0, February 2013 is 1,..., October 2015 is 33
item_name	name of item
shop_name	name of shop
item_category_name	name of item category

2.2、Exploratory Data Analysis

We check the duplicated and missing values in the training dataset. There are only 6 duplicated lines. Given the large scale of the dataset, we choose to overlook them. Besides, the training dataset consists no missing values.

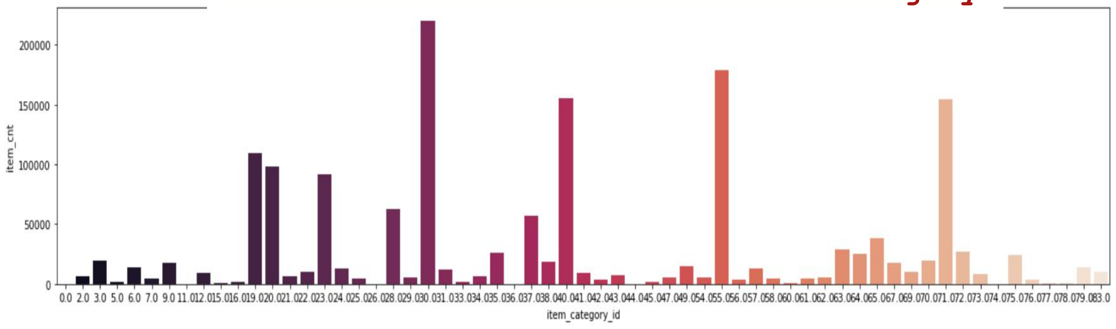
In order to gain some insights about the data, we attack from three angles: sum of all items sold per month, shops and item categories.

sum of all items sold per month

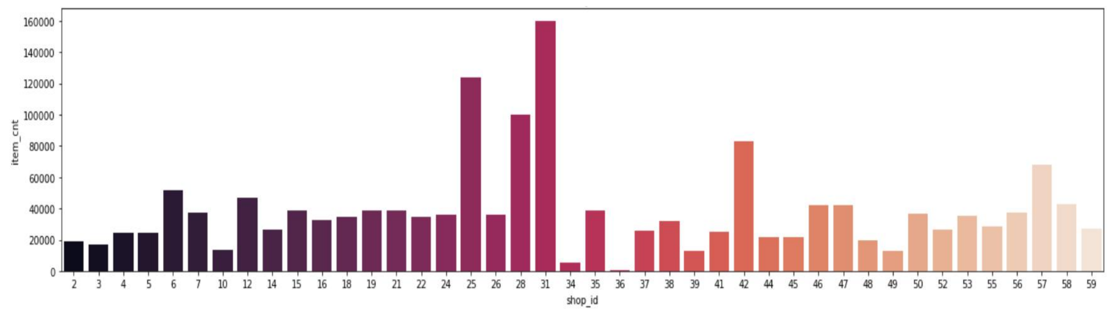


Clearly, there is a peak in sales at the end of the year, probably due to the holiday season. Therefore, it would be beneficial to add month and year, so that the model can pick up this pattern.

Sum of all items sold for each category



Sum of all items sold for each shop



We learn from the two plots above that only a very few item categories (out of 84) are sold in high numbers and not all 60 shops are in operations. Three shops are the main drivers. probably because these shops are Shopping malls or famous Retail stores. So it would be helpful to create features based on this information.

3、Feature Engineering

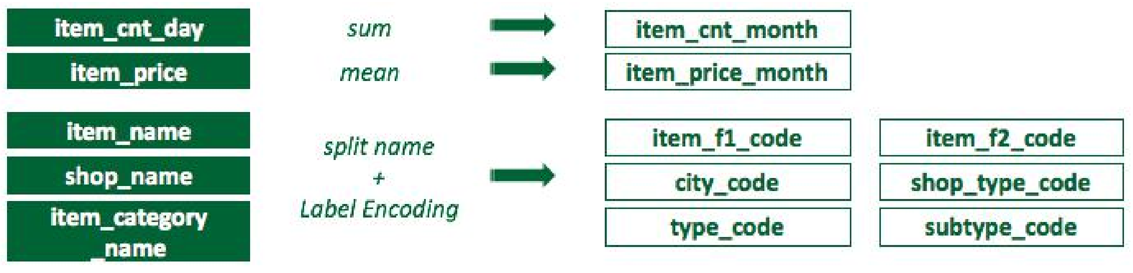
Since we'vecollected various discoveries through EDA, we can apply them into the feature engineering process. During this phase of work, it can be divided into five different parts: data preprocessing, split features & label encoding, lag features & mean encoding, dummy variable & other features and feature mergence.

We get started from doing data preprocessing by removing outliers, checking missing values, and filling out empty observations. In fact there are only a little work here. After that, we increment features in the following four ways.

3.1、Split Features & Label Encoding

The project aims to predict monthly sales, so the first feature incremented here is "item_cnt_month", which is the sum of the sales per day. For the same reason, daily price should be transferred into monthly mean price. As we know that there are subtle variations among different item names, shop names and category names, and they can be split into different parts, which have unique meanings for the prediction.

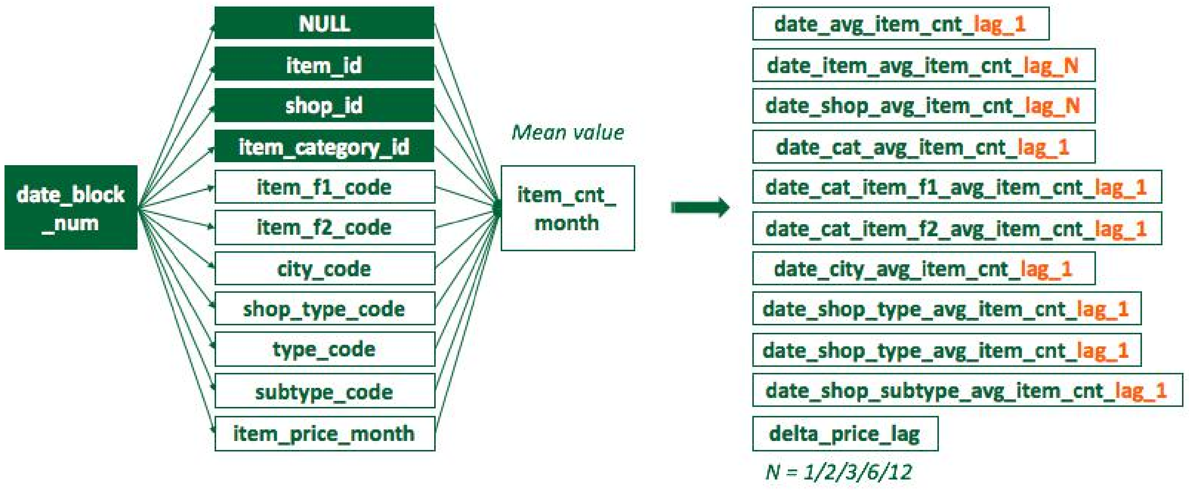
For instance, "shop_name" contains information about the city which the shop locates in, and the shop type of the city. After doing label encoding to these features, we can successfully increment 8 new features into our dataset, just like the picture shown as below:



3.2、Lag Features & Mean Encoding

When doing label encoding, it has some shortcomings. Since we have tem-million observations, the value of observations that have been encoded could have passive influences on the prediction result because they have real number meanings. To deal with the wide-scattered observations, we can process them with mean encoding. For instance, we can sum "item_cnt_month" of the same "data_block_num" and "city_code", then divide its frequency. We then assign the name "date_city_avg_item_cnt" to the new feature, and it can not only reduce the bias created by label encoding, but also reflect the business environment of the city that the shop locates in.

Also, to expand more information, we can introduce lag features to deal with time series data. As the picture shown below, we are able to create 19 new features in the end, and each one has real meanings contributing to our prediction result.


date_block_num
NULL
item_id
shop_id
item_category_id
item_f1_code
item_f2_code
city_code
shop_type_code
type_code
subtype_code
item_price_month
Mean value
item_cnt_month
date_avg_item_cnt_lag_1
date_item_avg_item_cnt_lag_N
date_shop_avg_item_cnt_lag_N
date_cat_avg_item_cnt_lag_1
date_cat_item_f1_avg_item_cnt_lag_1
date_cat_item_f2_avg_item_cnt_lag_1
date_city_avg_item_cnt_lag_1
date_shop_type_avg_item_cnt_lag_1
date_shop_type_avg_item_cnt_lag_1
date_shop_subtype_avg_item_cnt_lag_1
delta_price_lag
N = 1/2/3/6/12

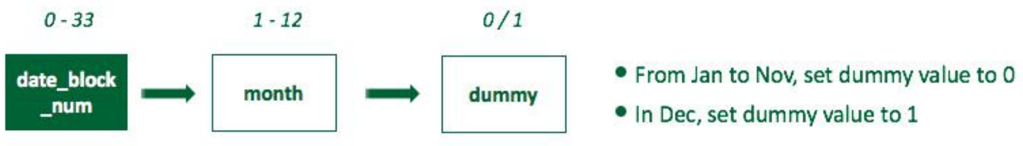
** Here are the real meanings:*

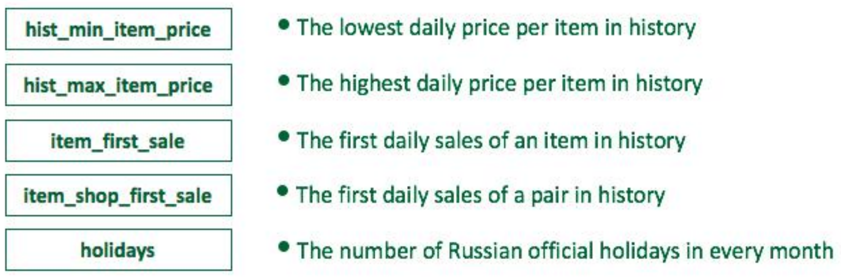
date_avg_item_cnt_lag_1: mean monthly sales for 1C company
date_item_avg_item_cnt_lag_N: mean monthly sales per ITEM
date_shop_avg_item_cnt_lag_N: mean monthly sales per SHOP
date_cat_avg_item_cnt_lag_1: mean monthly sales per ITEM-CATEGORY
date_cat_item_f1_avg_item_cnt_lag_1: mean monthly sales per ITEM-CATEGORY-F1
date_cat_item_f2_avg_item_cnt_lag_1: mean monthly sales per ITEM-CATEGORY-F2
date_shop_cat_avg_item_cnt_lag_1: mean monthly sales per SHOP per ITEM-CATEGORY
date_shop_type_avg_item_cnt_lag_1: mean monthly sales per SHOP-TYPE
date_shop_subtype_avg_item_cnt_lag_1: mean monthly sales per SHOP-SUBTYPE
date_city_avg_item_cnt_lag_1: mean monthly sales per CITY

3.3、Dummy Variables &Other Features

As we have noticed from EDA that every December has the highest sales of the year, we use dummy variable to deal with this cycle. At first, we can transfer"date_block_num" to "month", and then assign value 1 to December and 0 to the rest of eleven months.

Beside, we think that the lowest and highest price of an item in history, the first sales of an item can be a great reflection on new products' performances, so we incremented 3 new features here. Finally, we can independently introduce some external data, the number of holidays on each month, to help predict the seasonality of sales.

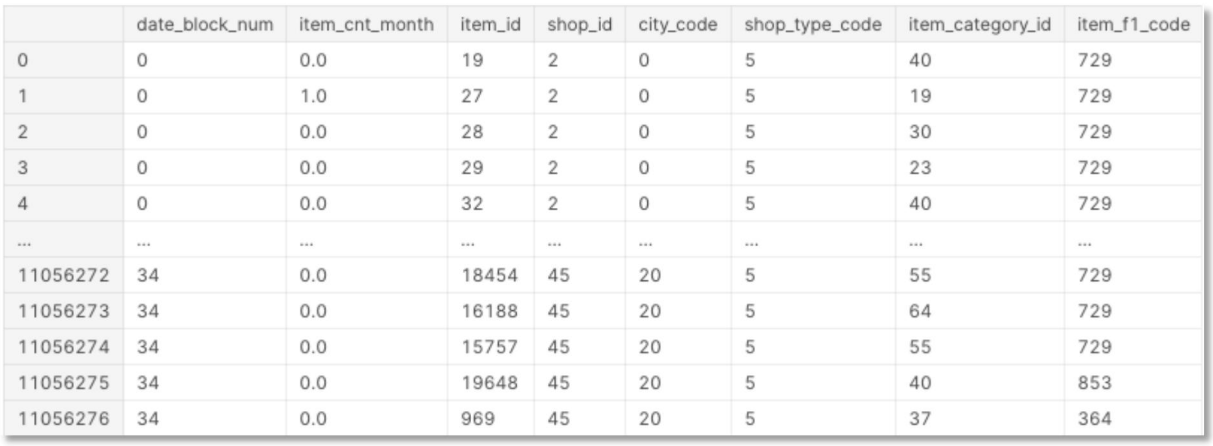

0 - 33
date_block
_num
1 - 12
month
0 / 1
dummy
From Jan to Nov, set dummy value to 0
In Dec, set dummy value to 1



hist_min_item_price	The lowest daily price per item in history
hist_max_item_price	The highest daily price per item in history
item_first_sale	The first daily sales of an item in history
item_shop_first_sale	The first daily sales of a pair in history
holidays	The number of Russian official holidays in every month

3.4、Feature Mergence

In the end, we can successfully increment 35 new features through all the work done above, merge them together, and enrich the original training set to a 11,056,276 rows plus 39 columns matrix. We believe that these features can greatly improve the prediction accuracy.



	date_block_num	item_cnt_month	item_id	shop_id	city_code	shop_type_code	item_category_id	item_f1_code
0	0	0.0	19	2	0	5	40	729
1	0	1.0	27	2	0	5	19	729
2	0	0.0	28	2	0	5	30	729
3	0	0.0	29	2	0	5	23	729
4	0	0.0	32	2	0	5	40	729
...	...	...	...	...	...	...	...	...
11056272	34	0.0	18454	45	20	5	55	729
11056273	34	0.0	16188	45	20	5	64	729
11056274	34	0.0	15757	45	20	5	55	729
11056275	34	0.0	19648	45	20	5	40	853
11056276	34	0.0	969	45	20	5	37	364

(part of the matrix)

4、 Methodology

4.1、Three Algorithms

We have employed three different algorithms:
Random Forest,lightGBMand**XGBoost.**

4.2、Results Evaluation

In this part, we choose Root Mean Square Error(RMSE) as evaluation metric for testing dataset.

To compare among three different algorithms, we build and train all three models with default parameters. And the results are shown below:

	Train_rmse	Valid_rmse	Test_rmse
Random Forest	0.83942	0.94997	1.00173
lightGBM	0.69266	**0.89392**	**0.91685**
XGBoost	0.68734	0.89816	0.91812

Among these three models, Random Forest plays worst with underfitting problem. LightGBM and XGBoost both work very well without overfitting or underfitting problems. However, LightGBM need less time and iterations for training model than XGBoost.

5、Parameter Tuning

XGBoost algorithm has become the ultimate weapon of many data scientist. It's a highly sophisticated algorithm, powerful enough to deal with all sorts of irregularities of data.

Building a model using XGBoost is easy. However, improving the model using XGBoost is difficult. This algorithm uses multiple parameters. To improve the model, parameter tuning is must. Hence, in this part we'll present the steps of parameter tuning along with some useful information about XGBoost's parameters.

5.1、Steps of parameter tuning for XGBoost with codes in Python:

Step 1. Fix number of estimators

In order to decide on boosting parameters, we need to set some initial values of other parameters. Therefore, we take the following values:

1. **learning_rate** = 0.1 : Generally a learning rate of 0.1 works but somewhere between 0.05 to 0.3 should work for different problems.
2. **max_depth** = 5 : This should be between 3-10. We've started with 5 but a different number can be chosen as well. 4-6 can be good starting points.
3. **min_child_weight** = 1 : We choose the default value for this parameter.
4. **gamma** = 0 : A smaller value like 0.1-0.2 can also be chosen for starting. This will anyways be tuned later.
5. **subsample, colsample_bytree** = 0.8 : This is a commonly used as start value. Typical values range between 0.5-0.9.

All the above settings are just initial estimates and will be tuned later. Then we check the optimum number of estimators using both cv function[1]and early-stopping method of XGBoost.

First, we use early-stopping method to find out the number of estimators until the model hasn't improved in 100 rounds.

Python code:

```
params = {'learning_rate': 0.1, 'max_depth': 5,
          'min_child_weight': 1, 'subsample': 0.8, 'colsample_bytree': 0.8,
          'gamma': 0, 'reg_alpha': 0, 'reg_lambda': 1, 'eval_metric':'rmse'}
watchlist = [(d_train, 'train'), (d_valid, 'valid')]
rgs = xgb.train(params, d_train, 10000, watchlist, early_stopping_rounds=100,
verbose_eval=10)
```

Result:

```
Stopping. Best iteration:[74]
```

Then we use GridSearchCV to find out the optimum number of estimators around 74.

Python code:

```
cv_params = {'n_estimators': [70,71,72,73,74]}
other_params={'objective':'reg:squarederror','learning_rate':0.1,
'n_estimators':500,'max_depth':5,'min_child_weight':1,
'subsample':0.8, 'colsample_bytree': 0.8,
                'gamma': 0,'reg_alpha': 0, 'reg_lambda': 1}
model = xgb.XGBRegressor(**other_params)
optimized_xgb=GridSearchCV(estimator=model,param_grid=cv_params,
              verbose=1,n_jobs=-1)
optimized_xgb.fit(x_train, y_train)
evalute_result = optimized_GBM.cv_results_
print('best value for the parameter: {0}'.format(optimized_xgb.best_params_))
```

Result:

```
best value for the parameter: {'n_estimators': 72}
```

Hence, we will set **72** as the value of **n_estimators** for tuning tree-based parameters in the following steps.

Step 2. Tune max_depth and min_child_weight
We tune these first as they will have the highest impact on model outcome. To start with, we set wide ranges: from 3 to 10 for max_depth and 1 to 6 for min_child_weight.

Python code:

```
cv_params = {'max_depth':range(3,11), 'min_child_weight':range(1,7)}
```

Result:

```
best value for the parameter: {'max_depth': 5, 'min_child_weight': 2}
```

Here, we have run 48 combinations with wide intervals between values. The ideal values are **5 for max_depth** and **2 for min_child_weight**.

Step 3.Tune gamma
Then we tune gamma value using the parameters already tuned above. Gamma can take various values but we only check for 5 values here.

Python code:

```
cv_params = {'gamma':[i/10.0 for i in range(0,5)]}
```

Result:

```
best value for the parameter: {'gamma': 0.0}
```

This shows that our original value of gamma, i.e. **0 is the optimum one**.

Step 4.Tune subsample and colsamle_bytree
The next step would be try different subsample and colsample_bytree values. Lets do this in 2 stages as well and take values 0.6,0.7,0.8,0.9 for both to start with.

Python code:
```
cv_params = {'subsample':[i/10.0 for i in range(6,10)],
             'colsample_bytree':[i/10.0 for i in range(6,10)]}
```

Result:
```
best value for the parameter: {'colsample_bytree': 0.8, 'subsample': 0.8}
```

Here, we find 0.8 as **the optimum value for both** subsample and colsample_bytree. Then we'll try values in 0.05 interval around these.

Python code:
```
cv_params = {'subsample':[i/100.0 for i in range(75,90,5)],
             'colsample_bytree':[i/100.0 for i in range(75,90,5)]}
```
Result:
```
best value for the parameter: {'colsample_bytree': 0.8, 'subsample': 0.8}
```

Again we get the same values as before. Thus the optimum values are 0.8 for both parameters.

Step 5.Tune regularization parameters
Next step is to apply regularization to reduce overfitting.

Python code:
```
cv_params = {'reg_alpha':[0,0.1,0.6,1,100],'reg_lambda':[0.1,0.3,0.6,1,100]}
```
Result:
```
best value for the parameter: {'reg_alpha': 0.6, 'reg_lambda': 0.3}
```

Then we fix alpha and try more values for lambda.

Python code:
```
cv_params = {'reg_lambda':[0.2,0.3,0.4,0.5]}
```
Result:
```
best value for the parameter: {'reg_lambda': 0.3}
```

Thus the optimum values are 0.6 for reg_alpha and 0.3 for reg_lambda.

Step 6.Reduce learning rate
Lastly, lower the learning rate and add more trees. Therefore, we use the cv function to do the job again.

Python code:
```
cv_params = {'learning_rate':[0.01,0.05,0.1,0.15,0.2,0.3]}
```
Result:
```
best value for the parameter: {'learning_rate': 0.01}
```

Then the final step is to re-calibrate the number of boosting rounds for the updated parameters by using early-stopping method.

Python code:

```
params = {'learning_rate': 0.01, 'max_depth': 5,
          'min_child_weight': 2, 'subsample': 0.8, 'colsample_bytree': 0.8,
          'gamma': 0, 'reg_alpha': 0.6, 'reg_lambda': 0.3, 'eval_metric':'rmse'}
watchlist = [(d_train, 'train'), (d_valid, 'valid')]
rgs = xgb.train(params, d_train, 10000, watchlist, early_stopping_rounds=100,
verbose_eval=10)
```

Result:

```
Stopping. Best iteration:
[850]   train-rmse:0.67492    valid-rmse:0.88371

test_rmse = 0.90912
```

6、Conclusion:

During the process of data analysis and preprocessing, we find that doing data analysis is of vital importance to the understanding of the data, which is the cornerstone of the building of the model. We also find that the data preprocessing can improve the performance of the model as the raw data can be quite confusing for the model to train.

We then focus on the parameters tuning for XGBoost algorithm and improve future sales prediction model based on them. Comparing with the model improvement by data preprocessing, parameter tuning does not feed back significant benefits which means parameter tuning can improve the performance of model but not a big leap. Besides, more efforts should be focus on feature engineering or other ways. For instance, it would be better if we crawl more specific features for the stores outside the dataset from Kaggle, like the weather information of the stores' location in the certain month. And we would like to do more work on this part in the future.