Pixabay

Using Gradient Boosting for Time Series prediction tasks

Easy Time series modelling

Rahul Agarwal Follow
Nov 17, 2019 · 7 min read ★

Time series prediction problems are pretty frequent in the retail domain.

Companies like Walmart and Target need to keep track of how much product should be shipped from Distribution Centres to stores. Even a small improvement in such a demand forecasting system can help save a lot of dollars in term of workforce management, inventory cost and out of stock loss.

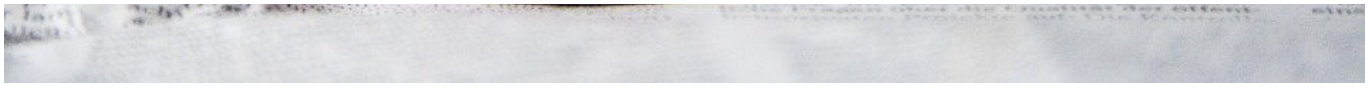
While there are many techniques to solve this particular problem like ARIMA, Prophet, and LSTMs, we can also treat such a problem as a regression problem too and use trees to solve it.

In this post, we will try to solve the time series problem using XGBoost.

The main things I am going to focus on are the sort of features such a setup takes and how to create such features.

. . .

Dataset



Kaggle master Kazanova along with some of his friends released a "How to win a data science competition" Coursera course. The Course involved a final project which itself was a time series prediction problem.

In this competition, we are given a challenging time-series dataset consisting of daily sales data, provided by one of the largest Russian software firms — 1C Company.

We have to predict total sales for every product and store in the next month.

Here is how the data looks like:

```
sales.head()
```

:

	date	date_block_num	shop_id	item_id	item_price	item_cnt_day
0	02.01.2013	0	59	22154	999.00	1.0
1	03.01.2013	0	25	2552	899.00	1.0
2	05.01.2013	0	25	2552	899.00	-1.0
3	06.01.2013	0	25	2554	1709.05	1.0
4	15.01.2013	0	25	2555	1099.00	1.0

We are given the data at a daily level, and we want to build a model which predicts total sales for every product and store in the next month.

The variable date_block_num is a consecutive month number, used for convenience. January 2013 is 0, and October 2015 is 33. You can think of it as a proxy to month variable. I think all the other variables are self-explanatory.

So how do we approach this sort of a problem?

. . .

Data Preparation

The main thing that I noticed is that the data preparation and feature generation aspect is by far the most important thing when we attempt to solve the time series problem using regression.

1. Do Basic EDA and remove outliers

```
sales = sales[sales['item_price']<100000]
sales = sales[sales['item_cnt_day']<=1000]
```

2. Group data at a level you want your predictions to be:

We start with creating a dataframe of distinct date_block_num, store and item combinations.

This is important because in the months we don't have a data for an item store combination, the machine learning algorithm needs to be told explicitly that the sales are zero.

```
from itertools import product
# Create "grid" with columns
index_cols = ['shop_id', 'item_id', 'date_block_num']

# For every month we create a grid from all shops/items combinations from that month
grid = []
for block_num in sales['date_block_num'].unique():
    cur_shops = sales.loc[sales['date_block_num'] == block_num, 'shop_id'].unique()
    cur_items = sales.loc[sales['date_block_num'] == block_num, 'item_id'].unique()
    grid.append(np.array(list(product(*[cur_shops, cur_items, [block_num]])),dtype='int32'))

grid = pd.DataFrame(np.vstack(grid), columns = index_cols,dtype=np.int32)
grid.head()
```

grid.py hosted with ♡ by GitHub view raw

```
grid
```

	shop_id	item_id	date_block_num
0	59	22154	0
1	59	2552	0
2	59	2554	0
3	59	2555	0
4	59	2564	0
...	...	...	...
10913845	21	7635	33
10913846	21	7638	33
10913847	21	7640	33
10913848	21	7632	33
10913849	21	7440	33

10913850 rows × 3 columns

The grid dataFrame contains all the shop, items and month combinations.

We then merge the Grid with Sales to get the monthly sales DataFrame. We also replace all the NA's with zero for months that didn't have any sales.

```
1  sales_m = sales.groupby(['date_block_num','shop_id','item_id']).agg({'item_cnt_day': 'sum','item_
2
```

```
# Merging sales numbers with the grid dataframe
sales_m = pd.merge(grid,sales_m,on=['date_block_num','shop_id','item_id'],how='left').fillna(0)

# adding the category id too from the items table.
sales_m = pd.merge(sales_m,items,on=['item_id'],how='left')
```

merge.py hosted with ♡ by GitHub view raw

```
sales_m.sample(5)
```

	shop_id	item_id	date_block_num	item_cnt_day	item_price	item_name	item_category_id
9689530	27	18260	28	0.0	0.0	СБ. Romantic Memories Sax Instrumental Hits ...	55
9976010	53	19981	29	0.0	0.0	Толстой Л.Н. Война и мир Роман-эпопея 4CD (...	43
6493710	45	11984	17	0.0	0.0	ИНДЮКИ: НАЗАД В БУДУЩЕЕ М/Ф (3D BD)	38
4745082	18	21552	12	0.0	0.0	ЦЕПЬ (Регион)	40
10105418	48	1325	30	0.0	0.0	ARMSTRONG LOUIS Hello Louis! 2CD	55

3. Create Target Encodings

To create target encodings, we group by a particular column and take the mean/min/sum etc. of the target column on it. These features are the first features we create in our model.

Please note that these features may induce a lot of leakage/overfitting in our system and thus we don't use them directly in our models. We will use the lag based version of these features in our models which we will create next.

```
groupcollist = ['item_id','shop_id','item_category_id']

aggregationlist = [('item_price',np.mean,'avg'),('item_cnt_day',np.sum,'sum'),('item_cnt_day',np

for type_id in groupcollist:
    for column_id,aggregator,aggtype in aggregationlist:
        # get numbers from sales data and set column names
        mean_df = sales_m.groupby([type_id,'date_block_num']).aggregate(aggregator).reset_index(
        mean_df.columns = [type_id+'_'+aggtype+'_'+column_id,type_id,'date_block_num']
        # merge new columns on sales_m data
        sales_m = pd.merge(sales_m,mean_df,on=['date_block_num',type_id],how='left')
```

target_encode.py hosted with ♡ by GitHub view raw

We group by `item_id`, `shop_id`, and `item_category_id` and aggregate on the `item_price` and `item_cnt_day` column to create the following new features:

```
sales_m.columns
```

:

```
Index(['shop_id', 'item_id', 'date_block_num', 'item_cnt_day', 'item_price',
       'item_name', 'item_category_id', 'item_id_avg_item_price',
       'item_id_sum_item_cnt_day', 'item_id_avg_item_cnt_day',
       'shop_id_avg_item_price', 'shop_id_sum_item_cnt_day',
       'shop_id_avg_item_cnt_day', 'item_category_id_avg_item_price',
       'item_category_id_sum_item_cnt_day',
       'item_category_id_avg_item_cnt_day'],
      dtype='object')
```

We create the highlighted target encodings

We could also have used featuretools for this. **Featuretools** is a framework to perform automated feature engineering. It excels at transforming temporal and relational datasets into feature matrices for machine learning.

4. Create Lag Features

The next set of features our model needs are the lag based Features.

When we create regular classification models, we treat training examples as fairly independent of each other. But in case of time series problems, at any point in time, the model needs information on what happened in the past.

We can't do this for all the past days, but we can provide the models with the most recent information nonetheless using our target encoded features.

```
lag_variables  = ['item_id_avg_item_price','item_id_sum_item_cnt_day','item_id_avg_item_cnt_day'
lags = [1 ,2 ,3 ,4, 5, 12]
# we will keep the results in thsi dataframe
sales_means = sales_m.copy()
for lag in lags:
    sales_new_df = sales_m.copy()
    sales_new_df.date_block_num+=lag
```

```
    # subset only the lag variables we want
    sales_new_df = sales_new_df[['date_block_num','shop_id','item_id']+lag_variables]
    sales_new_df.columns = ['date_block_num','shop_id','item_id']+ [lag_feat+'_lag_'+str(lag) fo
    # join with date_block_num,shop_id and item_id
    sales_means = pd.merge(sales_means, sales_new_df,on=['date_block_num','shop_id','item_id'] ,
```

lag_feats.py hosted with ♡ by GitHub view raw

So we aim to add past information for a few features in our data. We do it for all the new features we created and the `item_cnt_day` feature.

We fill the NA's with zeros once we have the lag features.

```
for feat in sales_means.columns:
    if 'item_cnt' in feat:
        sales_means[feat]=sales_means[feat].fillna(0)
    elif 'item_price' in feat:
    sales_means[feat]=sales_means[feat].fillna(sales_means[feat].median())
```

fillna.py hosted with ♡ by GitHub view raw

We end up creating a lot of lag features with different lags:

```
'item_id_avg_item_price_lag_1','item_id_sum_item_cnt_day_lag_1',
'item_id_avg_item_cnt_day_lag_1','shop_id_avg_item_price_lag_1',
'shop_id_sum_item_cnt_day_lag_1','shop_id_avg_item_cnt_day_lag_1','it
em_category_id_avg_item_price_lag_1','item_category_id_sum_item_cnt_d
ay_lag_1','item_category_id_avg_item_cnt_day_lag_1',
'item_cnt_day_lag_1',

'item_id_avg_item_price_lag_2',
'item_id_sum_item_cnt_day_lag_2','item_id_avg_item_cnt_day_lag_2',
'shop_id_avg_item_price_lag_2','shop_id_sum_item_cnt_day_lag_2',
'shop_id_avg_item_cnt_day_lag_2','item_category_id_avg_item_price_lag
_2','item_category_id_sum_item_cnt_day_lag_2','item_category_id_avg_i
tem_cnt_day_lag_2', 'item_cnt_day_lag_2',

...
```

• • •

Modelling

1. Drop the unrequired columns

As previously said, we are going to drop the target encoded features as they might induce a lot of overfitting in the model. We also lose the item_name and item_price feature.

```
cols_to_drop = lag_variables[:-1] + ['item_name','item_price']

for col in cols_to_drop:
    del sales_means[col]
```

drop_columns.py hosted with ♡ by GitHub view raw

2. Take a recent bit of data only

When we created the lag variables, we induced a lot of zeroes in the system. We used the maximum lag as 12. To counter that we remove the first 12 months indexes.

```
sales_means = sales_means[sales_means['date_block_num']>11]
```

3. Train and CV Split

When we do a time series split, we usually don't take a cross-sectional split as the data is time-dependent. We want to create a model that sees till now and can predict the next month well.

```
X_train = sales_means[sales_means['date_block_num']<33]
X_cv =  sales_means[sales_means['date_block_num']==33]

Y_train = X_train['item_cnt_day']
Y_cv = X_cv['item_cnt_day']

del X_train['item_cnt_day']
del X_cv['item_cnt_day']
```

4. Create Baseline

Before we proceed with modelling steps, lets check the RMSE of a naive model, as we want to have an RMSE to compare to. We assume that we are going to predict the last month sales as current month sale for our baseline model. We can quantify the performance of our model using this baseline RMSE.

```
from sklearn.metrics import mean_squared_error
sales_m_test = sales_m[sales_m['date_block_num']==33]

preds = sales_m.copy()
preds['date_block_num']=preds['date_block_num']+1
preds = preds[preds['date_block_num']==33]
preds = preds.rename(columns={'item_cnt_day':'preds_item_cnt_day'})
preds = pd.merge(sales_m_test,preds,on = ['shop_id','item_id'],how='left')[['shop_id','item_id',

# We want our predictions clipped at (0,20). Competition Specific
preds['item_cnt_day'] = preds['item_cnt_day'].clip(0,20)
preds['preds_item_cnt_day'] = preds['preds_item_cnt_day'].clip(0,20)
baseline_rmse = np.sqrt(mean_squared_error(preds['item_cnt_day'],preds['preds_item_cnt_day']))
```

```
print(baseline_rmse)
```

baseline.py hosted with ♡ by GitHub view raw

```
1.1358170090812756
```

5. Train XGB

We use the XGBRegressor object from the `xgboost` scikit API to build our model. Parameters are taken from this kaggle kernel. If you have time, you can use hyperopt to automatically find out the hyperparameters yourself.

```
from xgboost import XGBRegressor

model = XGBRegressor(
    max_depth=8,
    n_estimators=1000,
    min_child_weight=300,
    colsample_bytree=0.8,
    subsample=0.8,
    eta=0.3,
    seed=42)

model.fit(
    X_train,
    Y_train,
    eval_metric="rmse",
    eval_set=[(X_train, Y_train), (X_cv, Y_cv)],
    verbose=True,
    early_stopping_rounds = 10)
```

```
Stopping. Best iteration:
[131]   validation_0-rmse:0.856322      validation_1-rmse:0.928771
```

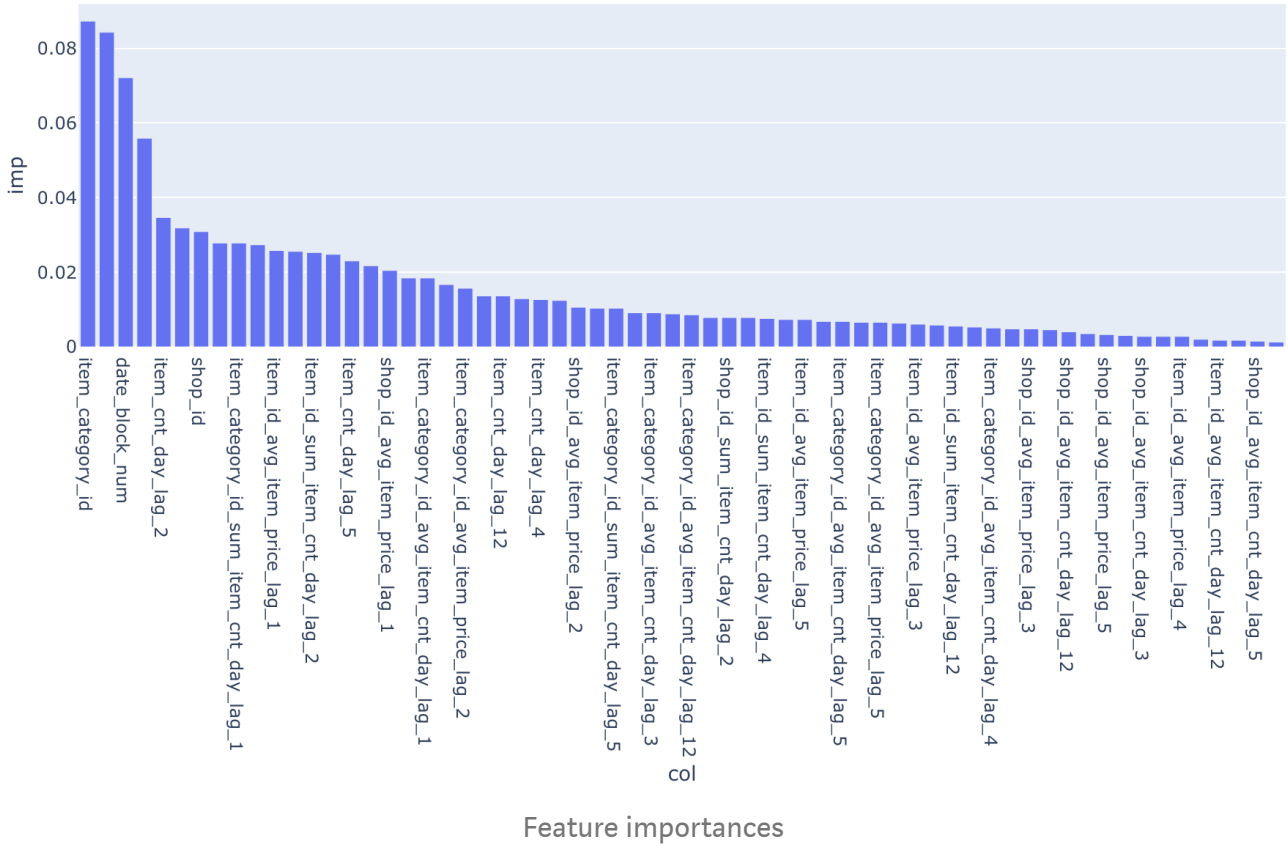
After running this, we can see RMSE in ranges of ***0.93*** on the CV set. And that is pretty impressive based on our baseline validation RMSE of ***1.13***. And so we work on deploying this model as part of our continuous integration effort.

5. Plot Feature Importance

We can also see the important features that come from XGB.

```
feature_importances = pd.DataFrame({'col': columns,'imp':model.feature_importances_})
feature_importances = feature_importances.sort_values(by='imp',ascending=False)
px.bar(feature_importances,x='col',y='imp')
```

plot_importance.py hosted with ♡ by GitHub view raw



. . .

Conclusion

In this post, we talked about how we can use trees for even time series modelling. The purpose was not to get perfect scores on the kaggle leaderboard but to gain an understanding of how such models work.

When I took part in this competition as part of the course, a couple of years back, using trees I reached near the top of the leaderboard.

Over time people have worked a lot on tweaking the model, hyperparameter tuning and creating even more informative features. But the basic approach has remained the same.

You can find the whole running code on GitHub.

. . .

Take a look at the How to Win a Data Science Competition: Learn from Top Kagglers

course in the Advanced machine learning specialization by Kazanova. This course talks about a lot of ways to improve your models using feature engineering and hyperparameter tuning.

I am going to be writing more beginner-friendly posts in the future too. Let me know what you think about the series. Follow me up at **Medium** or Subscribe to my **blog** to be informed about them. As always, I welcome feedback and constructive criticism and can be reached on Twitter @mlwhiz.

Also, a small disclaimer — There might be some affiliate links in this post to relevant resources, as sharing knowledge is never a bad idea.

Machine Learning | Data Science | Programming | Artificial Intelligence | Technology